# A Self-tuning Page Cleaner for DB2[*]

Wenguang Wang Rick Bunt
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
Email: {wang, bunt}@cs.usask.ca

## Abstract

*The buffer pool in a DBMS is used to cache the disk pages of the database. Because typical database workloads are I/O-bound, the effectiveness of the buffer pool management algorithm is a crucial factor in the performance of the DBMS. In IBM's DB2 buffer pool, the page cleaning algorithm is used to write changed pages to disks before they are selected for replacement. We conducted a detailed study of page cleaning in DB2 version 7.1.0 for Windows by both trace-driven simulation and measurements. Our results show that system throughput can be increased by 19% when the page cleaning algorithm is carefully tuned. In practice, however, the manual tuning of this algorithm is difficult. A self-tuning algorithm for page cleaning is proposed in this paper to automate this tuning task. Simulation results show that the self-tuning algorithm can achieve performance comparable to the best manually tuned system.*

Keywords: DBMS, DB2, Buffer Pool Management, Page Cleaning, Self-tuning Algorithms

## 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 to 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 the IBM® Corporation's popular DBMS, DB2® (specifically, version 7.1.0 for Windows®).

The On-Line Transaction Processing (OLTP) workload is an important type of workload for a DBMS. Because typical queries used in OLTP workloads are very simple, query optimization does not play a significant role in performance, but the management of buffer pool is crucial because of the randomness of the disk I/Os. Therefore, the OLTP workload is an ideal workload to study buffer pool management. The TPC benchmark™ C [7] provides a workload representing an OLTP environment, and is used as the workload in our study. 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.

OLTP workloads are normally I/O-bound. There are four types of I/O requests in the DB2 buffer pool: prefetched reads, normal reads, synchronous writes, and asynchronous writes. Since there is almost no sequential access of data in the workload studied, prefetching is not used in this workload, thus prefetched reads are not discussed in this paper. Whenever a page is read into the buffer pool, a physical disk read occurs. If the buffer pool is full, a page needs to be selected for replacement before a new page can be read into the buffer pool. If the page selected for replacement has been modified (called a dirty page), it must be written to disk before the new page is read in. This is a *synchronous* write. At the same time, one or more "page cleaners" (threads or processes) are at work in the buffer pool. These page cleaners collect changed pages and write them to the disks before they are selected for replacement. These write operations are *asynchronous* since they aren't initiated directly as a result of read operations. The asynchronous writes which are caused by the page cleaning activities can be managed by changing the number of page cleaners. This

[*]This research was supported by IBM's Centre for Advanced Studies (CAS) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

is the focus of this paper.

Several events can trigger the page cleaners to start cleaning. Dirty replacement (when a dirty page is selected for replacement) is one of these events. Since dirty replacements happen frequently in the workload we use in this paper when the system is untuned, other events that might trigger page cleaning are not considered.

The page cleaning speed can affect system throughput significantly. The page cleaning speed can be controlled by the number of page cleaners, and the number of page cleaners can be set by the database administrator before the users connect to the database server. Simulation results presented later in the paper show that tuning the number of page cleaners to an "optimal" value can improve system throughput by as much as 19%, but manually tuning this parameter is difficult for the following reasons:

- A workload of sufficient length must be available to determine how system performs under a particular setting.
- Each performance "experiment" must run long enough to skip the buffer pool warmup period and to eliminate statistical fluctuations resulting from short-term transient effects.
- The database must contain enough data to provide a realistic operating environment.
- When the system configuration or workload changes (e.g., more disks or memory are used, the database becomes larger, or more users are using the system), the tuning must be performed again.

For the research reported in this paper, a detailed study was conducted of the I/O activities of the buffer pool and of the effect on performance of the number of page cleaners. From the insights gained, a self-tuning algorithm for page cleaning is developed to automate the tuning task by periodically adjusting the page cleaning speed. Simulation results show that, on average, the throughput of the self-tuned system is close to the best manually tuned system, and that with larger buffer pools or more disks, the self-tuning algorithm performs even closer to the best manually tuned system. Furthermore, the performance of the self-tuning algorithm is not sensitive to the particular parameter values used.

This paper is organized as follows: Section 2 gives a brief review of some previous work related to database buffer pool management and self-tuning algorithms; Section 3 describes the page cleaning algorithm used in DB2; Section 4 presents the results of the experiments conducted to study the relationships among buffer pool I/O activities and the effect of the number of page cleaners on system performance; Section 5 discusses the self-tuning algorithm for page cleaning; Section 6 presents the simulation results of the self-tuning algorithm; and Section 7 gives conclusions and future work.

## 2 Related work

A buffer pool is used in a DBMS to reduce disk I/Os and effective management of the buffer pool is an important performance factor. Various buffer pool management algorithms, including LRU, FIFO, CLOCK, LRD, DGCLOCK, and Working Set, are analyzed in [4]. In addition to simple replacement algorithms such as these, there are many more sophisticated algorithms that utilize information of various sorts in making their replacement decisions. For example, algorithms such as ILRU [5], OLRU [5], HOTSET [6], and DBMIN [2] are designed to respond to the reference behaviour of database indexes and queries.

Because of the complexity of real database systems, the effective configuration and tuning of any management algorithm is a significant challenge for the DBA. To ease the task, some goal-oriented tuning algorithms have been proposed. An approach that dynamically adjusts the buffer pool sizes of a DBMS based on response time goals is presented in [3]. A goal-oriented tuning architecture to convert the low level control knobs (buffer pool size, working buffer size, etc.) to high level transaction response time goals is proposed in [1]. These goal-oriented tuning algorithms are feedback algorithms. Response time goals of different transaction classes must be specified by the database administrator (DBA) before the application starts. These algorithms collect system states periodically and adjust the buffer pool allocated for each transaction class to meet the response time goals. The self-tuning algorithm proposed in this paper is also a feedback algorithm. It does not need a goal from the DBA, however, since its goal is to maximize system throughput.

## 3 The DB2 page cleaning algorithm

The buffer pool is a buffer which caches disk pages for the DBMS. Figure 1 shows the structure of the buffer pool. Since many users can use DB2 simultaneously, there is one database agent (a thread or a process) corresponding to each current user. Each agent processes that user's queries and requests database pages from the buffer pool.

Unlike 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 [4] is used. Agents send *fix* requests to the buffer pool while processing queries. If the page is already in the buffer pool, no physical I/O is needed; otherwise, the page is read into the buffer pool from the disk. DB2 can access this page freely from the buffer pool since it cannot be evicted from memory after

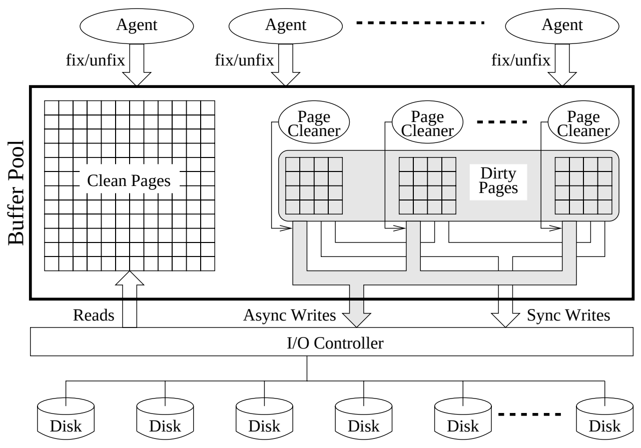


**Figure 1. The structure of the buffer pool**

the fix. When DB2 finishes using this page, an *unfix* request is sent to the buffer pool. After the unfix, this page is allowed to be replaced as long as there are no other fixes still outstanding.

A common performance metric for a buffer pool management algorithm is the read miss ratio, but this considers only the cost of physical reads. Although this may be sufficient for virtual memory management in operating systems, it might be less useful in a DBMS running OLTP workloads where many physical writes take place. In such an environment, ignoring write cost may lead to improper design of the buffer pool replacement algorithm. To overcome this problem, both the read cost and the write cost should be considered. Since this kind of system is normally I/O bound, the system throughput is inversely proportional to the total I/O time, which is equal to the sum of read time and write time. Because of the randomness of the disk I/O in the OLTP workload, the disk level caching and prefetching do not have large effect on the I/O time. Moreover, there is no extra write overhead due to the RAID-0 disk organization used in the disk system. Therefore, the read cost and write cost are considered the same in the system studied in this paper. Thus, the lower the sum of the reads and writes, the higher the system throughput.

When the buffer pool is full, a page must be selected for replacement when a new page needs to be read into the buffer pool. When a page is selected, its status is checked. If it is clean, the space it occupies can be used immediately, but if it is dirty, a synchronous write must take place before the user agent can fetch the new page.

In addition to synchronous writes, DB2 also uses page cleaners to perform asynchronous writes. Each page cleaner manages a subset of dirty pages as shown in Figure 1. All page cleaners are asleep initially. When a page cleaner wakes up, it collects dirty pages and writes them to the disk. During the cleaning, the user agents can continue to request buffer pool pages from the buffer pool since the writes generated by page cleaners are performed asynchronously.

DB2's page cleaners wake up if a synchronous write occurs, if the proportion of dirty pages exceeds a threshold value, or if the proportion of changes recorded in the log file exceeds a threshold value. Since synchronous writes are very frequent occurrences with the workload we used in this paper when the system is untuned, the page cleaners are always awake, and so other triggering mechanisms are not studied.

## 4 Experiments with the page cleaning algorithm

### 4.1 Methodology

Both simulation and measurements are used to study the performance of the page cleaning algorithm in DB2 version 7.1.0 for Windows, and the factors that affect it. A trace-driven buffer pool simulator was developed and validated [8], and traces of buffer pool activities were collected to provide realistic input. The simulator is being used to study the performance of the page cleaning algorithm and various activities of the buffer pool and the I/O system. The performance of the new self-tuning page cleaning algorithm was studied in the simulator as well.

The TPC-C benchmark is used to provide the workload to the simulator and to provide the system against which simulation results are compared. The TPC-C Benchmark is an OLTP workload made up of five different transactions. System throughput is measured by the number of New-Order transactions the system is able to process each minute. The size of the TPC-C database is given by the number of "warehouses" defined (each warehouse holds about 100MB of data). Since this study focuses on the server workload, remote terminal emulators are not used to generate the TPC-C transactions. All transactions are generated on the DBMS server with no think time between transactions. Because the TPC-C benchmark prohibits the public disclosure of TPC-C performance results that have not been audited by independent auditing agencies, the absolute values of any simulation or experimental results are withheld, and only normalized values are presented in this paper. This does not compromise the performance comparisons.

The trace of the buffer pool activities needed for the simulator was collected from transactions against a TPC-C database with 50 warehouses spanning 11 physical disks. The benchmark was run on a PC Server running Windows NT® Server 4.0 in the Distributed Systems Performance Laboratory at the University of Saskatchewan. During trace recording, all fix/unfix requests to the buffer pool through the trace point are recorded by the trace tool in a memory buffer and stored periodically for subsequent processing. This technique (described in [8]) enables us to obtain arbitrarily large traces for our experiments.

## 4.2 I/O activities in the buffer pool

Some general observations about the buffer pool management and its I/O behaviours are presented in this subsection in order to understand the impact of the page cleaning algorithm on performance.

In the experiments described, an untuned configuration of DB2 was used, with 2 page cleaners. Figure 2 shows the system throughput under this configuration measured in both the simulator and DB2. The y-axis is the TPM which is normalized relative to the average TPM of the measurement results when the system enters the stable state. The x-axis is the running time which is normalized relative to the total running time of the measurement experiment. This figure shows that the simulation results are quite close to those obtained from measurement. The throughput spike shown in the figure occurs only in the system warmup phase when the page cleaners just start cleaning.

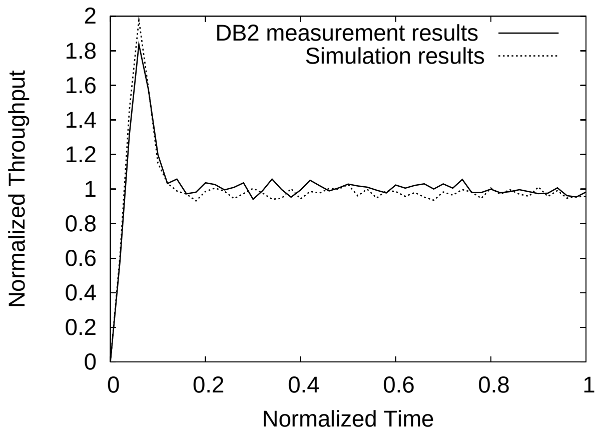


**Figure 2. Simulation vs. measurement**

We carried out a series of experiments to determine how various factors of the buffer pool contribute to performance. In our initial experiments, we found that 90% of the pages in the buffer pool are dirty, which seems high. This motivated more experiments on the distribution of pages in the buffer pool. Figure 3 shows the evolution of pages in the buffer pool over the first 30% time of the simulation. Also shown on this graph is the throughput, scaled so that its shape can be compared with the other curves plotted. The number of pages is normalized relative to the buffer pool size. 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 under this untuned configuration. Finally, when 90% of the buffer pool pages are dirty, the system enters a steady state. The number of clean pages at this point is much lower than it is when the buffer pool is just full, implying that there are too many dirty pages in the buffer pool in the steady state. To investigate the reason for this, the I/O activities of the buffer pool were examined in more detail. The results are shown in Figure 4.

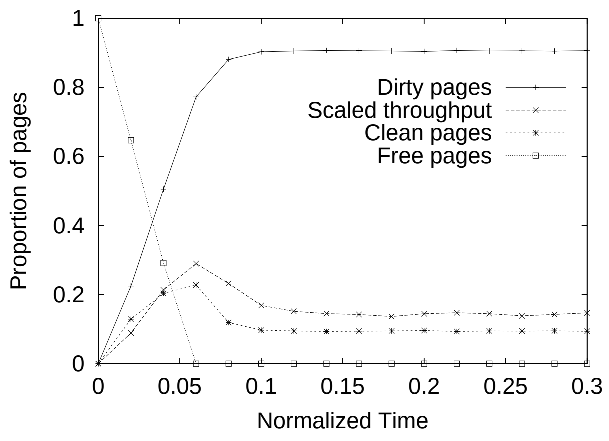


**Figure 3. Pages in the buffer pool in the untuned configuration (2 page cleaners)**

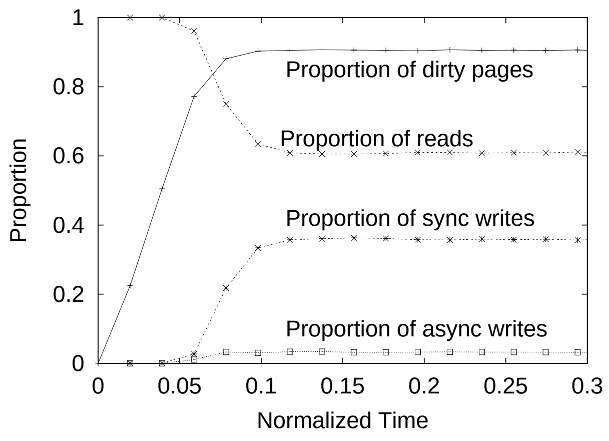


**Figure 4. I/O activities of the buffer pool in the untuned configuration (2 page cleaners)**

When the buffer pool is almost full, the page cleaners are triggered by one of the related events. They begin to clean out dirty pages by asynchronous writes. However, asynchronous writes cannot clean out pages fast enough in this untuned configuration, so dirty pages must be selected for replacement. This means that 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 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 (i.e., fewer pages in the buffer pool are changed per unit time). When the number of dirty pages generated by the TPC-C requests equals the number of dirty pages cleaned by writes in the same time interval, the system enters a steady state.

As shown in Figure 4, the proportion of synchronous writes is high (close to 40% of all I/O activity), which implies that the page cleaning speed is too low. 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. The aggregate page cleaning speed can be increased by using more page cleaners.

## 4.3 The impact of more page cleaners

The number of page cleaners in the untuned configuration is 2. Figure 5 shows that throughput improves when more than 2 page cleaners are used - in this case 44 page cleaners. The run time on the x-axis and the throughput on the y-axis are normalized relative to the run time and average throughput with 2 page cleaners. Figure 6 shows the I/O activities and the proportion of dirty pages with 44 page cleaners: there are very few synchronous writes left and the proportion of dirty pages drops significantly.

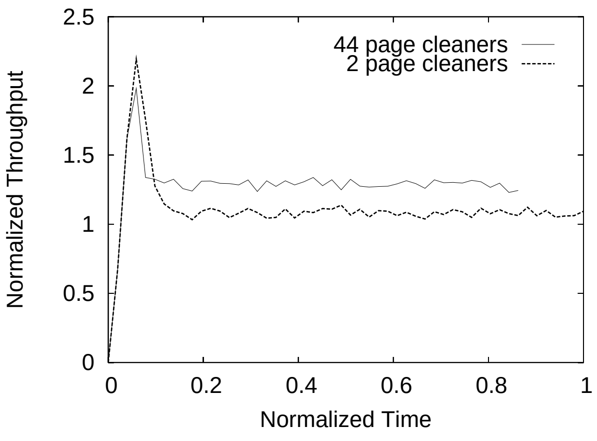


**Figure 5. Effect of number of page cleaners**

Figure 7 shows the effect of varying the number of page cleaners from 1 to 100. Figure 7(a) shows the effect of the number of page cleaners on throughput. All throughput values are normalized relative to the throughput under the untuned configuration. The simulation results match the measurement results quite closely. When the number of page

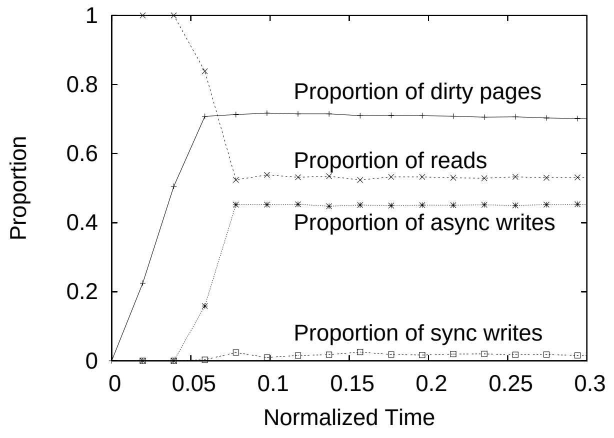
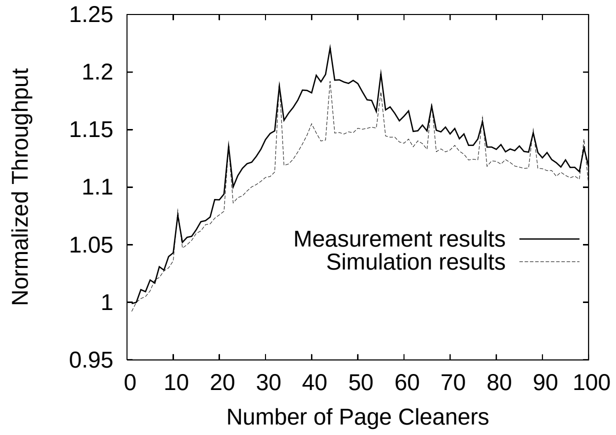
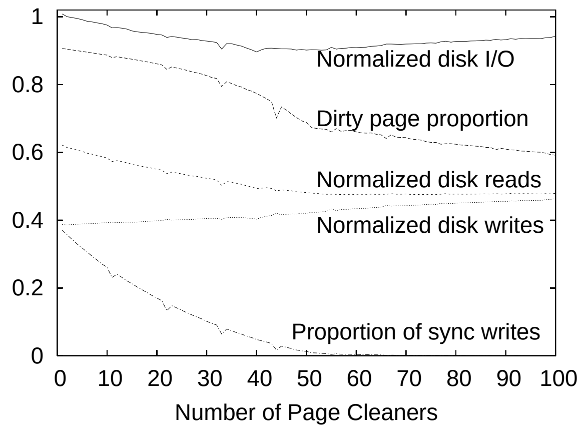


**Figure 6. I/O activities with 44 page cleaners**



(a) Effect on throughput



(b) Effect on dirty pages, synchronous writes, and disk I/Os

**Figure 7. Effect of multiple page cleaners**

cleaners is an integral multiple of the number of physical disks, which is 11 in this case, better load balance across disks can be achieved. Therefore, the performance spikes occur on these specific points.

When the number of page cleaners is less than 44, the throughput generally increases with more page cleaners. After that point, however, putting more page cleaners to work is unable to improve performance any further. More is not always better. The selection of the appropriate number of page cleaners is clearly important in tuning such a system.

Figure 7(b) shows the effect of the number of page cleaners on dirty pages, synchronous writes, and disk reads, disk writes, and total I/Os. The number of disk reads, disk writes, and total I/Os are normalized relative to the disk I/Os under the untuned configuration. When the number of page cleaners increases, the number of read misses drops and the number of write misses increases. The number of disk I/Os first decreases then increases. Increasing the number of page cleaners reduces both the proportion of dirty pages and the proportion of synchronous writes. Further increasing the number of page cleaners after the proportion of synchronous writes is close to 0 brings no additional benefits: the number of read misses becomes almost flat; the decrease of dirty pages slows down; the number of disk I/Os starts to increase; and throughput drops. Figure 7(b) shows a criterion for tuning the page cleaning activity: the number of page cleaners should be tuned to the minimum number so that the synchronous writes are just eliminated. A self-tuning algorithm for changing the page cleaning speed based on this principle is described in the next section.

These simulation experiments were performed on systems with different numbers of disks and different buffer pool sizes, and the effect of the number of page cleaners on throughput was very similar to Figure 7(a), although the locations of spikes was different because there was a different number of disks. The optimal number of page cleaners for each system configuration is shown in Table 1. The optimal number of page cleaners is always an integral multiple of the number of physical disks, but the optimal value is different under different configurations, which increases the difficulty of the tuning of the number of page cleaners.

**Table 1. Optimal number of page cleaners**

| Number of Disks | 7 | 8 | 11 | 15 |
|---|---|---|---|---|
| Buffer Pool=380MB | 49 | 48 | 44 | 45 |
| Buffer Pool=440MB | 49 | 56 | 44 | 45 |

## 5 A self-tuning algorithm for page cleaning

The number of page cleaners that maximizes system performance is different for different workloads or different configurations. Tuning this parameter manually is difficult and time-consuming. A self-tuning page cleaning algorithm was designed to overcome this problem, the objective of which is to maximize throughput by dynamically changing the page cleaning speed.

The number of page cleaners is fixed in the self-tuning algorithm to keep the algorithm simple. In the page cleaning algorithm used in DB2 7.1.0 for Windows, each page cleaner collects many pages and sends out one page at a time for cleaning. One more page is sent out after the previous write is done.

In order to change the page cleaning speed without changing the number of page cleaners in the self-tuning algorithm, each page cleaner keeps more than one outstanding asynchronous write. A parameter $N$ is introduced for this purpose – a real number whose integral part $\lfloor N \rfloor$ indicates the number of outstanding asynchronous writes kept by each page cleaner. The page cleaner compares the number of outstanding asynchronous writes with $\lfloor N \rfloor$ whenever an asynchronous write sent by this page cleaner finishes. If there are more than $\lfloor N \rfloor$ asynchronous writes outstanding, the page cleaner stops sending new writes to disk; otherwise, more writes are sent to disks until the number of outstanding asynchronous writes sent by this cleaner equals $\lfloor N \rfloor$. $N$ thus has the same effect as the number of page cleaners in the current algorithm: the bigger the $N$ value, the faster the page cleaning speed.

The initial value of $N$ is its minimum value 1. $N$ is adjusted periodically in order to dynamically tune the page cleaning speed to its optimal value. An *adjustment interval* is defined for this purpose. Some statistics of the buffer pool and the disk activities are collected during each adjustment interval. $N$ is adjusted at the end of each adjustment interval based on the data collected.

An adjustment goal must be defined so that $N$ can be adjusted to make the system achieve the goal. The results presented in Section 4.3 show that the page cleaning speed should be increased to the point where the number of synchronous writes just reaches 0. It is easy to determine the number of synchronous writes that occurred in any adjustment interval, but it is hard to tell whether the page cleaning speed is too high or not if the observed number of synchronous writes is 0. As Figure 7(b) shows, the number of synchronous writes is 0 when the number of page cleaners is more than necessary. Therefore, adjusting the number of synchronous writes to 0 is not a goal that is easy to reach in the self-tuning algorithm, although this is possible with manual tuning. Instead, the self-tuning algorithm seeks to keep the *proportion* of synchronous writes small (say, 5%).

The following notation is used in describing the adjustment operation performed in every adjustment interval:

- $w_o$: proportion of synchronous writes observed in an adjustment interval.
- $w_d$: the desired proportion of synchronous writes.
- $\Delta$: the scale parameter.

At the end of each adjustment interval, the following adjustment is performed:

$$N = \max(1, N + \Delta \cdot (w_o - w_d)) \qquad (1)$$

During each adjustment interval, a count is maintained of synchronous writes and total disk I/Os. The ratio is the observed proportion of synchronous writes, $w_o$. At the end of every adjustment interval, $w_o$ is compared with the desired proportion of synchronous writes $w_d$. The greater the difference between $w_o$ and $w_d$, the more $N$ needs to be changed. The change to $N$ should be proportional to $|w_o - w_d|$. The value of $\Delta \cdot (w_o - w_d)$ in Equation 1 shows the amount that $N$ needs to be changed. The scale parameter $\Delta$ is used to amplify the difference between $w_o$ and $w_d$. If $w_o$ equals $w_d$, the current page cleaning speed is the desired value and $N$ can remain unchanged. If $w_o > w_d$, the proportion of synchronous writes is more than desired. Thus $N$ needs to be increased to clean pages faster. If $w_o < w_d$, the proportion of synchronous writes is less than desired, which indicates that the page cleaning speed is too high. Thus $\Delta \cdot (w_o - w_d))$ is negative and its absolute value indicates the amount that $N$ should be decreased. Since the minimum value of $N$ is 1, the use of the $\max$ function guarantees that $N \geq 1$ after the adjustment.

## 6 Simulation results

The results of the simulation experiments with the self-tuning algorithm are presented in this section. This algorithm uses three parameters *Adjustment Interval*, $\Delta$, and $w_d$. The values listed in Table 2 were used to generate the simulation results presented in this section.

**Table 2. The parameter values**

| Parameter | Value |
|---|---|
| Interval | 1 second |
| $w_d$ | 5% |
| $\Delta$ | 7.5 |

Figure 8 shows the throughput of the system under the untuned configuration (2 page cleaners), the best manually tuned configuration (44 page cleaners), and the self-tuning algorithm. The performance of the self-tuning algorithm is close to that of the best manually tuned system. The throughput of the best manually tuned system is 19.2% higher than that of the untuned configuration, and the throughput of the self-tuning algorithm is 16.3% higher. This result shows that the self-tuning algorithm performs comparably to the best manually tuned system.

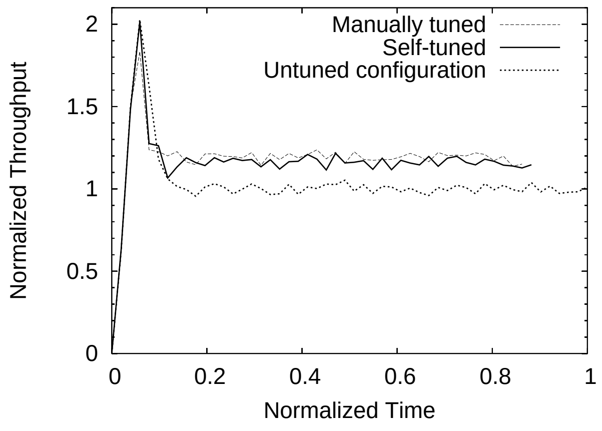


**Figure 8. Throughput comparison**

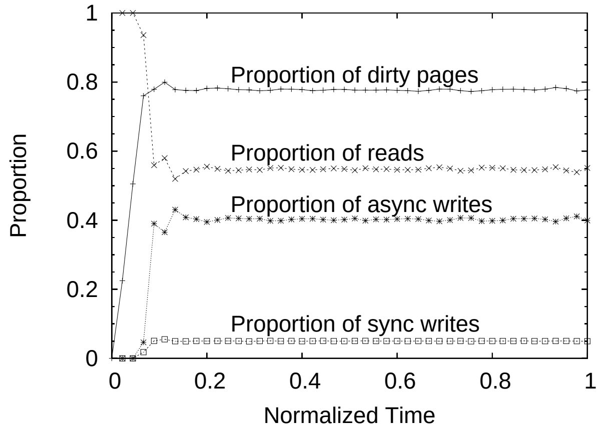


**Figure 9. I/O activities with the self-tuning algorithm**

Figure 9 shows the system I/O activities when running the self-tuning algorithm. The proportion of synchronous writes is kept very close to 5% which is the same as the value of $w_d$. Because of the higher page cleaning speed, the proportion of dirty pages is lower than that of the untuned configuration, indicating that the self-tuning algorithm can effectively control the proportion of synchronous writes. Figure 10 shows how the parameter $N$ is adjusted in a ten-minute interval. The value of $N$ fluctuates in a small range (between 3 and 5), because the characteristics of this workload do not change.

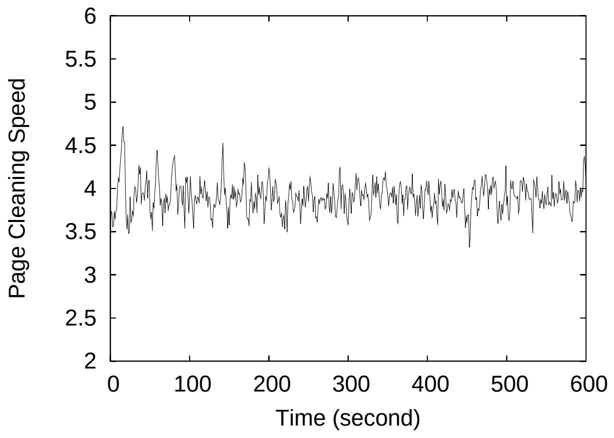


**Figure 10. How page cleaning speed is adjusted**

This self-tuning algorithm was also tested in several other system configurations (different numbers of physical disks and different buffer pool sizes). The results are summarized in Figure 11. All throughput values are normalized relative to the throughput under the untuned configuration (2 page cleaners). The results show that the self-tuning algorithm performs close to the best manually tuned algorithm.

In order for this algorithm to be robust, the performance must not be unduly sensitive to the selection of values for the three parameters (Adjustment Interval, $w_d$, and $\Delta$). More simulation experiments were performed to determine the sensitivity of the results to the values of these parameters. All throughput values in the following figures are normalized relative to the average throughput under the parameter values shown in Table 2.

Figure 12 shows the impact of the adjustment interval on performance. Even though the adjustment interval is varied from 0.1 sec to 20 secs (two orders of magnitude), the system throughput changes by less than 1%. Further experiments showed that as long as the adjustment interval is at least several times longer than the average disk access time (about 10ms for typical hard drives), there is no significant difference in performance. A small interval can respond promptly to a workload change, while a large interval can reduce system overhead. Since the workload of TPC-C does not change in the simulation experiments performed, the adjustment interval is not important to throughput. Figure 13 shows that when the desired synchronous write proportion $w_d$ changes from 0.2% to 10%, the throughput also varies by less than 1%. This indicates that as long as $w_d$ is a small value, performance does not change significantly. Figure 14 shows the impact of the scale parameter $\Delta$ under two different adjustment intervals. Again the performance difference is within 1%. The results of these experiments indicate that the self-tuning algorithm can be used for different systems without changing the parameters.

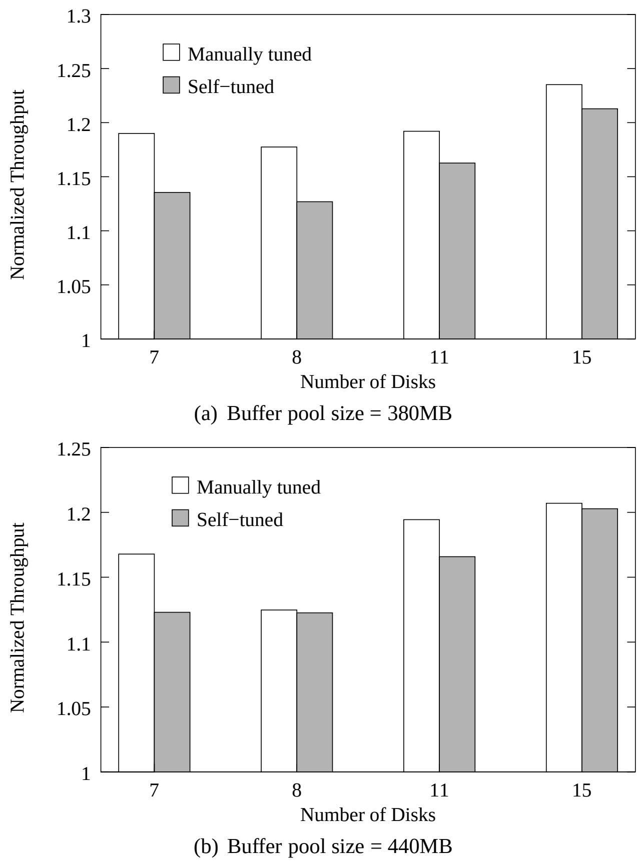


**Figure 11. Comparison of self-tuned and manually tuned algorithms**

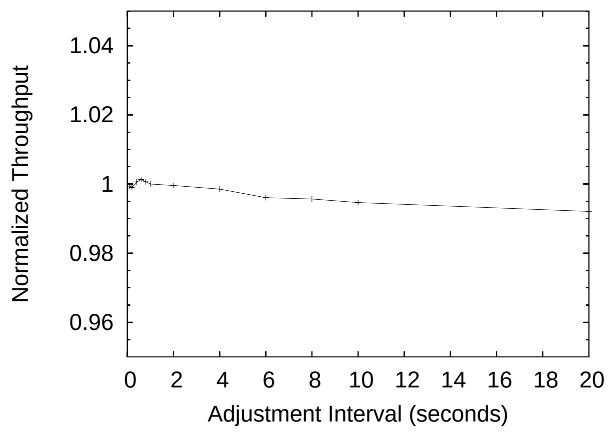


**Figure 12. Impact of the adjustment interval**

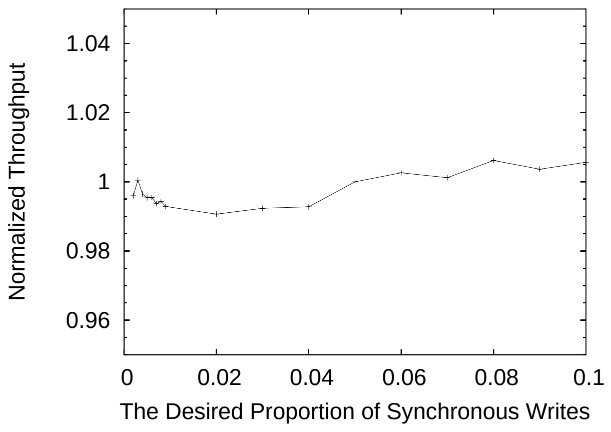


**Figure 13. The effect of parameter $w_d$**

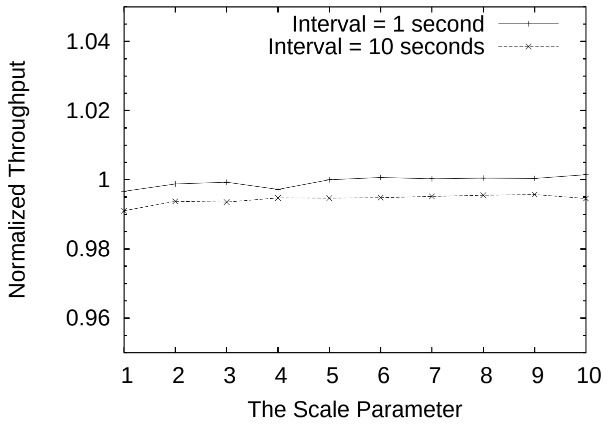


**Figure 14. Impact of the scale parameter $\Delta$**

## 7 Conclusions and future work

Buffer pool management is important to the performance of any 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. Our analysis of the I/O activities of the buffer pool tells us that properly tuning the number of page cleaners is important to performance. A self-tuning algorithm is proposed to make this tuning easier. The algorithm is simple to implement and requires no special configuration. The results of our simulation experiments show that the self-tuning algorithm can achieve throughput comparable to that of the best manually tuned algorithm.

Our current and future work includes implementing this algorithm in DB2 and measuring its effectiveness in a larger system, and testing the effectiveness of this approach for real workloads with changing characteristics – first in our simulator and then in real systems.

## Acknowledgments

We are grateful to many people at the IBM Toronto Laboratory who involved themselves in this work, answered many technical questions, and gave helpful suggestions: in particular, Berni Schiefer, Keri Romanufa, Aamer Sachedina, Steve Rees, Kelly Lyons, Yongli An, and John Li. Some initial development on both the simulator and the tracing tool was done at the University of Saskatchewan by Kevin Froese. We thank our colleague Dwight Makaroff for reading the paper and providing comments.

## References

[1] K. P. Brown. *Goal-oriented Memory Allocation in Database Management Systems*. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI, 1995.

[2] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the* $11^{th}$ *International Conference on Very Large Data Bases (VLDB'85)*, pages 174–188, Stockholm, Sweden, August 1985.

[3] J.-Y. Chung, D. Ferguson, G. Wang, C. Nikolaou, and J. Teng. Goal oriented dynamic buffer pool management for data base systems. Technical Report TR94-0125, ICS/FORTH, Heraklion, Crete, Greece, October 1994.

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

[5] G. M. Sacco. Index access with a finite buffer. In *Proceedings of the* $13^{th}$ *International Conference on Very Large Data Bases (VLDB'87)*, pages 301–309, Brighton, England, September 1987.

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

[7] TPC Benchmark$^{TM}$ C. `http://www.tpc.org/tpcc/`.

[8] W. Wang and R. Bunt. Simulating DB2 buffer pool management. In *Proceedings of CASCON 2000*, pages 88–97, Toronto, Canada, November 2000.