Simulating DB2 Buffer Pool Management

Wenguang Wang Richard B. Bunt
Department of Computer Science
University of Saskatchewan
57 Campus Drive
Saskatoon, SK S7N 5A9
Canada
Email: {wew036, bunt}@cs.usask.ca

Abstract

Storage management is an important part of any DBMS. The buffer pool in DB2 is used to cache the disk pages of the database, and its management algorithm can significantly affect performance. In order to investigate performance issues relating to the buffer pool management algorithm, a trace of buffer pool requests was collected and a trace-driven simulator was developed so that the impact of various parameters of the buffer pool management algorithm could be investigated under controlled circumstances. Relationships among different activities competing for storage space and the I/O channel are examined, and some issues affecting performance are identified.

1 Introduction

In a Database Management System (DBMS), a buffer pool is used to cache the disk pages of the database. Because the speed gap between disk and memory is large, the effectiveness of the buffer pool management algorithm is very important to the performance of the DBMS. Tuning such an algorithm is a complex task because of the number of parameters involved and the complexity of their interactions. It requires a detailed understanding of the nature of activities that compete for the resources being managed, including storage space and the I/O channel. Because of the complexity of a commercial DBMS, however, it is often difficult to analyze the buffer pool algorithm, or implement a new one and test it directly. Simulation provides an effective alternative.

For our research, a blend of direct experimentation and trace-driven simulation is being used in a detailed study of buffer pool management in DB2, IBM's popular commercial relational DBMS. The TPC-C [7] benchmark provides the workload. A detailed buffer pool simulator has been developed for DB2 and verified. To provide a realistic reproducible workload for the simulation experiments, a trace of the buffer pool requests when running the TPC-C benchmark was captured. The effects of parameters of the buffer pool algorithm are examined by simulation. Activities of the I/O channel in the buffer pool are investigated as well. The purpose of this paper is to present our simulator and give some examples of the insights it has made possible.

This paper is organized as follows: §2 gives a brief review of some previous work relating to database buffer pool management; §3 describes the buffer pool management algorithm used in DB2; §4 presents the trace collection method and the trace formats; §5 discusses the simulator and its verification; §6 presents the results of the experiments conducted to study the relationships among buffer pool I/O activities and their impacts on the buffer pool performance; §7 presents conclusions and future work.

2 Related Work

It has been observed [5, 3, 6] that the references to the disk pages of a database have locality. Furthermore, this locality of reference is more regular, predictable, and exploitable than the localized reference activity found in programs in general. For this reason, a buffer pool is employed by a DBMS as a cache of disk pages. Various database buffer pool management algorithms, including LRU, FIFO, CLOCK, LRD, DGCLOCK, and Working Set [1], are analyzed and compared in [2]. A set of flexible predictive buffer management algorithms was proposed in [4].

All approaches aim to improve the hit ratio of the buffer pool in order to reduce disk reads. However, another problem is also important—how and when to clean out dirty pages (i.e. writing back to disk pages that have been changed since being fetched into the pool). Two approaches can be applied: page cleaning on demand and asynchronous page cleaning. In page cleaning on demand, a dirty page will be cleaned only when it is selected for replacement. In asynchronous page cleaning, dirty pages are cleaned asynchronously before they are selected for replacement. In traditional virtual memory systems, processes write only to their data segments, and normally not many dirty pages are generated. However, in a database buffer pool, transaction-based workloads may generate so many dirty pages that page cleaning on demand is not efficient. The effect of the asynchronous page cleaning strategy is investigated in this paper, and the approach proposed can be applied to all algorithms mentioned above.

3 The DB2 Buffer Pool Management Algorithm

The buffer pool management algorithm contains several parts: fetching, placement, and replacement. The fetching algorithm brings new pages into the buffer pool. Normally, fetching on demand is used, but when there are sequential accesses to pages, prefetching can be a helpful supplement to fetching on demand. DB2 supports both fetching on demand and prefetching. For this analysis, only fetching on demand is used because there is no sequential access in the TPC-C workload. The placement algorithm defines how to place a page into the free pool of the buffer. The replacement algorithm defines how to maintain the free pool of the buffer. Replacement on demand makes a free space only when needed. Pre-replacement tries to always keep spaces available by writing back to disk dirty pages that are deemed not to be needed. Traditional buffer management algorithms use only replacement on demand, but both approaches are used in DB2 buffer pool management. By performing page cleaning, the buffer pool maintains some space holding clean pages that can be used immediately for new pages.

3.1 Basic Replacement Algorithm

Unlike traditional virtual memory management, there is no hardware support in the buffer pool to access a page that is not in memory. Therefore, a *fix/unfix* mechanism is used. When the DBMS needs to access a page, it will send a *fix* request to the buffer pool. If the page is already in the buffer pool, no physical I/O is needed; otherwise, it is read into the buffer pool from disk. The DBMS can access this page freely from the buffer pool and it cannot be evicted from memory after the fix. When the DBMS finishes using this page, an *unfix* request is sent to the buffer pool. After the unfix, this page is allowed to be evicted when needed.

The buffer pool of DB2 is managed using a variant of the CLOCK algorithm.

3.2 Page Cleaning Algorithm

When a page is selected for replacement, its status is checked. If it is clean (unchanged), its space can be used immediately. If it is dirty, a *synchronous* write must be performed to clean this page. Synchronous writes negatively impact the throughput of the system because the space occupied by this dirty page cannot be used until the synchronous write completes.

To alleviate this problem, one or more *page cleaners* are used to clean out pages asynchronously before synchronous writes are needed. At the beginning, all page cleaners are sleeping. Three kinds of events can wake up the page cleaners:

- *Exceeding the Dirty Page Threshold.* The dirty page threshold is a configurable parameter which indicates the desired percentage of dirty pages in the buffer pool. Page cleaners will be awakened when the percentage of dirty pages exceeds the threshold. The default value of the threshold is 60%.
- *Dirty Replacement.* When a dirty page is selected from the buffer pool for replacement, i.e. a synchronous write occurs, the page cleaners wake up.
- *Exceeding the softmax value.* When the percentage of changes recorded in the log file exceeds the softmax value, the page cleaners wake up.

When a page cleaner wakes up, it collects some dirty pages and writes them back to the disks. If the condition for page cleaning still exists, the page cleaner will begin another round of collecting and cleaning. If the conditions for page cleaning do not exist, the page cleaner will sleep. When a page cleaner writes pages, the clients do not need to wait for the writes directly. Therefore, these writes are called asynchronous writes.

Figure 1 shows the I/O activities between the buffer pool and the disks. The buffer pool can be considered as two regions: the *clean region* which contains clean pages, and the *dirty region* which contains dirty pages. When the applications are running, many pages are read from the disks. At the same time, many pages in the buffer pool are changed, which causes the dirty region to expand. The synchronous writes and asynchronous writes write the dirty pages back to the disks which causes the dirty region to shrink. When the system is in stable state, the number of pages changed is equal to the number of pages cleaned so that a constant supply of free pages is available.

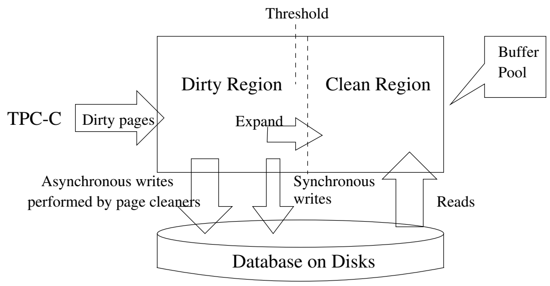

Figure 1: Buffer Pool and Its I/O Activities with Disks

4 Trace Collection

4.1 Workload – TPC-C Benchmark

The TPC-C benchmark is used to provide the workload to the simulator and to provide the "real" system against which simulation results are compared. This benchmark is among a group of benchmarks defined by the Transaction Processing performance Council[1] (TPC). The TPC-C Benchmark is an On-Line Transaction Processing (OLTP) workload. It is a mixture of read-only and update-intensive transactions that simulate the activities found in complex OLTP application environments. Multiple transactions are used to simulate the business activity of processing an order, and each transaction is subject to a response time constraint.

There are five kinds of transactions in the TPC-C benchmark, and these are listed in Table 1. The percentage column shows their respective percentages in the total number of transactions. For example, 45% transactions are New-Order transactions. Because the New-Order transaction is the backbone of the workload, the performance of the TPC-C benchmark is expressed in Transactions Per Minute (TPM), which is defined as the number of New-Order transactions processed per minute. The size of the TPC-C database is given by the

[1] `http://www.tpc.org/`

number of "warehouses" defined. There are about 100M bytes of data for one warehouse.

Table 1: TPC-C Transactions

Name	Percentage
New-Order	45%
Payment	43%
Order-Status	4%
Delivery	4%
Stock-Level	4%

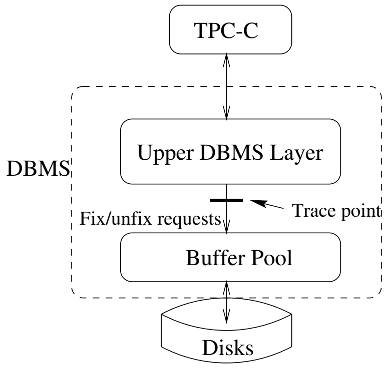

Figure 2: Trace collection structure

4.2 Trace Collection and Contents

In order to collect the trace of the buffer pool activities needed by the simulator, a TPC-C database with 50 warehouses was created. The benchmark runs on a PC Server running Windows NT Server 4.0 in the Distributed Systems Performance Laboratory at the University of Saskatchewan. The DB2 version used is 6.1. The TPC-C database is installed on 9 disks.

When the TPC-C benchmark is running, transactions are sent to the DBMS. The DBMS executes the transactions and requests disk pages from the buffer pool. All fix/unfix requests to the buffer pool are needed for the simulation. The information of every request going through the trace point shown in Figure 2 is recorded in a trace buffer in memory by the trace tool. Because the trace buffer cannot hold all the trace information needed for the simulator at one time, the TPC-C program was changed so that it suspends before the trace buffer is full, dumps the trace into a file, and continues to run. By using this technique, arbitrarily large traces can be obtained.

Because of the overhead associated with this data collection, the system runs more slowly when recording the trace. Therefore, the sequences of the aggregated trace records from all agents may change, but the sequence for each agent does not change. The trace used by the simulator is separated based on the agent ID. Therefore, although the aggregated trace sequence changes, it will not affect the simulation result. The begin and end time of each transaction is recorded in the TPC-C program. This transaction information is inserted into the trace recorded from the buffer pool sorted by the timestamps. After these processes, the trace can be used by the simulator. The trace used in these experiments includes about 60 million buffer pool requests from 200 thousand transactions.

Necessary information about fix and unfix requests is recorded in the trace. Table 2 shows the important fields of a trace record. The *fix mode* indicates the type of fix request. There are two kinds of fix: *exclusive* and *shared*. Any page can have at most one exclusive fix but can have multiple shared fixes at the same time.

5 The Buffer Pool Simulator

The buffer pool simulator simulates DB2's buffer pool management algorithm and the disk subsystem. This simulator contains about 8000 lines of C++ code. An event-driven architecture is used in the simulator. Different components of the simulator communicate through events. Figure 3 shows the components and main event types.

There are four basic components in the simulator.

- The *Buffer Pool Manager* contains the basic buffer pool management algorithm. It manages the placement and replacement of the buffer pool pages. It accepts the fix/unfix events, and sends out disk I/O events (Read and SyncWrite). It also sends StartCleaning events to trigger Page

Table 2: Important Fields of the Trace

Field	Value
type	The type (fix or unfix) of this record
tid	The number to identify requests from different clients
object type	Can be INDEX or DATA
object id	The number to identify the table this page belongs to
page number	The disk address of the page
fix mode (only for fix)	Can be EXCLUSIVE or SHARE
modify flag (only for unfix)	Indicates whether or not this page has been modified

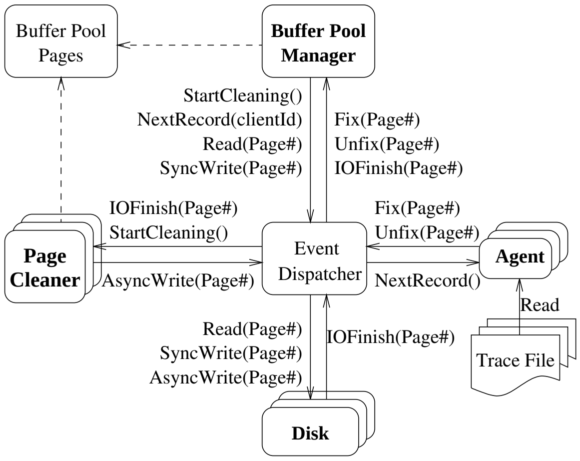

Figure 3: The Structure of the Simulator

Cleaners to start page cleaning.

- The *Page Cleaner* manages the page cleaning of the buffer pool. It accepts StartCleaning events and performs page cleaning on the dirty pages of the buffer pool. There can be more than one page cleaners in the simulator. Both the Buffer Pool Manager and the Page Cleaner can access *Buffer Pool Pages* which is the data structure holding all the pages of the buffer pool.
- An *Agent* represents a client that sends out requests to the Buffer Pool. Requests belong to each client are organized into a separated *Trace File*. Therefore, the Agent simply reads a record (either fix or unfix) from its trace file and returns it to the Buffer Pool Manager. The number of Agents is the same as the number of clients when running TPC-C.
- The *Disk* accepts disk I/O events (Read, SyncWrite, and AsyncWrite) and return IOFinish events. Because the track and sector information of disk pages is not available from the trace, it is assumed that a disk has constant read and write time. Unprocessed requests to a disk are queued and served in a first-in-first-out order. When there are multiple disks in the simulator, different disks can perform reads and writes simultaneously.

A timestamp is associated with each event. All events will be sent to the event queue and sorted by their timestamps. The *Event Dispatcher* selects the event with the minimum timestamp and sends it to the corresponding component. This is a typical event flow when fixing a page: a fix event is read from the Agent and sent to the Buffer Pool Manager. If the page is in the buffer pool, the fix finishes, and the Buffer Pool Manager sends a NextRecord event to the Agent for the next record. If the page is not in the buffer pool, and a clean page is found for the replacement, a Read event will be sent to the Disk, and when the Buffer Pool Manager receives an IOFinish event, it will send a NextRecord event to the Agent asking for the next record.

All times in the simulator are relative, with the crucial activities shown in Table 3. This set of values was derived experimentally in order to give realistic behaviour. By comparing simulator results with results from our experimental DB2 testbed it was determined that roughly 40 million units of simulator time corresponds to 1 minute of real time on the DB2 testbed. This means, for example, that a disk operation on our system requires approximately 9 msec, a reasonable value for our configuration. These time parameters can be changed to reflect different configurations.

Table 3: Time Parameters in the Simulator

CPU operation time	
FixTime	20
UnfixTime	16
GatherCleanPage	14
ListReadTime	1
BufferpoolReadTime	1
ReplacementTime	4
Disk operation time	
ReadTime	6000
WriteTime	6000
AsynchronousWriteTime	6000

In real life, a buffer pool of a DBMS is usually IO bound. Therefore, the CPU subsystem is not part of the simulator in order to reduce the complexity. This could be a defect when the CPU overhead is high (e.g. the number of users is high), but it is sufficient for the current investigations.

5.1 Performance Measures

In order to determine the behaviour of the buffer pool management algorithm, various quantities related to page activity, throughput and the I/O channel are measured. The detailed measures describe the change of the state of the buffer pool over time and activities associated with the movement of pages in and out of the buffer pool. Quantities measured include the number of dirty pages in the buffer pool, the number of pending I/O requests, the number of pending writes (synchronous writes and asynchronous writes), and the number of reads over some interval of interest. The metric *Transactions Finished per Interval* is used to measure the throughput of the buffer pool. If the time interval is one minute, the *Transactions Finished per Interval* is the same as TPM. In accordance with TPC-C practice, only the number of New-Order transactions finished is reported.

5.2 Simulator Calibration and Verification

Running the simulator with the time parameters shown in Table 3 generates the throughput curve shown in Figure 4. For comparison, results from a TPC-C benchmark run on our experimental testbed (the "real system") are presented as well. To show the initial portion of the curve more clearly, only half the transactions (roughly 100 thousand) of the trace were used to produce this particular plot. Both the simulator and the real system complete the same number of transactions in this test. For the simulated results, throughput is expressed in terms of *Transactions Finished per Interval*. The interval used is $\frac{1}{30}$ of the run time. For the real system, the interval used is 30 seconds which is also $\frac{1}{30}$ of the run time. Because the real system and the simulator achieve a similar peak throughput as the buffer pool fills up and then fall to a similar stable value as the management algorithm kicks in, these two throughput curves are considered similar.

Other verification was performed. For example, after the system throughput becomes stable, the percentage of dirty pages in the buffer pool of the simulator is 87%, which is close to the measurement result of 85%. The buffer pool hit ratio from simulation is 96.6%, which is also close to the measurement result of 96.5%.

6 Experimental Results

6.1 Effect of CPU and Disk Speed

Two experiments were performed to determine the extent to which device speeds affect performance, one with a much faster disk and the other with a much faster CPU. To focus on the change in throughput at the beginning, a shorter trace is used. Figure 5 shows the result for the original parameters. Figures 6 and 7 show the results for a faster CPU and a faster disk, respectively.

These results indicate that disk speed is more important to DB2 than CPU speed. With a faster disk (5 times faster), the system runs faster (in this case it spends $\frac{1}{5}$ the time to complete all transactions), but with a faster CPU (10 times faster), the performance is the same

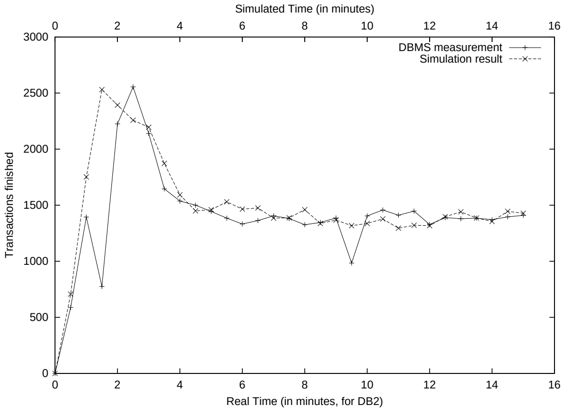

Figure 4: Simulation vs. Measurement

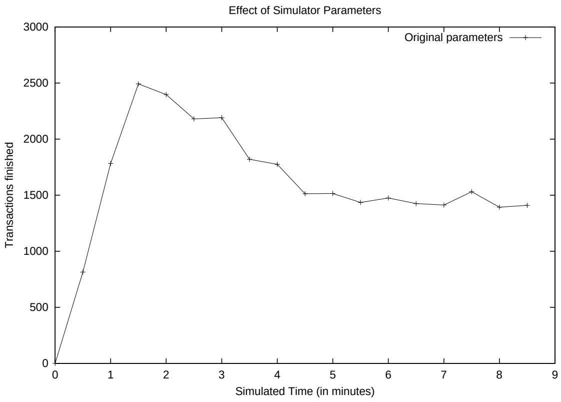

Figure 5: Original Parameters

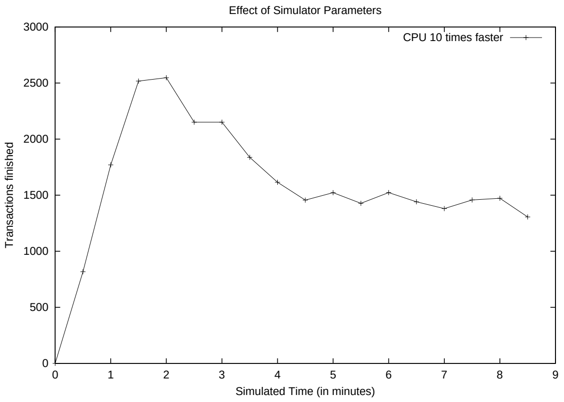

Figure 6: Faster CPU

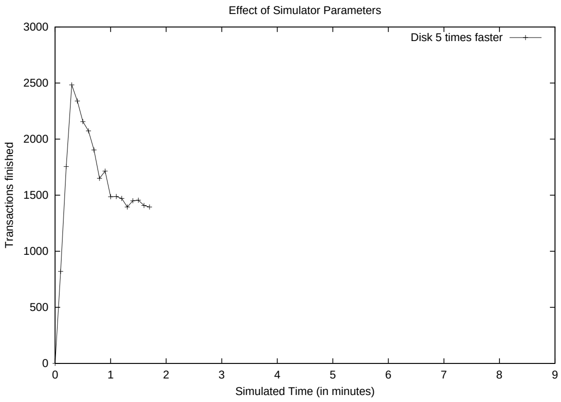

Figure 7: Faster Disk

as the original test. This indicates that the system is likely I/O bound.

6.2 Analysis of the Dirty Page Distribution and I/O Activities

In the experiments performed in this subsection, the default configuration of DB2 (the number of page cleaners is 2, and the dirty page threshold is 60%) was used. Figure 4 shows that the stable throughput is lower than the peak throughput under this configuration. We carried out a series of experiments to determine how various factors of the buffer pool contribute to performance.

During the verification, we found that 87% of the pages in the buffer pool are dirty, which seems high. This motivated more experiments on the page distribution in the buffer pool. Figure 8 shows the evolution of pages in the buffer pool over the course of the simulation. Also shown on this graph is the (scaled) throughput (finished transactions per interval×20). At the beginning, all pages in the buffer pool are free pages. Both the number of dirty pages and the number of clean pages increase as time goes on. After the buffer pool is full, the number of dirty pages continues to increase, but the number of clean pages drops. At the same time, the throughput drops as well. At last, when 87% of the buffer pool pages are dirty, the system enters a steady state. The number of dirty pages at this point is much higher than it is when the buffer pool is just full, implying that there are too many dirty pages in the buffer pool. Therefore, the effect of the *dirty page threshold*, which is used to control the percentage of dirty pages in the buffer pool, was subjected to further testing.

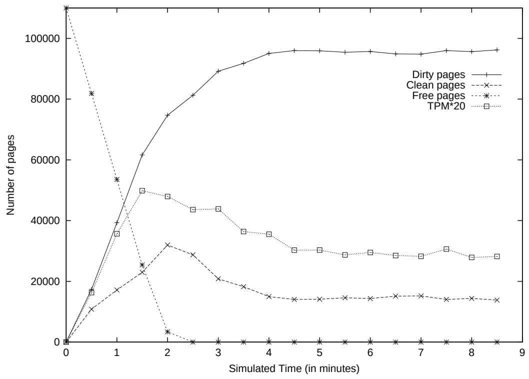

Figure 8: Pages in the Buffer Pool

Different dirty page threshold values are tested in both the simulator and DB2. The results show that the percentages of dirty pages are nearly the same regardless of the value of the threshold. This indicates that in the TPC-C workload, the threshold alone cannot effectively control the percentage of dirty pages.

To investigate the reason of this observation, the I/O activities of the buffer pool were examined in more detail. The results are shown in Figure 9. We can see that the total I/O bandwidth is constant, and is shared by reads, synchronous writes, and asynchronous writes. The read speed is consistent with the throughput of the system. In an OLTP environment, the faster transactions are processed, the faster read requests arrive, and the faster the pages are modified.

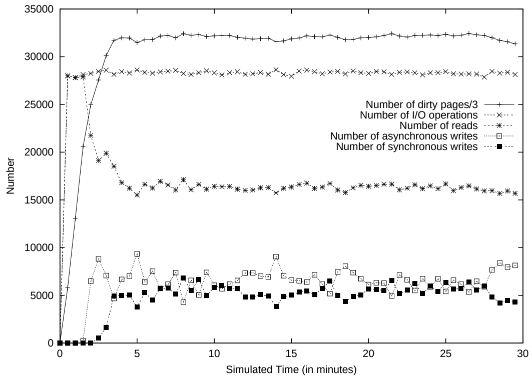

Figure 9: I/O Activities of the Buffer Pool

When the buffer pool is full, page cleaners begin to clean out dirty pages by asynchronous writes. However, asynchronous writes cannot clean out pages fast enough, and so dirty pages must be selected for replacement. Therefore, synchronous writes occur. The synchronous writes not only delay the reads directly (since a read cannot proceed before the synchronous write finishes), but also compete with other activities for I/O bandwidth. Therefore, the read speed is slowed down by the need to write in order to create space for the incoming pages. When the read speed becomes slower, the throughput drops, and dirty pages are generated more slowly. When the number of dirty pages generated by TPC-C equals the number of dirty pages cleaned by writes in the same time interval, the system enters a steady state.

As shown in Figure 9, the number of synchronous writes is high. The existence of too many synchronous writes impacts throughput. The number of asynchronous writes should be increased in order to decrease the number of synchronous writes. To do this, the aggregate page cleaning speed must be increased.

6.3 The Impact of More Page Cleaners

The aggregate page cleaning speed can be increased if more page cleaners are used. The default number of page cleaners is 2. Figure 10 shows that with 50 page cleaners, the throughput increases. Figure 11 shows the I/O activities and the number of dirty pages when the number of page cleaners is 50. There are nearly no synchronous writes left. The number of dirty pages also drops significantly.

Figure 12 illustrates the throughput and dirty page percentage under different numbers of page cleaners. Increasing the number of page cleaners reduces both the percentage of dirty pages and the percentage of synchronous writes. Up to 50 page cleaners, throughput is improved with more cleaning. After that point, however, throughput drops sharply as putting more page cleaners to work is unable to improve performance. The selection of the appropriate number of page cleaners is clearly important in tuning such a system. Similar tests on the DB2 testbed show similar results as Figure 12.

The page cleaning speed has an upper limit,

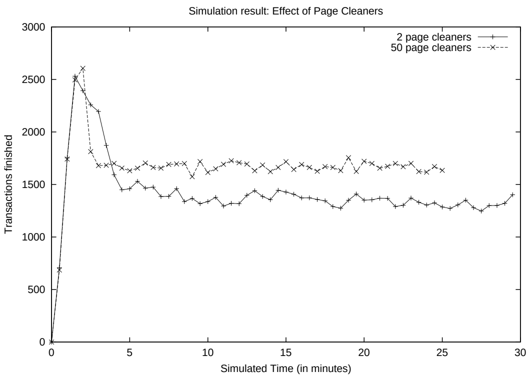

Figure 10: Effect of Number of Page Cleaners

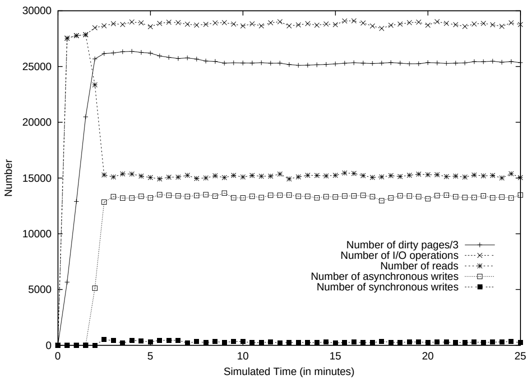

Figure 11: I/O Activities for 50 Page Cleaners

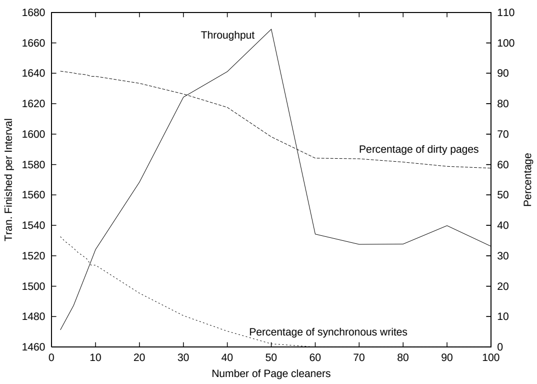

Figure 12: Effect of Multiple Page Cleaners

which is achieved when all page cleaners are busy. However, the read speed has no upper limit; reads are aggressive in the TPC-C benchmark, i.e., they will consume all available disk I/O bandwidth. If the upper limit of the page cleaning speed is too low, reading can be slowed down only by synchronous writes. This is the situation with the default configuration. Increasing the number of page cleaners can increase the upper bound of the page cleaning speed. Therefore, it can decrease synchronous writes and increase throughput.

7 Conclusions and Future Work

Buffer pool management is important to the performance of a DBMS. Some elements of the buffer pool management algorithm of DB2 were analyzed by trace-driven simulation, using traces captured from running the TPC-C benchmark. The effects of different parameters of the buffer pool were tested, and the I/O activities of the buffer pool were examined. Under the default configuration, although all page cleaners keep busy during the running period, dirty pages still make up more than 85% of the buffer pool. The results from further simulation experiments suggest that for this workload the default number of page cleaners are insufficient to clean out dirty pages fast enough, and synchronous writes have to be applied as a result. Throughput is impacted negatively by the need to wait on synchronous writes. The aggregate page cleaning speed can be increased by increasing the number of page cleaners. The results of our simulation experiments suggest that this change can increase system throughput.

The TPC-C database used in this study for both trace collection and live measurement contains 50 warehouses, which is quite small. Future work includes extending this research to a much larger TPC-C database, and testing alternative page cleaning strategies, first in the simulator and then in the real system.

Acknowledgments

The research reported in this paper has been supported by the Centre for Advanced Studies (CAS) at the IBM Toronto Laboratory. The authors would like to thank many people at CAS who associated them in their work, answered many technical questions, and gave helpful suggestions: in particular, Berni

Schiefer, Steve Rees, Yongli An, Keri Romanufa, Aamer Sachedina, and John Li. Some initial development on both the simulator and the tracing tool was done at the University of Saskatchewan by Kevin Froese.

The authors would also like to thank Yanping Zhao, for her suggestions and helpful discussions.

About the Authors

Wenguang Wang received the BS degree in computer science (honours) from the Shandong University of China, and the ME degree in computer application from the Institute of Computing Technology at the Chinese Academy of Sciences. He is presently a Ph.D. student in the Department of Computer Science, at the University of Saskatchewan. His areas of research include performance evaluation of buffer management and I/O support in database systems.

Richard B. Bunt received the BSc degree from the Queen's University, and the Ph.D. degree from the University of Toronto. He is a faculty member in the Department of Computer Science, at the University of Saskatchewan, and his areas of research include performance of distributed computing environments, in particular, operating systems, distributed systems, and mobile computing.

References

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

[2] Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[3] Wolfgang Effelsberg and Mary E. S. Loomis. Logical, internal, and physical reference behavior in CODASYL database systems. *ACM Transactions on Database Systems*, 9(2):187–213, June 1984.

[4] Christos Faloutsos, Raymond T. Ng, and Timos K. Sellis. Flexible and adaptable buffer management techniques for database management systems. *IEEE Transactions on Computers*, 44(4):546–560, April 1995.

[5] John P. Kearns and Samuel DeFazio. Locality of reference in hierarchical database systems. *IEEE Transactions on Software Engineering (SE), Vol.SE-9*, (2):128–134, March 1983.

[6] John P. Kearns and Samuel Defazio. Diversity in database reference behavior. In *Proc. of the ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems*, volume 17(1) of *ACM SIGMETRICS Performance Evaluation Review*, pages 11–19, Berkeley, California, USA, May 23–26 1989. ACM Press.

[7] TPC Benchmark^{TM} C. `http://www.tpc.org/benchmark_specifications/TPC_C/tpc-c35.pdf`, October 1999.