An Eager and Pessimistic Space Reservation Method for Tables Frequently Accessed by Concurrent Transactions

Kang-Woo Lee and Hyoung-Joo Kim

Dept. of Computer Science

Seoul National University

Seoul, Korea, 151-742

kwlee,hjk@oopsla.snu.ac.kr

October 8, 1998

Abstract

Space reservation is important in allocating and releasing storage space in DBMSs to support recoverable actions. Since most existing space reservation methods are designed to perform well when few data pages have spaces reserved by more than one transaction, they are not suitable for some of the newly emerging applications, such as workflow and inventory control systems. In these applications, many concurrent transactions frequently insert and delete records into/from a relatively small table.

This paper proposes a new space reservation method that performs well for these types of applications by extending the lock control structures to keep detailed and up-to-date information in a reserved state. This paper also shows, by simulation, that the new method performs better than existing ones for applications in which many concurrent transactions frequently insert and delete records into/from a table, and even for applications where data pages rarely have space reserved by more than one transaction.

1 Introduction

Space reservation[12, 13] is an important concept in the allocating and releasing of storage space in DBMSs that provide record locking with flexible storage management to support recoverable actions. Methods[8, 12] to handle space reservation have been developed, but most of the existing methods are designed to be optimized for applications in which few storage pages have space reserved by more than one transaction.

However, in many DBMS applications, including newly emerging applications, it is commonplace that a large number of concurrent transactions frequently insert and delete records into/from a relatively small table, and therefore many storage pages are reserved by many transactions. For example, a data container placed between the steps in a workflow management system[3, 4, 11] is used in transferring data between steps and is often implemented using tables. Transferring a datum through the container is realized by inserting a data record and subsequently deleting it from the table. In inventory control systems and order processing systems, a table is used to store order records received from clients. Order records exist in the table only until they are processed by servers. Many clients and servers concurrently insert and delete order records into/from the order table. A queue[1, 6] has played an important role in transaction processing systems, and has recently been introduced to DBMSs, such as ORACLE$^{\mathrm{TM}}$[5, 14]. Enqueue and dequeue are the main operations applied to the queue and can be implemented by inserting and deleting of data records. For these kinds of applications the existing methods are not suitable, and hence there is a need for a new space reservation method.

In this paper, we propose a new space reservation method that gives better performance for these applications as well as for traditional applications. This new method exploits the extended control structure of the lock manager. We also present simulation results which show that the proposed method provides better performance than existing ones, especially when many concurrent transactions frequently insert and delete records into/from the table.

The paper is organized as follows: Section 2 briefly explains space reservation and some existing space reservation methods. This section also discusses the problems that could arise when existing methods are used for the applications considered in this paper. Section 3 presents a new space reservation method, and explains qualitatively why it performs better than others under these applications. Section 4 evaluates the performance of the proposed method and compares it to those of existing ones through simulations. Finally, our conclusions are presented in Section 5, followed by proposals for future work.

2 Space Reservation Methods

2.1 Space Reservation Problem

When a transaction deletes or shrinks a record in a storage disk, it releases the space occupied by the record[1]. This released space, called a *reserved space*, must not be allocated for other transactions until the space-releasing transaction commits because, should the transaction abort, the space might be re-used in order to rollback the action of space-releasing.

Let us assume that a transaction T_1 releases 50 bytes of space in a page P that has 100 bytes of free space. In that case, the page P has 150 bytes of free space. Now, suppose another transaction T_2 allocates 80 bytes of space from the page P and commits. Then, the page P has only 70 bytes of free space. What happens if transaction T_1 is doomed to abort its work? Transaction T_1 fails to undo the space-releasing action in page P since it has only 70 bytes of free space, whereas T_1 needs 100 bytes of free space to undo its actions.

To avoid this problem, the space released by the transaction T_1 in the page P must be reserved, as long as it is running. A test, called a *reservation test (R-test)*, is introduced to prevent a transaction allocating space reserved for other transactions. Only transactions that pass the R-test in a page can allocate free space from the page. The allocator that fails an R-test on a page has to seek another page. To allow R-tests to make the right decisions, space reservation methods keep the information about the reserved state for each page that has reserved space. Most existing methods do not keep detailed reservation information, to avoid high costs paid in managing the information. Thus, due to the limited information, R-tests cannot know exactly how much reserved space the page has, which would lead R-tests to make wrong decisions; that is, R-tests may reject the space allocation requests on a page even though it has enough free space, excluding reserved area. We refer to such R-tests '*wasted*'. In traditional applications, wasted R-tests rarely occur since most storage pages in a table are not in a reserved state or, if so, are reserved by at most one transaction. Therefore the

[1]In this paper, we consider only the space released by removing records. However, the results of this paper can also be applied for the space released when records shrink.

problems caused by wasted R-tests have rarely been considered.

Space reservation methods can be classified into two categories: *pessimistic* and *optimistic*. In pessimistic methods, an R-test is performed on a target page before it is placed in the buffer, and it is brought into the buffer only after the R-test is passed. In optimistic methods, an R-test is performed on the page after it is brought into the buffer. A useless buffer call might occur when an R-test on the page fails, since allocation cannot be performed in the failed page.

The reserved space set aside for a transaction is released after the reserving transaction terminates. Techniques for updating reservation information upon a reserver's termination fall into two categories[9]: *eager* and *lazy*. An eager reservation method updates all related reservation information immediately after a reserver terminates. A lazy reservation method does not update reservation information upon a reserver's termination; instead, the update is usually delayed until another transaction finds the page having no reserved space. The lazy methods, while reducing the overhead spent upon transaction termination, might suffer from wasted R-tests. This is because the delayed update of reserved states tends to exaggerate the reserved state and could cause R-tests to make wrong decisions.

We assume that *free space inventory pages (FSIPs)*[2] are used in finding a page eligible for an insertion. FSIPs are the pages that contain summary information on the amount of free space in each of the set of other pages[9, 13], and they are consulted by inserters in finding a page with sufficient free space. However, a space allocation on the page suggested by FSIPs may sometimes fail since FSIPs keep only the summary information and do not consider space reservation.

2.2 Existing Space Reservation Methods

2.2.1 Basic Method: `BASIC`

In `BASIC`, R-tests rely on lock compatibility tests. A transaction that releases a space in a page locks the page in `IX` mode for reservation. This lock must be commit-duration[13] because the reserved space must be kept until the reserver terminates. A transaction that wants to

[2]They are also called Space Map Pages (SMPs).

allocate space on a page locks the page in `EX` mode, and performs a space-allocating operation only after the lock is granted; `BASIC` is a pessimistic method. When the page already has some reserved space, the `EX` mode lock is not granted immediately since the page has already been locked in `IX` mode. The space allocator locks the page in conditional and instant mode so as not to be blocked if it already has reserved space, and to unlock it immediately after the lock is granted since the lock is requested only to check whether it has reserved space or not. From now on, we will call such locks, requested for reserving space and testing reservation, *reservation locks (R-locks).* `BASIC` needs an R-lock call for each space reservation and R-test.

Since all locks are released upon the termination of a transaction, all reserved space pertaining to the transaction is released automatically; `BASIC` is an eager method. Since R-tests are performed using only lock compatibility tests, any R-tests on pages having reserved space fail, regardless of the amount of reserved space in it, causing many R-tests to be wasted. However, there is one exception: an allocator that is the only reserver in a page can pass the R-test since it is the owner of the lock granted in `IX` mode.

2.2.2 Starburst method: `STAR`

`STAR` [8] uses three fields, `FREE`, `RSVD`, `TRSVD`, and `TRANS`, in the header of each data page. The fields `FREE` and `RSVD` keep the total amount of free space and reserved space of the page, respectively. The fields `TRANS` and `TRSVD` keep the identifier of the most recent transaction that reserved space and the amount of the space in the same page respectively. To keep track of the reserved state, these fields are updated as transactions allocate and release space in the page. A transaction that needs B bytes of space is allowed to allocate space if $B \leq \texttt{FREE} - \texttt{RSVD}$. If the identifier (id) of an allocator is the same as `TRANS`, space allocation is allowed if $B \leq \texttt{FREE} - \texttt{RSVD} + \texttt{TRSVD}$, since the transaction can use '`TRSVD`' bytes more of space reserved by itself. This is the only case when a transaction can re-use its already reserved space. Since these fields are stored in data pages, data pages must be brought into buffer before carrying out R-tests; `STAR` is an optimistic method.

The main advantage of `STAR` is that no R-locks are required for space reservations and R-tests. By keeping detailed information on reserved space in a page, `STAR` permits the transaction to use a page that already has some space reserved by others. Upon its termination, a space reserver does not visit each page containing its reserved space to update the field `RSVD`; `STAR` is a lazy method. The field `RSVD` is updated to be zero when the page is met by a transaction whose `TRANS` is less than the ids of all active transactions. This reduces the costs paid upon transaction termination, but it makes `RSVD` larger than the size of the actual reserved space and it causes some R-tests to make wrong decisions. `STAR`, being optimistic, wastes one buffer call for each failed R-test.

2.2.3 C. Mohan's Method: `CMOHAN`

`CMOHAN` assumes that each data page is slotted and has two fields in its header called `TFS` (Total Free Space) and `CFS` (Contiguous Free Space). `TFS` holds the total amount of free space and `CFS` holds the size of the last free slot. `NFS` (Non-contiguous Free Space) is defined as `TFS` − `CFS`.

To track reserved states, two bits, `RSB1` and `RSB2`, are placed in the header of each page. `RSB1` has a value of '1' if some of the free spaces in `NFS` is in a reserved state, and `RSB2` has a value of '1' if some of the free space in `CFS` is in the reserved state. A transaction, which needs B bytes of space from a page, examines the header of the page:

- If the values of both `RSB1` and `RSB2` are 0, the transaction can allocate space whenever $B \leq$ `TFS`, because this means that there is no reserved space in this page.
- If the value of `RSB2` is 0, but `RSB1` is 1, the transaction can take up some space in `CFS` whenever $B \leq$ `CFS`, because this means that there is no reserved space in `CFS`.
- If the values of both `RSB1` and `RSB2` are 1, the page might have reserved space in both `NFS` and `CFS`; therefore an R-lock is requested to make sure that there is no reserved space in the page. If the lock is granted immediately and $B \leq$ `TFS`, the transaction is allowed to take up the space in this page.

Since `CMOHAN` uses `RSB1` and `RSB2`, the target page must be brought into the buffer before an R-test is performed; `CMOHAN` is optimistic. `CMOHAN` is lazy in updating `RSB1` and `RSB2` upon transaction termination, since revisiting and updating the corresponding pages incurs high execution overheads. Instead, these two bits are reset when any transaction that visits a page is sure that the page has no reserved space, using the *Commit_LSN*[10] technique, or the hint from the lock manager and the buffer control blocks.

`CMOHAN` is designed to perform well in cases where few pages in the table are reserved by more than one transaction. It avoids many R-lock calls in such situations since the values of `RSB1` and `RSB2` of pages are likely to be 0. `CMOHAN` allows, though in a limited way, a transaction that has reserved some space already in a page to re-use that space during a subsequent space-allocating operation involving the same page. However, in `CMOHAN`, many R-tests can be wasted because `RSB1`, `RSB2` and R-locks cannot describe the details of the reservation state. Moreover, since `CMOHAN` is lazy, additional R-tests may be wasted due to incorrect values of `RSB1` and `RSB2`. Unfortunately, the overhead paid when an R-test fails is higher than in other methods. For each failed R-test, `CMOHAN` spends both one buffer call and one R-lock, whereas `BASIC` spends only one R-lock and `STAR` spends only one buffer call.

2.3 Discussion on Existing Space Reservation Methods

In the existing reservation methods, R-tests may be wasted because they use inaccurate and out-dated information on the reservation states of pages. Since wasted R-tests rarely appear in most traditional DBMS applications, the problems caused by wasted R-tests have so far been ignored. However, many R-tests will be wasted when these methods are used in the applications considered in this paper: many concurrent transactions frequently insert and delete records on the tables, since many pages might have space reserved by more than one transaction. Therefore the problems due to wasted R-tests should be given serious consideration, and new reservation methods that can overcome the problem must be devised.

Frequent wasted R-tests badly affect the performance of space-allocating operations, for the following reasons. First, FSIPs must be re-read to search for another target data page, and the more a transaction reads FSIPs—one of the bottlenecks in DBMSs—the longer it could block others or be blocked by others. Second, for optimistic methods such as `CMOHAN` and `STAR`, the number of wasted buffer calls for a data page increases with wasted R-tests and hence a transaction needs to read more pages for an insertion. This may enlarge the working-set of transactions and lower the performance of the system owing to the heavy paging activity. Finally, for reservation methods that use R-locks for R-tests, many R-lock calls are wasted as the number of failed R-tests increases.

In the following sections, we propose a new space reservation method that eliminates avoidable wasted R-tests, and show that this new method outperforms the others for applications where many concurrent transactions frequently insert and delete records into/from a table.

3 A Proposed Space Reservation Method: `NEW`

This section presents an eager and pessimistic space reservation method, which extends the lock table in order to keep the details of the reservation of pages. The lock table, consisting of lock headers and lock entries, keeps track of the locked data. A lock header keeps the state of a locked object, such as the name of the lock, the aggregate lock mode and a queue of lock entries. A lock entry is assigned to each lock requester and contains information on the requester, such as the requested lock mode, the held lock mode and the status of the lock: granted or waiting[7, 16].

Like `BASIC`, `NEW` locks a page to reserve space and to perform an R-test on that page. The two fields `TFS` and `RSVD` are introduced in the lock header, allocated to a locked page, to keep the information on available space on the page. They keep the total free space and the total reserved space of the corresponding page, respectively. For each transaction that reserves space on a page, a lock entry is allocated for the page. The lock entry is also extended, by attaching `TRSVD`, to hold the total amount of space that the transaction has reserved so far in the page. For instance, Figure 1 shows that two pages $P1$ and $P2$ have reserved space. Page $P1$ has 3500 bytes of free space, including 1000 bytes of reserved space occupied by transactions $T1$, $T2$ and $T3$. Transaction $T1$ is the only reserver in the page $P2$, where there are 500 bytes of free space.

Algorithm 1 shows the procedure that releases B bytes of space from page P. The fields

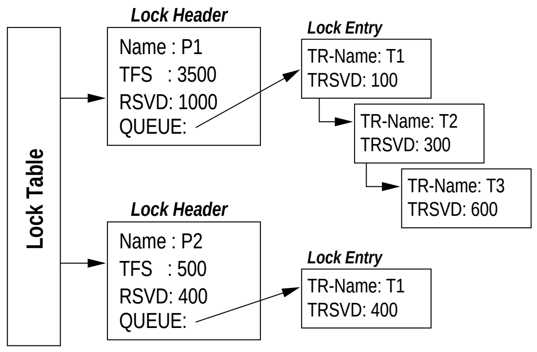

Figure 1: The extended control structure for lock table

`TFS` and `RSVD` in the lock header and `TRSVD` in the lock entry are updated according to the amount of space released, B.

Algorithm 1 Releasing and reserving B bytes of space in the page P

```
1: release $B$ bytes in page $P$
2: try to find the lock header $H$ for page $P$
3: if not found then
4:     allocate a new lock header $H$ for page $P$
5:     $H.RSVD \leftarrow 0$          # the page $P$ has no reserved space.
6: end if
7: $H.TFS \leftarrow H.TFS + B$        # the total free space (TFS) expands.
8: $H.RSVD \leftarrow H.RSVD + B$  # the released space is reserved.
9: try to find lock entry $E$ of transaction $T$ in the $H.QUEUE$.
10: if not found then
11:     allocate a new lock entry $E$ for transaction $T$ in the $H.QUEUE$.
12:     $E.TRSVD \leftarrow 0$         # the transaction has no reserved space in this page.
13: end if
14: $E.TRSVD \leftarrow E.TRSVD + B$   # the released space is reserved for transaction $T$.
```

Algorithm 2 shows a procedure that occupies B bytes of space on page P. Since a transaction can use all of the space except for the portion reserved by others, an R-test is passed whenever $B \leq \texttt{TFS} - \texttt{RSVD} + \texttt{TRSVD}$. This means that whenever possible, in contrast to existing methods, `NEW` allows a transaction to re-use space that has already been reserved by itself during subsequent space-allocating operations involving the same page. Therefore, `NEW` permits a transaction to allocate the maximum amount of space available to itself, and hence eliminates wasted R-tests that could appear in the existing reservation methods. Since space allocation is always performed after the R-test is passed, useless buffer calls, due to failed R-tests, are never produced. Fields `TFS` and `TRSVD` are updated appropriately after the space allocation is successfully completed. No buffer calls are required for the update since these fields are resident in memory.

If a transaction performs an R-test on a page in which it has not yet reserved space, no lock entries exist for the transaction on that page. In this case, `TRSVD` is assumed to be 0. Therefore, the R-test is passed if $B \geq \texttt{TFS} - \texttt{RSVD}$(5–7). If the page has no reserved space, the corresponding lock header might not be found in the lock table. In this case, both `RSVD` and `TRSVD` are assumed to be 0, and `TFS` is assumed to be the physical page size (`PAGESIZE`)(8–10) and therefore, `TFS` − `RSVD` − `TRSVD` becomes `PAGESIZE`. This means an R-test is always successful because B is less than `PAGESIZE`[3]

Algorithm 2 Allocating of B bytes of space in page P by transaction T

```
1: try to find the lock header $H$ and the lock entry $E$ for the page $P$ and transaction $T$
2: if both are found then
3:     # the transaction $T$ has already reserved '$E.TRSVD$' bytes of space in the page $P$.
4:     $availsize \leftarrow H.TFS - H.RSVD + E.TRSVD$
5: else if only $H$ is found then
6:     # the transaction $T$ has reserved no space in page $P$.
7:     $availsize \leftarrow H.TFS - H.RSVD$
8: else
9:     # the page $P$ has no reserved space at all.
10:    $availsize \leftarrow PAGESIZE$
11: end if
12: if $B > availsize$ then
13:    allocation has failed so try another page
14: end if
15: fix page $P$ in the buffer and try to allocate $B$ bytes from the page
16: if allocation has failed then
17:    unfix the page and try another page
18: end if
19: # the transaction $T$ prefers to use its reserved space.
20: $H.RSVD \leftarrow H.RSVD - min(E.TRSVD, B)$
21: $E.TRSVD \leftarrow E.TRSVD - min(E.TRSVD, B)$
22: $H.TFS \leftarrow H.TFS - B$
23: unfix the page
```

On terminating, a transaction releases all reserved space that it has. This work can be integrated with the procedure that the transaction releases all commit-duration locks granted to itself. Algorithm 3 shows this integrated procedure. Unlike `STAR` and `CMOHAN`, the reserved space information is updated immediately after a reserving transaction commits.

We believe that `NEW` is suitable for applications in which many concurrent transactions frequently insert and delete records into/from the tables, because it eliminates wasted R-tests

[3]In this paper, we do not consider allocating space larger than the physical page size.

Algorithm 3 Integrated "unlock-all" procedure for transaction T

```
1: for all lock entries $E$ allocated to transaction $T$ do
2:     if $E$ is allocated for reserving space then
3:         find the corresponding lock header $H$ of $E$.
4:         $H.RSVD \leftarrow H.RSVD - E.TRSVD$   # the reserved space shrinks.
5:     else
6:         release the lock $E$.
7:     end if
8: end for
```

and useless buffer calls that could often occur in these applications if existing methods are used.

As mentioned in Section 2.1, FSIPs sometimes suggest wrong pages during space allocation, and this may cause useless buffer calls. `NEW`, however, can filter these pages out before the page is placed in the buffer, since the exact total free space (`TFS`) is known to R-tests. According to Algorithm 2, such pages fail R-tests. This allows `NEW` to further reduce the number of useless buffer calls. However, since `TFS` is provided for only the pages whose lock headers exist in the lock table, space allocation on a page having no reserved space might suffer useless buffer calls as in other methods. To minimize such useless buffer calls, the lock header of the page is not returned to the pre-allocated pool, even if the page becomes free of reserved space, but instead is kept in the lock table until the pre-allocated pool becomes empty. Some of the useless buffer calls are avoided by this modified scheme.

`NEW`, like `BASIC`, needs a lock call for every space reservation and R-test, but we consider that this overhead is relatively low when compared to the overall costs of inserting and deleting a record on the table. `NEW` also requires additional shared memory resources for extended lock headers and lock entries. However, this will not impact on the system, given the memory available in current DBMSs. In the next section we will show by simulation that our method outperforms other methods, especially when there are many transactions inserting and deleting in the system.

Parameter	Meaning	Assigned Value
`CPUMIPS`	Instruction rate of CPU	50 MIPS
`NClients`	Number of clients	1–50 clients
`PAGESIZE`	Size of a page	2K bytes
`MaxRecSize`	Maximum record size	250 bytes
`MinRecSize`	Minimum record size	150 bytes
`MaxTransSz`	Maximum no. of records inserted/deleted	10
`MinTransSz`	Minimum no. of records inserted/deleted	5
`MaxDiskTime`	Maximum disk access time	30 milliseconds
`MinDiskTime`	Minimum disk access time	10 milliseconds
`LogDiskTime`	Log disk access time	5 milliseconds
`LogGenInstr`	No. of instr. per generating a log record	500 instructions
`FSIPUpdInstr`	No. of instr. per update an entry in FSIP	10 instructions
`FixInstr`	No. of instr. per fix/unfix pair	200 instructions
`RLockNEWInstr`	No. of instr. per R-lock/unlock pair in NEW	250 instructions
`RLockInstr`	No. of instr. per R-lock/unlock pair in other methods	220 instructions
`RecInsInstr`	No. of instr. per inserting a record	1,000 instructions
`RecDelInstr`	No. of instr. per deleting a record	1,000 instructions
`CompactInstr`	No. of instr. per compaction	2,000 instructions
`InsFailInstr`	No. of instr. per failed insertion	50 instructions
`CommitOvhd`	No. of instr. per transaction commit	1,000 instructions

Table 1: Simulation parameters and their values

4 Simulation

4.1 Simulation Model

We have developed a simulation model using C++SIM[2] to evaluate the performance of `NEW` and compared it with that of other methods. Three workloads, SMALL-FF, LARGE-NF and QUEUE, are developed for evaluation. Workload-independent simulation parameters are listed in Table 1 with their assigned values.

The simulation models a transaction processing system where clients issue transactions that insert and delete records into/from a table. Each client executes only one transaction at a time; and the new transaction is not started until the running one terminates. The number of clients (`NClients`) varies from 1 to 50 and defines the number of concurrent transactions. The size of a transaction is defined by the number of insertions and deletions performed by the transaction and is selected randomly in the range of `MinTransSz` to `MaxTransSz`.

The table is implemented as a collection of pages of `PAGESIZE` bytes each, and stored in a disk. The access time of the disk is selected randomly in the range of `MinDiskTime` to `MaxDiskTime` milliseconds for each I/O operation. The size of the record to be inserted is chosen randomly in the range `MinRecSize` to `MaxRecSize` bytes. A log disk is provided to store the log records generated during simulation, the access time of which is defined as `LogDiskTime` milliseconds. `LogGenInstr` specifies the number of instructions required to generate a log record.

`FixInstr` specifies the number of instructions required to fix the page into the buffer, including the unfix cost to subsequently release the fixed page. `RecInsInstr` and `RecDelInstr` are, respectively, the instructions required to allocate and release space in the page fixed in the buffer. Sometimes, a space-allocating operation requires a compaction of free space slots scattered over the page to make a large one. `CompactInstr` is the cost for a compaction action. `InsFailInstr` is the overhead spent when the page, fixed in the buffer, is found to have insufficient available space at record insertion time.

When free space in a page is changed, the corresponding FSIP entry might be updated to keep up with the page. The ways to update FSIPs in all space reservation methods are same. `FSIPUpdInstr` is the cost required to update an entry in the FSIP. `RLockInstr` is the number of instructions for an R-lock required to reserve space and an R-test for the methods `BASIC`, `STAR` and `CMOHAN`. For `NEW`, we assigned `RLockInstrNEW` instructions for an R-lock since it pays more instructions to keep reservation information in the lock table.

For the workloads SMALL-FF and LARGE-NF, we assume that the table for inserting/deleting records is implemented by a heap file. The heap file is a file structure on which no special rules are imposed when a record is inserted, unlike a keyed-sequential file or an indexed file where the position (i.e. page) to be inserted is defined from the key value of a record. Many techniques, called *page allocation*, have been developed, whose purpose is to choose a page when a record is inserted into a heap file with the objective of maximizing space utilization and minimizing search time.

We have implemented most of the existing page allocations known from the literature: First-Fit (FF), Next-Fit (NF), a variation of FF used in DB2 (DB2), and NF with four witnesses (WH), and have carried out simulations under each scheme. FF searches for a target from the beginning of the file, and selects a page that has enough free space for a new record to be inserted. NF searches for a target page from the page where the last record was inserted. When failing to find a page, it continues searching from the beginning of the file. DB2 works like FF if the size of the newly inserted record is less than that of the last inserted one; otherwise it works like NF. WH first refers to witness, the in-memory table holding candidate pages indexed by their free space sizes, to find the target page, and if this fails, it works like NF. More details of these techniques are given and their performances are extensively evaluated in [9].

4.2 Workload SMALL-FF: Small File with Small Buffer Pools

Due to the space limitations of this paper, we present only the results of an experiment carried out under FF for this workload. Experiments under other page allocation techniques have been carried out for this workload and show similar results. FF was chosen because it shows the benefits of `NEW` most clearly and it is easy to explain why `NEW` gives better performance than other space reservation methods under FF.

Workload SMALL-FF is designed for the applications considered in this paper in which many concurrent transactions frequently insert and delete records into/from a small table. The table has some records (130) before the experiment starts. To penalize methods that generate many useless buffer calls, only 32 buffer pages are assigned. Under this workload, many pages are expected to be reserved by more than one transaction at a time, which is favorable to `NEW`.

Figure 2 shows the number of failed R-tests generated in inserting a record into the table. As more clients are added to the system, `BASIC`, `CMOHAN` and `STAR` produce more than fourteen failed R-tests per insertion, while `NEW` generates only about three failed R-tests. This shows that using detailed and up-to-date reservation information during R-tests is very effective in reducing the number of failed R-tests. In fact, in `CMOHAN` and `STAR` more than twelve R-tests are wasted due to inaccurate and out-dated reservation information.

Figure 3 shows the number of wasted buffer calls produced per insertion. In `CMOHAN` and

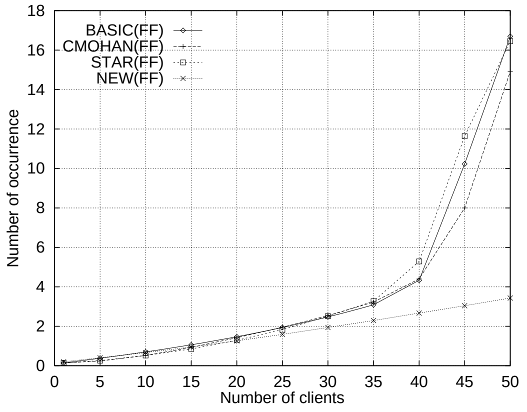

Figure 2: Number of failed R-tests generated per insertion

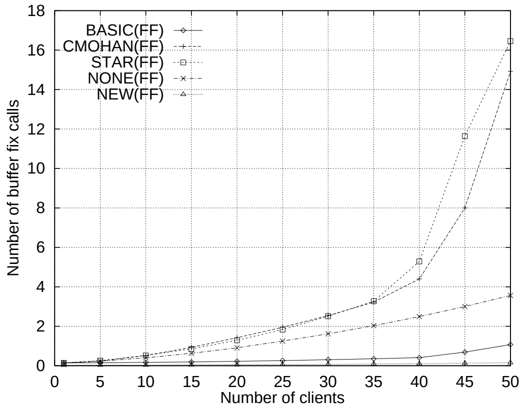

Figure 3: Number of wasted buffer calls generated per insertion

Reservation Method	`Avoidable`	`FSIP`	`Failed Lock`	Total
`BASIC`	0	1.07	0	1.07
`CMOHAN`	13.18	0.95	0.01	14.14
`STAR`	12.83	4	0	16.83
`NONE`	0	0.69	2.88	3.57
`NEW`	0	0.01	0.14	0.15

Table 2: A breakdown in the wasted buffer calls of 50 clients under SMALL-FF

`STAR`, the number of buffer calls increases as the number of failed R-tests increases, because these methods are optimistic. Therefore, every failed R-test results in useless buffer calls. Since `NEW` is pessimistic, the number of wasted buffer calls per insertion is hardly affected by the number of failed R-tests, and remains almost 0 which is ideal. Though `BASIC` is also pessimistic, the number of buffer calls is slightly greater than that in `NEW`, since FSIPs sometimes suggest wrong pages (1.07 times per insertion).

In this figure, we plot the number of buffer calls for the case (`NONE`) where no space reservation is applied during a record insertion. Interestingly, the number of buffer calls spent in `NONE` is larger than those in `NEW` and `BASIC`. This is due to the fact that an insertion could fail because a lock for the newly inserted record is not granted immediately. If the name of the newly inserted record is the same as that of the one deleted by another active transaction, the record-level lock for this record is not granted immediately because the deleter had already locked the record in the exclusive mode. We assume, for this case, that the transaction will seek another page for insertion instead of waiting for the lock. Table 2 shows a breakdown of wasted buffer calls issued for 50 clients. The number under the column `Avoidable` means the number of wasted buffer calls that could have been avoided if the R-test had been performed before the page was brought into the buffer. The number under the column `FSIP` denotes the number of wasted buffer calls issued because the FSIPs suggested wrong pages. Finally, the number under the column `Failed Lock` denotes the number of wasted buffer calls issued because record lock was not granted immediately.

This table shows that most of the wasted buffer calls in `NONE` are due to the failed record lock. It also shows that most of the wasted buffer calls in `CMOHAN` and `STAR` originate from the fact that data pages are brought into the buffer before R-tests are executed. The pessimistic reservation method is far more effective than the optimistic one when many pages are reserved by more than one transaction. The difference in the number of wasted buffer calls between `BASIC` and `NEW` is due to the pages wrongly suggested by FSIPs. Since `NEW` can use the exact total free space size of the page from the corresponding lock header during R-tests, it can filter these pages out and the R-test prevents them from being read into the buffer.

Figure 4 shows the number of R-locks requested in inserting a record. In `BASIC` and `NEW`, the number of R-locks plots the same curve as that in Figure 2, since an R-lock is required for each R-test. `STAR`, which does not use locks for R-tests, shows no R-lock calls. In `CMOHAN`, owing to the optimization technique, the number of R-locks starts at 0 and remains less than 1, until 20 clients are added to the system. However, the number then rapidly increases and becomes larger than that in `NEW` when the number of clients exceeds 35. This shows that the benefits gained from the optimization of `CMOHAN` are offset by the wasted R-tests due to wrong decisions, as more pages have space reserved by more than one transaction. `NEW` saves more R-locks than `CMOHAN` in this situation.

Figure 5 shows the elapsed transaction execution time. In all methods except `NEW`, the execution time increases as more than 35 clients are added to the system. This is because `BASIC`, `CMOHAN` and `STAR` suffer heavy paging activity as more pages are brought into the buffer per insertion, as shown in Figure 3, which makes the total working-set of transactions larger than the buffer pool size. Figure 6 shows the average number of accesses of different pages per

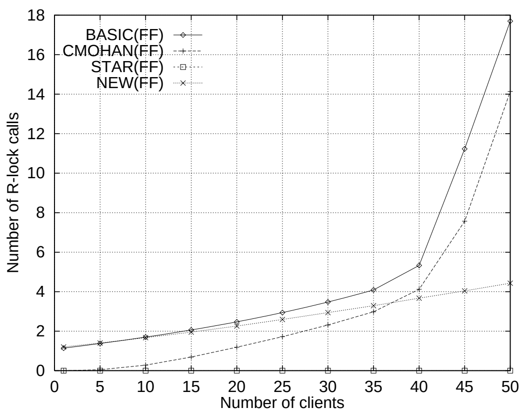

Figure 4: Number of R-locks to insert a record

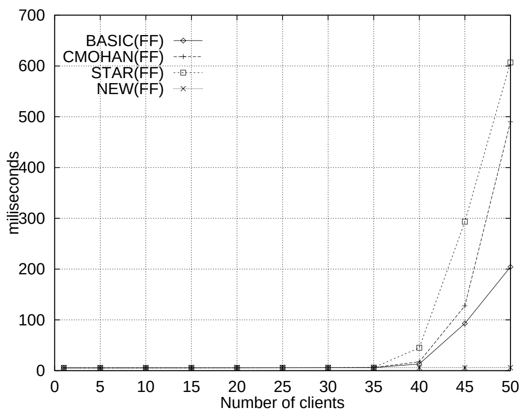

Figure 5: Elapsed transaction processing time

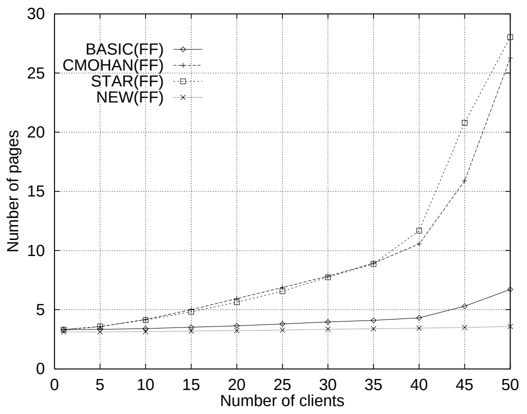

Figure 6: Number of pages touched

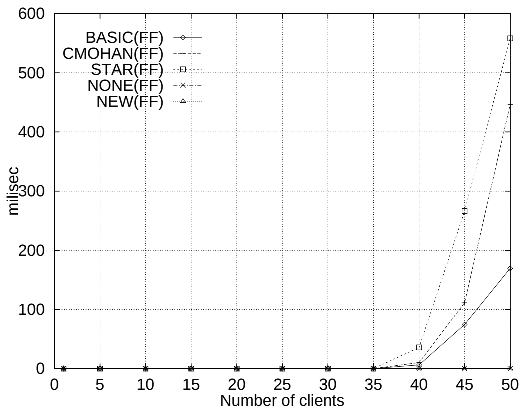

Figure 7: I/O waiting time per transaction

transaction. For CMOHAN and STAR, the number of different pages accessed increases rapidly and exceeds 26. Accessing only the pages that pass the R-test, NEW and BASIC show a far fewer number of different pages accessed than the others. Moreover, since NEW does not suffer wasted buffer calls due to pages wrongly suggested by FSIPs, the number of pages touched remains the same though the number of clients increases. Figure 7 shows the average amount of waiting time for I/Os experienced per transaction. When comparing this with Figure 5, we can confirm that the gaps in elapsed transaction execution time between NEW and the other methods mainly originate from the I/O waiting time gap.

In this experiment, we find that CMOHAN, STAR and BASIC perform very poorly when compared to NEW for applications where many concurrent transactions frequently insert and delete records on a relatively small table. This is because R-tests using inaccurate and out-dated information on reserved pages frequently make wrong decisions. Moreover, optimistic methods such as `CMOHAN` and `STAR` can trigger a heavy paging activity, since they also produce many buffer calls during insertions as the number of failed R-tests increases. In contrast, `NEW` gives very good performance for these applications, since the R-tests always make the right decisions using detailed and up-to-date reservation information and only the pages that are passed by R-tests are accessed, which minimizes the buffer calls required during insertions.

4.3 Workload LARGE-NF: Large File with Large Buffer Pools

In workload LARGE-NF, we want to evaluate the performance of `NEW` for cases in which the benefits of `NEW` are minimized while the benefits of other methods are maximized. First, in order for each page to be reserved by at most one transaction, we assigned a set of pages in the table to each client, which is allowed to delete records only from the dedicated set of pages. Second, we start the simulation using a table that has many more data pages, and deletions are performed on a wider range of pages so that fewer pages are in the reserved state at one time, which increases the chance of meeting a non-reserved page during insertions. Third, the buffer pool has many buffer slots to eliminate performance degradation due to high paging activity, which is the main cause of the reduced performance of the traditional methods in workload SMALL-FF. During this experiment, there was no paging activity for buffer replacements. Finally, for this workload we carried out this experiment under NF page allocation. The NF method prevents each client starting to find data pages from the first page in the table, and consequently the insertion point floats over the table as records are inserted. Since each client uses its own insertion point, fewer clients at a time start from the same page for insertions. These features reduce the number of failed R-tests, and therefore are beneficial to `CMOHAN`, `BASIC` and `STAR`.

Figure 8 shows the number of failed R-tests in inserting a record. Since few pages have space reserved by more than one transaction, the number of failed R-tests is considerably reduced for all reservation methods, when compared to workload SMALL-FF. `NEW` still shows a minimum number of failed R-tests as the number of active transactions increases. Figure 9 shows the number of wasted buffer calls issued in inserting a record, and Table 3 shows a

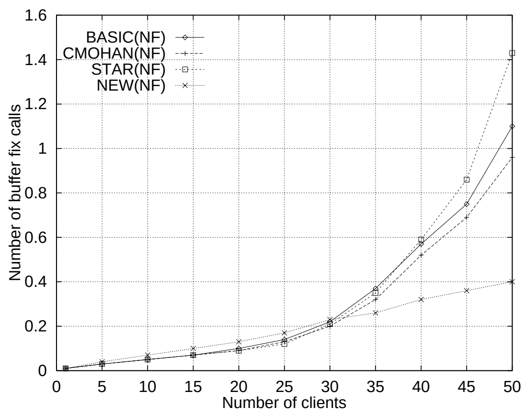

Figure 8: Number of failed R-tests per insertion

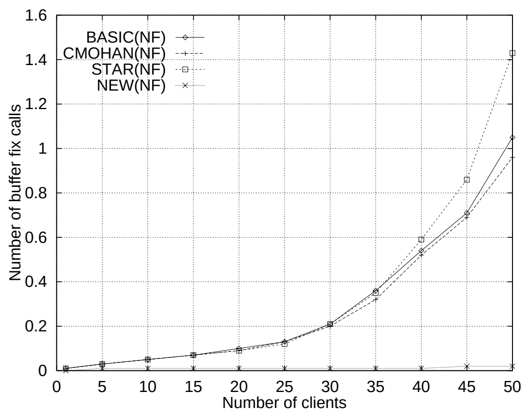

Figure 9: Number of buffer calls to insert a record

Reservation Method	Avoidable	FSIP	Failed Lock	Total
`BASIC`	0	1.05	0	1.05
`CMOHAN`	0.04	0.91	0	0.95
`STAR`	0.14	1.29	0	1.43
`NEW`	0	0.01	0	0.01

Table 3: A breakdown of wasted buffer calls at 50 clients under LARGE-NF

breakdown in wasted buffer calls issued during insertion of 50 clients. The number of wasted buffer calls that can be avoided is significantly reduced because most R-tests are passed for this workload. Instead, the most useless buffer calls are wasted due to wrong pages being suggested by FSIPs. Since those pages wrongly suggested by FSIPs are recognized only after they are brought into the buffer, `BASIC`, though pessimistic, shows as many wasted buffer calls as `CMOHAN` and `STAR`. On the other hand, in `NEW`, since most of the pages wrongly suggested by FSIPs are filtered out during R-tests, almost no wasted buffer calls are generated[4], and therefore `NEW` yields the fewest useless buffer calls among reservation methods.

Figure 10 shows the number of R-locks issued in inserting a record by each space reservation method as clients are added to the system. `CMOHAN` saves many R-locks and shows almost 0 R-locks even when many clients are added to the system. Although we do not present the graph in this paper, `CMOHAN` also generates fewer than 0.4 R-locks for a record deletion. This

[4]In fact, `NEW` can filter out all pages wrongly suggested by FSIPs and guarantee no wasted buffer calls from this cause. However, to avoid the complex implementation of the R-test in `NEW`, we reduced the level of R-test requirements when we implemented `NEW` in this simulation.

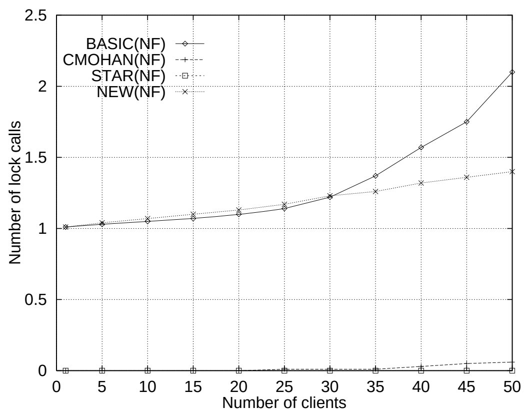

Figure 10: Number of R-lock calls per record insertion

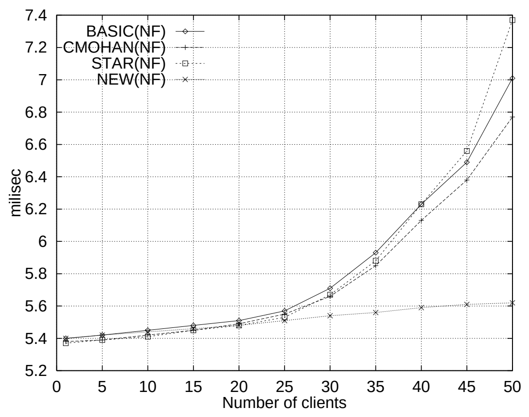

Figure 11: Elapsed transaction processing time

implies that `CMOHAN` reduces R-locks most when pages are rarely reserved by more than one transaction. For both `BASIC` and `NEW`, the number of R-locks increases as the number of failed R-tests increases. Since `BASIC` generates more failed R-tests, the number of R-locks for `BASIC` increases faster than that for `NEW`.

Figure 11 and Figure 12 show the elapsed transaction execution time and the number of transactions processed per second, respectively. `NEW` shows a 1.2–1.3 times better performance result than others, and all methods except `NEW` plot curves similar to one another, which implies that the performance gains from saving R-locks in `CMOHAN` and `STAR` have little effect on the performance of transactions as the number of clients increases. Instead, we think that the performance in this workload is mainly dependent on the buffer page latch waiting time. A failed R-test requires additional access to FSIPs and other data pages. The more a transaction fixes data pages and FSIPs, the greater the chance of the transaction blocking other transactions or being blocked by others, since fixing data pages and FSIPs requires EX mode and SH mode latches of the page, respectively. Moreover, block time would be relatively long for data pages because it takes a long time to complete the insertion of a record onto a page; it includes the generation of a log record, lock requests, sometimes space compaction and FSIP update.

Figure 13 plots the overall buffer latch waiting time experienced by a transaction. The time gap between `NEW` and other methods is 1.14–1.72 milliseconds per transaction of 50

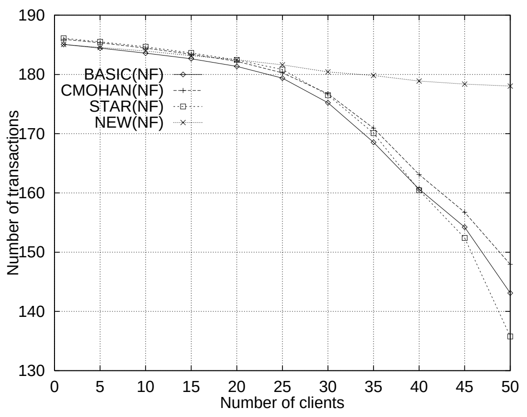

Figure 12: Number of transactions processed per second

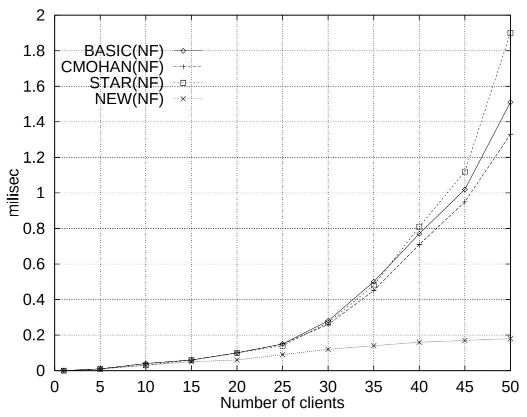

Figure 13: Total buffer page waiting time per transaction

clients, which is almost the same as the elapsed time gap as shown in Figure 11, where it is 1.15–1.75 milliseconds per transaction.

From the result of this experiment, we found that the effects of failed R-tests are minimal for applications in which few pages are reserved. However, `NEW` is still a good reservation method for these kinds of applications because it eliminates some of the wasted buffer calls due to wrongly suggested FSIPs, and shortens buffer page latch waiting time when many concurrent transactions frequently insert and delete records on the table.

4.4 Workload QUEUE: Queue with Large Buffer Pools

In workload QUEUE, we evaluate the performance of several space reservation methods when they are used on queues. In this experiment, we have implemented a modification of the strict-FCFS queue, where the system allows concurrent dequeue operations and ignores the occasional out-of-order dequeuing[1, 15]. Before starting the experiment, we enqueued 1000 records in the queue to prevent it from becoming empty and a dequeuing transaction failing to get a record from it. This also ensures that an enqueuing transaction hardly ever sees a reserved record, because the queue has enough records so that the head and the tail of the queue never exist on the same page of the queue. We think that this is very advantageous for `CMOHAN` and `STAR`. As for LARGE-NF, sufficient buffer pages (250) are allocated to the buffer to eliminate performance degradation from high paging activity, which is the main

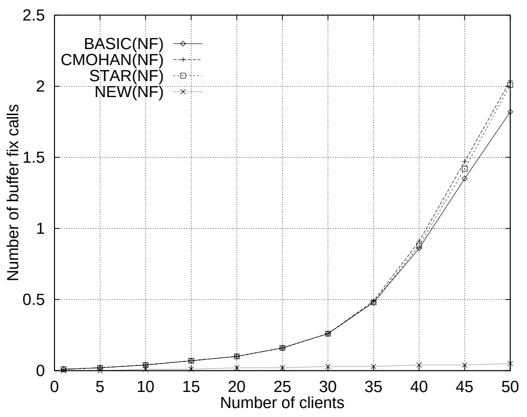

Figure 14: Number of wasted buffer calls in inserting a record

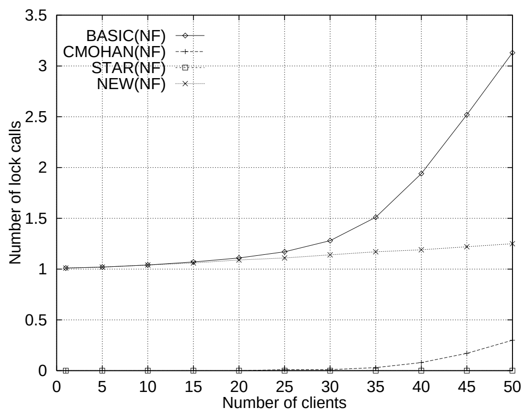

Figure 15: Number of R-lock calls per record insertion

Reservation Method	Avoidable	FSIP	Failed Lock	Total
`BASIC`	0	1.82	0	1.82
`CMOHAN`	0.27	1.76	0	2.03
`STAR`	0.38	1.63	0	2.01
`NEW`	0	0.04	0.01	0.05

Table 4: A breakdown of wasted buffer calls at 50 clients under QUEUE

cause of the reduced performance of traditional methods, as shown in SMALL-FF. During this experiment, there was no paging activity for buffer replacement.

Figure 14 shows the number of wasted buffer calls issued in inserting a record, and Table 4 shows a breakdown of wasted buffer calls issued in an insertion of 50 clients. Since few pages have space reserved by more than one transaction, the number of wasted buffer calls is considerably reduced for all reservation methods compared to workload SMALL-FF. However, `NEW` still generates the fewest wasted buffer calls as the number of active transactions increases. Most of the wasted buffer calls, as in LARGE-NF, are produced due to pages being wrongly suggested by FSIPs. Filtering out most of the wrongly suggested pages, `NEW` generates almost no wasted buffer calls.

Figure 15 shows the number of R-locks issued in inserting a record as clients are added to the system. `CMOHAN` saves many R-locks and shows almost 0 R-locks even when many clients are added, which implies that `CMOHAN` reduces the number of R-locks the most because a transaction hardly sees a reserved page during an enqueuing operation. For `BASIC` and `NEW`,

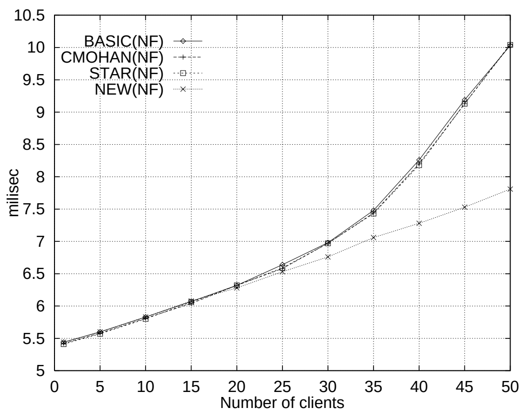

Figure 16: Elapsed transaction processing time

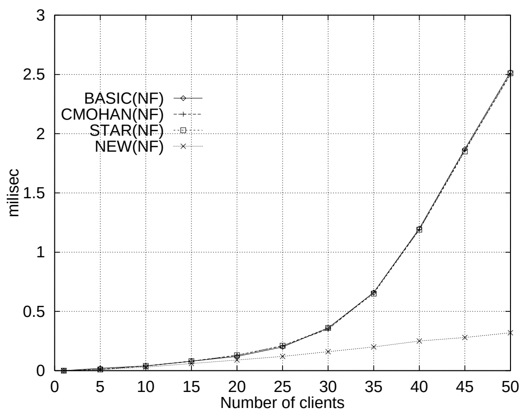

Figure 17: Total buffer page waiting time per transaction

the number of R-locks starts at 1 and increases with the number of failed R-tests.

Figure 16 and Figure 17 show the elapsed transaction execution time and the overall buffer latch waiting time experienced by a transaction, respectively. For the same reason mentioned in Section 4.3, the time gap (2.22–2.3) between `NEW` and other methods in Figure 16 is due to the buffer latching time gap (2.19–2.26) between them. In particular, this is because all transactions that try to enqueue records at one time fix the same data page of the queue (i.e. the head page of the queue). The more frequently a transaction fixes data pages, the more transactions might be blocked and the longer transactions might have to wait, because only one transaction at a time is allowed to fix the head of the queue. Also, it takes a long time to complete the enqueuing of a record in a page, including generation of a log record and lock requests.

5 Conclusions and Future Work

Space reservation must be handled carefully to perform space-allocating operations efficiently. Since previous methods assume that pages in a table (or file) are rarely reserved by more than one transaction, many R-tests and useless buffer fix calls are wasted. This impacts on the performance of space-allocating operations.

We have proposed a new eager and pessimistic space reservation method, which minimizes the number of R-tests and useless buffer calls in applications where many concurrent transactions frequently insert or delete records into/from tables. This can be achieved by keeping detailed and up-to-date reservation information in the lock table. The proposed method further reduces failed buffer calls because it filters out wrong pages suggested by FSIPs. This makes the proposed method perform better than other methods even in applications in which few pages are reserved by more than one transaction.

We expect that `TFS` in the lock header will be a good source from which to select a page during record insertion, instead of using FSIPs. Therefore, we are currently working on devising a page allocation method for the heap file. This page allocation method directly accesses the fields `TFS` and `RSVD` in the lock header in order to select a page. We expect that this integration of a page allocation method and a space reservation method through the lock table would shorten the path length of the insertion routine and give a better solution to the free space management of heap files.

References

[1] Philip A. Berstein and Eric Newcomer. *Principles of Transaction Processing.* Data Management Systems. Morgan Kaufman Publishers, Inc., 1997.

[2] Department of Computing Science, University of Newcastle upon Tyne. *"C++SIM User's Guide"*, public release 1.5 edition. http://cxxsim.ncl.ac.uk.

[3] Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. "Coordinating Multi-Transaction Activities". Technical Report CS-TR-247-90, Princeton University, February 1990.

[4] Hector Garcia-Molina and Kenneth Salem. "Services for a Workflow Management System". *IEEE Database Engineering bulletin*, 17(1):40–44, March 1994.

[5] Dieter Gawlick. "Messaging/Queuing in Oracle8^{TM}". In *Proc. of the Conf. on Data Engineering*, pages 66–68, February 1998.

[6] Jim Gray. "THESIS: Queues are Databases". In *HPTS 95 Position Paper*, 1995.

[7] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufman Publishers, Inc., 1993.

[8] B. G. Lindsay, C. Mohan, and M. H. Pirahesh. "Method for Reserving Space Needed for "Rollback" Actions". *IBM Technical Disclosure Bulletin*, 29(6):2743–2746, November 1986.

[9] Mark L. McAuliffe, Michael J. Carey, and Marvin H. Solomon. "Towards Effective and Efficient Free Space Management". In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 389–400, Montreal, Canada, June 1996.

[10] C. Mohan. "Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems". In *Proc. of the Conf. on VLDB*, Brisbane, Australia, August 1990.

[11] C. Mohan, G. Alonso, R. Günthör, and M. Kamath. "Exotica: A Research Perspective on Workflow Management Systems". *IEEE Database Engineering bulletin*, 18(1):19–26, March 1995.

[12] C. Mohan and Don Haderle. "Algorithms for Flexible Space Management in Transaction Systems Supporting Fine-Granulairty Locking". In *Proc. of International Conference on Extending Database Technology*, Cambridge, United Kingdom, March 1994.

[13] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging". *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[14] Oracle Corporation. *"Oracle8TM Server Application Developer's Guide"*, June 1997.

[15] Peter M. Schwarz and Alfred Z. Spector. "Synchronizing Shared Abstract Types". *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[16] Tobin J. Lehman Vibby Gottemukkala. "Locking and Latching in a Memory-Resident Database System". In *Proc. of the Conf. on VLDB*, pages 533–544, Vancouver, Canada, August 1992.