Cache Modelling in a Performance Evaluator of Parallel Database Systems

Zhou, S.; Tomov, Neven; Burger, Albert Georg; Taylor, Hamish

Publication date:
1997

Cache Modelling in a Performance Evaluator of Parallel Database Systems

S Zhou, N Tomov, M H Williams, A Burger and H Taylor

Modelling Cache Memory in a Parallel Database Performance Evaluator

Keywords: parallel DBMS, cache behaviour prediction, throughput estimation.

I. Introduction

Parallel database management systems (DBMS) are generally recognised as one of the most important application areas for commercial parallel systems today. However, the task of understanding, managing and especially predicting the performance of a parallel database system is complex. Performance depends on many factors, such as the characteristics of the DBMS, the architecture of the system, the pattern of queries, the way in which queries are optimised and the way in which items of data are distributed across the nodes of the system (data placement). A small change in configuration, data placement or workload can have a significant effect on performance. Consequently, prediction of the effect on performance of changes to various system parameters relies heavily on having an accurate model of performance measures such as throughput, response time and utilisation for given system configurations, data placement and workloads. For such a model to be realistic it must take into consideration a number of factors, among them DBMS concurrency control strategy, logging policy, background activity and cache coherency protocol.

DBMSs running on current commercial parallel machines are based either on *shared data* (or *shared disk*) model or on *partitioned data* (or *shared nothing*) model. In the case of shared data, each of the nodes of the parallel machine may access and modify data residing on the disks of any of the nodes of the configuration. In the case of partitioned data, each node owns the portion of the database that resides on its disks, and it cannot directly read or modify data owned by other nodes. In both cases the nodes maintain caches which store copies of database pages, recently accessed by transactions running on the node. This reduces I/O activity at the node, as well as remote data access. However, for parallel database systems utilising the shared data model it introduces the *cache coherency problem* [5]: since the same database page(s) may be cached at more than one node at any one time, cache coherency policies must be used to ensure that transactions only see current data. There are a number of cache coherency policies developed in the

literature, and a number of studies comparing their performance. In this paper we describe the Oracle7 Parallel Database Server cache coherency policy as implemented in the GoldRush MegaServer [7] and develop an analytical cache memory model for it. The model is developed in the context of a parallel database performance evaluation tool, called STEADY.

The rest of the paper is organised as follows. Section 2 describes STEADY and shows how the cache memory model relates to other modules of the tool. Section 3 briefly reviews several types of cache coherency policies, and discusses Oracle7 parallel cache management in detail. In section 4 we develop an analytical model of this policy. Finally, in Section 5 we present some preliminary results from our model.

II. An Analytical Parallel DBMS Performance Evaluator

STEADY (System Throughput Estimator for Advanced Database sYstems) [10], [9] is an analytical tool for performance estimation of shared-nothing parallel database systems. Apart from a graphical user interface, it consists of four major modules:

1. **DPtool** is used to generate various data placement schemes for a parallel database using different strategies;
2. The **Profiler** is a statistical tool primarily responsible for generating base relation profiles and estimating the number of tuples resulting from data operations;
3. The **Modeller** is responsible for producing the workload profile for a particular benchmark or application with assistance from the Profiler;
4. The **Evaluator** takes in workload profiles and predicts the utilisation of each resource and from it the maximum system throughput values and system bottlenecks.

Relations are partitioned into fragments by DPtool using declustering methods such as *hash* and *range*. These fragments are then allocated to processing elements (PEs) in such a way as to achieve a balance of disk access frequencies or data size across the PEs or to minimise the network traffic for transferring data. Based on the generated data placement and the chosen DBMS architecture, a workload profile for a particular benchmark or application can be generated by the Modeller, which includes an estimation of disk I/O requirements in terms of the number of pages read and written to each disk in the system. This estimation is derived on the basis of the estimated cache hit ratios for the

particular relation fragments in the cache memory of each PE. The workload profiles are then converted by the Evaluator into resource utilisation profiles to determine the system bottlenecks. From this the maximum throughput rate is determined. Thus gives users an indication on the upper limit of system capacity in terms of throughput.

III. Cache Coherency Policies

A. Overview

Cache coherency policies are studied in two somewhat different multi-system database environments. In the case of a *client/server database architecture*, the database resides on the server, while applications programs run on client workstations and access the database by communicating with the server. The number of messages between client and server may be reduced by caching a portion of the database on the client. A cache coherency protocol is used to ensure that each client's cache remains consistent with the shared database. Wilkinson and Neimat [8], and Wang and Rowe [6] address this problem, and detail several cache coherency policies. Common to them is the fact that they are integrated with the concurrency control policy of the DBMS.

In the case of a *parallel database architecture* using the shared data model, the database is accessible from each node in the system. The nodes maintain caches that store recently accessed database pages. A cache coherency policy must ensure that database pages in several caches are up to date. This means that a page updated by a transaction on a given node must either be propagated to all other nodes, or all nodes that have copies of the page in their caches must invalidate them.

In [3] six coherency policies are examined and compared. They fall into three categories. Under the *notification* of invalidated pages category, the updating node sends the identity of the updated pages to remote nodes at transaction commit time, so that they may invalidate old copies of the pages, if they are present in their caches. Under the *detection* of invalidated pages category no messages are sent, but the validity of a page is established by the accessing node at page access time. Under the *propagation* of invalidated pages category, the updated pages themselves are propagated to the remote nodes at transaction commit time. Common to these policies is the assumption that the DBMS operates under

the *force* scheme: at the transaction commit point all updates are propagated to disk and all associated locks are released.

Several cache coherency policies have been proposed that support a *no-force* or deferred write scheme. Under this scheme the propagation of updated database pages to disk is delayed beyond the transaction commit point and locks on the pages are retained. While increasing the recovery complexity, this has the advantage of reducing the number of disk writes. However, care must be taken to guarantee cache coherency in cases when an updated page which has not yet been written to disk (referred to as a dirty page) is needed by another node. Mohan and Narang [5] propose several coherency policies based on the no-force scheme and exclusive lock retention. Similarly, in [2] Dan and Yu give details of five coherency policies and compare their performance through analytical models. Two of the policies are based on the force scheme, while the remaining three retain locks on cached database pages. The policies differ in the degree of database recovery complexity that they impose; however, common to all is their integration with the concurrency control mechanism of the DBMS [4].

B. Oracle7 Parallel Cache Management

The Oracle7 Parallel Database Server [1] architecture is given in Figure 1. The system consists of a number of loosely coupled nodes on which instances of the Oracle7 Parallel Server are running, sharing a common database at the disk level. There is a fast interconnecting network joining the nodes and a Distributed Lock Manager. The distributed lock manager (DLM) of the loosely coupled system maintains the status of distributed locks, thereby coordinating access to resources such as rollback segments, dictionary entries and database pages, required by the instances of the Parallel Server. Here, we are only concerned with database pages. The locks obtained through the DLM are only used for the purpose of cache coherency management. For the purpose of concurrency control, conventional transaction locks are used. The owners of DLM locks are Oracle7 instances, while transaction locks are owned by individual transactions.

The Oracle7 instance on each node maintains a local buffer (or cache) under the Least Recently Used (LRU) buffer replacement scheme, used to cache database pages. When a transaction on a particular instance requires access to a page, the instance acquires a

DLM lock on the page on behalf of the transaction and reads the page into the cache. The DLM locks associated with each database page in the cache are retained after the transaction that requested the page has committed, provided no transactions in other nodes are waiting for the same lock. A retained DLM lock is released or downgraded, and the associated cache copy of the database page may be purged (depending on the lock conflict – see below), if a DLM lock on the same page is subsequently requested on behalf of another transaction executing on a different node. A DLM lock may be requested on behalf of a remote transaction many times during the lifetime of a local transaction. A DLM lock may, therefore, be acquired and released many times if the database page it covers is needed by transactions on other instances. For example:

- instance A becomes the owner of the DLM lock covering the data page containing row R1 and updates the row;
- instance B requests the page from instance A to update row R2;
- instance A releases the DLM lock;
- instance B becomes the owner of the page and the DLM lock and does its update of row R2;
- instance A requests the page from instance B to update row R3;
- instance B releases the page and the DLM lock;
- instance A becomes the owner of the page and DLM lock and updates row R3;
- instance A commits its transaction and still owns the PCM lock and the page until another instance requests the page.

The retained DLM lock is also released if the buffer copy of the page is pushed out of the cache. This happens when an instance must read in a new database page and the LRU buffer is full. In this situation one or more database pages from the bottom of the LRU buffer are written to disk to free the space required. This is also called a *foreground write*.

The DLM locks are of two types: shared (S) and exclusive (X). With shared DLM locks only the shared read requests can be granted to a local transaction, and the share-locked page may be present in the LRU buffers of other instances. Exclusive DLM locks ensure that no other instance has a copy of the page and update requests can be granted

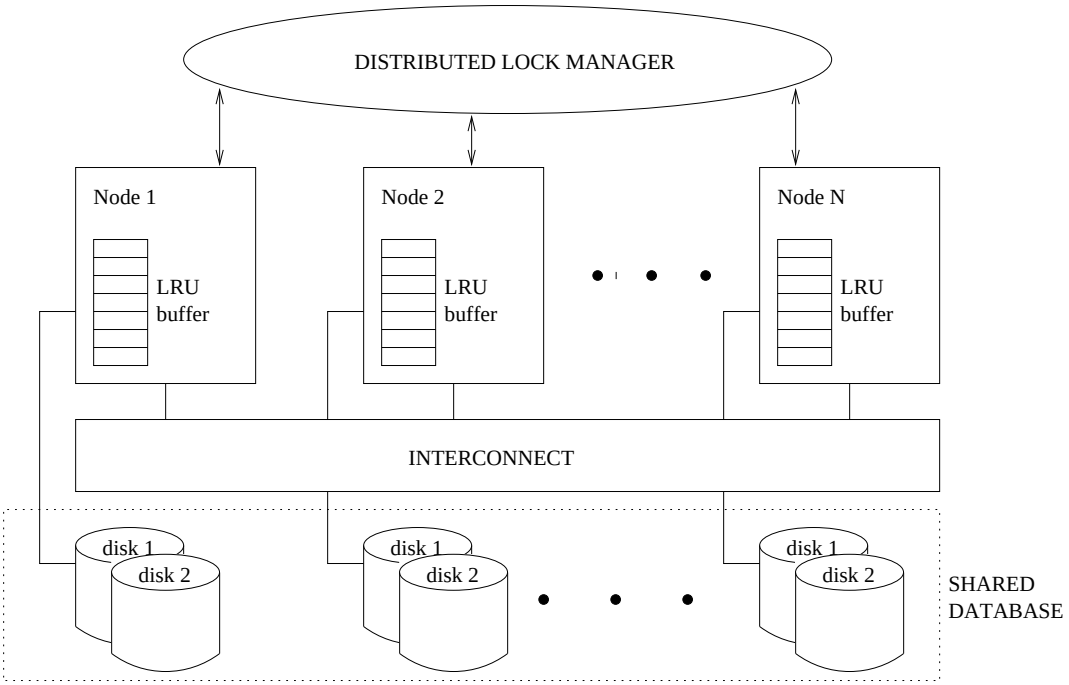

Fig. 1. The Oracle7 Parallel Database Server architecture to be modeled.

locally. At transaction commit time, redo-log entries are written to disk to guarantee that the updates are permanently reflected in the database. The updated database pages, however, are not immediately propagated to the disk (exclusive DLM locks are retained on these pages). If subsequent transactions under the same instance update the same blocks, only the final versions need to be propagated to disk, before releasing the DLM X locks, thereby saving write I/O and improving overall system throughput. A DLM X lock is either downgraded to a DLM S lock or released, only if a DLM lock on the same database block is requested by a remote node either in S or X mode. When a DLM X lock is about to be released or downgraded, the (dirty) data block is written to disk. The updated database page can then be read by the requesting node, after the propagation to disk is completed.

The cache coherency policy as used in Oracle which we want to capture in the model, is as follows. Depending on the current status of the requested page and the requested lock mode, the following cases arise:

1. An instance requests a database page (for read only or update access) on behalf of a transaction and discovers that it is not present in any of the other instances' caches.

In this case the page is read in from disk and the appropriate DLM lock is set (S or X).

2. An instance requests a database page for read only access on behalf of a transaction. The required page may be:
   (a) Held locally under DLM X lock. In this case no action is taken.
   (b) Held locally under DLM S lock. In this case no action is taken.
   (c) Held by a remote instance under DLM X lock. In this case the remote instance writes to disk the (dirty) page and downgrades its lock on this page from X to S. The requesting node's request for an S lock is granted after it reads the copy of the page from disk.
   (d) Held by one or more remote instances under DLM S lock. In this case the instance reads the page from disk and locks it in S mode.
3. An instance requires a database page for update access. Again, the required page may be:
   (a) Held locally under DLM X lock. In this case no action is taken.
   (b) Held locally under DLM S lock. In this case, if one or more remote instances hold a copy of the page under DLM S lock, their locks are revoked. Then the requesting instance upgrades its lock on the page from S to X.
   (c) Held by a remote instance under DLM X lock. In this case, the remote instance first writes the dirty database page to disk. The remote instance then releases its lock and the requesting instance acquires it.
   (d) Held by one or more remote instances under DLM S lock. In this case all the S locks on the remote instances are revoked. The instance reads the page from disk and places a DLM X lock on the page.

Writes are also propagated to disk, if an updated page is pushed out of the buffer (buffer flushing) due to the buffer filling up. When a page is pushed out of the LRU buffer, its associated DLM lock (S or X) is also released.

IV. A Parallel Cache Memory Model

This section presents an analytical model of the cache coherency policy from the previous section. Our model is derived following the approach originally developed by Dan and Yu

in [2]. The Oracle7 parallel cache management as described above is similar to the Deferred until Transfer or Flushing policy detailed in [2]. The model allows one to estimate the number of DLM X and S locks held on pages in a particular node's cache. This, in turn, allows one to obtain an estimation of local and remote buffer hit probabilities.

Suppose

$\theta^l_{m,n,d,f}$= the number of reads required on the data of node n, disk d, fragment f by a local transaction running on node m

$\theta^r_{m,i,n,d,f}$= the number of reads required on the data of node n, disk d, fragment f by a remote (to m) transaction running on node i ($i \neq n$)

$\theta^r_{m,n,d,f}$= the number of reads required on the data of node n, disk d, fragment f by remote (to m) transactions

Let

- $y_{m,n,d,f}(k)$ be the probability that the k^{th} buffer location from the top of the LRU buffer at node m contains a page of data from node n, disk d, fragment f;
- $x_{m,n,d,f}(k)$ be the probability that the page is from node n, disk d, fragment f and is holding an X lock;
- $Y_{m,n,d,f}(k)$ denote the average number of pages of data from node n, disk d, fragment f present in the top k buffer locations of node m;
- $X_{m,n,d,f}(k)$ and $S_{m,n,d,f}(k)$ denote the average number of pages from node n, disk d, fragment f holding X and S locks, respectively, in the top k locations of the LRU buffer at node m.

Then

$$Y_{m,n,d,f}(k) = \sum_{l=1}^{k} y_{m,n,d,f}(l) \quad (1)$$

$$X_{m,n,d,f}(k) = \sum_{l=1}^{k} x_{m,n,d,f}(k) \quad (2)$$

$$S_{m,n,d,f}(k) = Y_{m,n,d,f}(k) - X_{m,n,d,f}(k) \quad (3)$$

A recursive formula is used to determine $y_{m,n,d,f}(k+1)$ and $x_{m,n,d,f}(k+1)$ for $k \geq 1$

given $y_{m,n,d,f}(l)$ and $x_{m,n,d,f}(l)$ for $l = 1, ..., k$. Consider a smaller buffer consisting of the top k locations only. The buffer location $(k+1)$ receives the block that is pushed down from location k.

Suppose

$\sigma_{m,n,d,f}(k)$= the number of pages from node n, disk d, fragment f that are pushed down from location k in the LRU buffer at node m in a transaction running on node m

$\sigma^X_{m,n,d,f}(k)$= the number of pages from node n, disk d, fragment f holding X locks that are pushed down from location k in the LRU buffer at node m in a transaction running on node m

$D_{n,d,f}$= the size (in pages) of data on node n, disk d, fragment f

$\gamma_{m,n,d,f}$= The probability that a page on node n, disk d, fragment f accessed by a transaction running on node m is also updated

The next two equations are based on the following assumption. Under steady-state conditions, in the long term the number of pages that get pushed down from the top k locations of the buffer equals the number of pages that are brought into the top k locations.

Hence

$$\sigma_{m,n,d,f}(k) \;=\; \theta^l_{m,n,d,f}[1 - \frac{Y_{m,n,d,f}(k)}{D_{n,d,f}}] \;-\; (\sum_{i=1,i\neq m}^{N} \theta^r_{m,i,n,d,f}\gamma_{i,n,d,f})\frac{Y_{m,n,d,f}(k)}{D_{n,d,f}} \qquad (4)$$

The first term on the RHS of the equation denotes the number of pages required by a local transaction and not found in the buffer. Such pages are brought into the buffer from disk and placed under DLM X or S locks. The second term is the number of pages that are required by remote transactions for update and are found in instance m's buffer (under an X or S lock). Such pages are taken out of the buffer (after being written to disk).

$$\sigma^X_{m,n,d,f}(k) \;=\; \theta^l_{m,n,d,f}\gamma_{m,n,d,f}[1 - \frac{X_{m,n,d,f}(k)}{D_{n,d,f}}] \;-\; \theta^r_{m,n,d,f}\frac{X_{m,n,d,f}(k)}{D_{n,d,f}} \qquad (5)$$

The first term on the RHS of the last equation denotes the number of pages required for update by a local transaction and not found in the buffer. Such pages are brought into the buffer from disk and placed under a DLM X lock. The second term is the number of pages

that are requested by remote transactions and are found to be under a DLM X lock in instance m's buffer. Such pages must have their DLM X lock revoked: either downgraded to S or set to "null".

The next two equations follow from another assumption: the expected value of finding a page from node n, disk d, fragment f in the $(k+1)$th position of the LRU buffer of node m, $y_{m,n,d,f}(k+1)$, is approximately the same as the probability of finding a page from node n, disk d, fragment f in the $(k+1)$th position of the LRU buffer of node m in the event that a page is pushed down from location k to location $(k+1)$.

Therefore, the probability $y_{m,n,d,f}(k+1)$ can be approximated as

$$y_{m,n,d,f}(k+1) \approx \frac{\sigma_{m,n,d,f}(k)}{\sum_{i\epsilon\{1..N\},j\epsilon\{1..D\},l\epsilon\{1..F\}} \sigma_{m,i,j,l}(k)} \tag{6}$$

and

$$x_{m,n,d,f}(k+1) \approx \frac{\sigma^{X}_{m,n,d,f}(k)}{\sigma_{m,n,d,f}(k)} y_{m,n,d,f}(k+1) \tag{7}$$

Given that the next page accessed by a transaction at node m is from node n, disk d, fragment f let

- $h^{l,X}_{m,n,d,f}$ be the probability that the page is found in the local buffer of node m under a DLM X lock;
- $h^{r,X}_{m,n,d,f}$ be the probability that the page is found in one of the remote buffers under a DLM X lock;
- $h^{l,S}_{m,n,d,f}$ be the probability that the page is found in the local buffer of node m under a DLM S lock;
- $h^{r,S}_{m,n,d,f}$ be the probability that the page is found in one of the remote buffers under a DLM S lock, and it is not found locally.

The first two probabilities are easy to obtain, since a page under a DLM X lock can be in at most one instance's buffer. Thus

$$h^{l,X}_{m,n,d,f} = \frac{X_{m,n,d,f}(B)}{D_{n,d,f}} \tag{8}$$

$$h^{r,X}_{m,n,d,f} = \frac{\sum_{i=1}^{m-1} X_{i,n,d,f}(B) + \sum_{i=m+1}^{N} X_{i,n,d,f}(B)}{D_{n,d,f}} \tag{9}$$

The sets of pages with DLM S and X locks are mutually exclusive, and therefore:

$$h^{l,S}_{m,n,d,f} = \frac{S_{m,n,d,f}(B)}{D_{n,d,f}} \tag{10}$$

Also,

$$\begin{aligned} h^{r,S}_{m,n,d,f} &= [1-(h^{l,X}_{m,n,d,f}+h^{r,X}_{m,n,d,f})][1-\frac{S_{m,n,d,f}(B)}{D_{n,d,f}-\sum_{i=1}^{N} X_{i,n,d,f}(B)}] \\ &\quad [1-\prod_{j=1,j\neq m}^{N}(1-\frac{S_{j,n,d,f}(B)}{D_{n,d,f}-\sum_{i=1}^{N} X_{i,n,d,f}(B)})] \end{aligned} \tag{11}$$

Here, the first term is the probability that the page does not lie in the set of pages under an X lock. The second term is the conditional probability that the page is not present in the local buffer, given the first condition. The third term is the conditional probability that the page appears in one of the remote buffers, given the first condition (the second condition has no effect on this term since the effects are independent).

Finally, the overall buffer hit probability for a page of the data from node n, disk d, fragment f, $h_{m,n,d,f}$, can be written as:

$$h_{m,n,d,f} = h^{l,X}_{m,n,d,f} + h^{r,X}_{m,n,d,f} + h^{l,S}_{m,n,d,f} + h^{r,S}_{m,n,d,f} \tag{12}$$

V. Some Preliminary Results of the Approach

This section presents some results of a study of cache hit ratios under varying database size and varying number of participating processing elements using the Oracle7 Parallel Server cache memory model described in the previous section. In this study, the number of relation fragments assigned to each PE was varied from 20 to 90, with each fragment having 400 pages. The size of the cache memory on each PE was set as 10K pages and each page was 2K bytes. The number of participating PEs was varied among 4, 14, 24, 34, 44, 54 and 64. There are four disks attached to each PE. The database fragments were distributed across disks in a round-robin fashion. The number of page read/write operations required in a transaction on each fragment were generated randomly.

Fig. 2 illustrates the mean values of the four cache hit ratios for a PE - the mean values of $h^{l,X}_{m,n,d,f}$ (local X), $h^{r,X}_{m,n,d,f}$ (remote X), $h^{l,S}_{m,n,d,f}$ (local S) and $h^{r,S}_{m,n,d,f}$ (remote S) over all

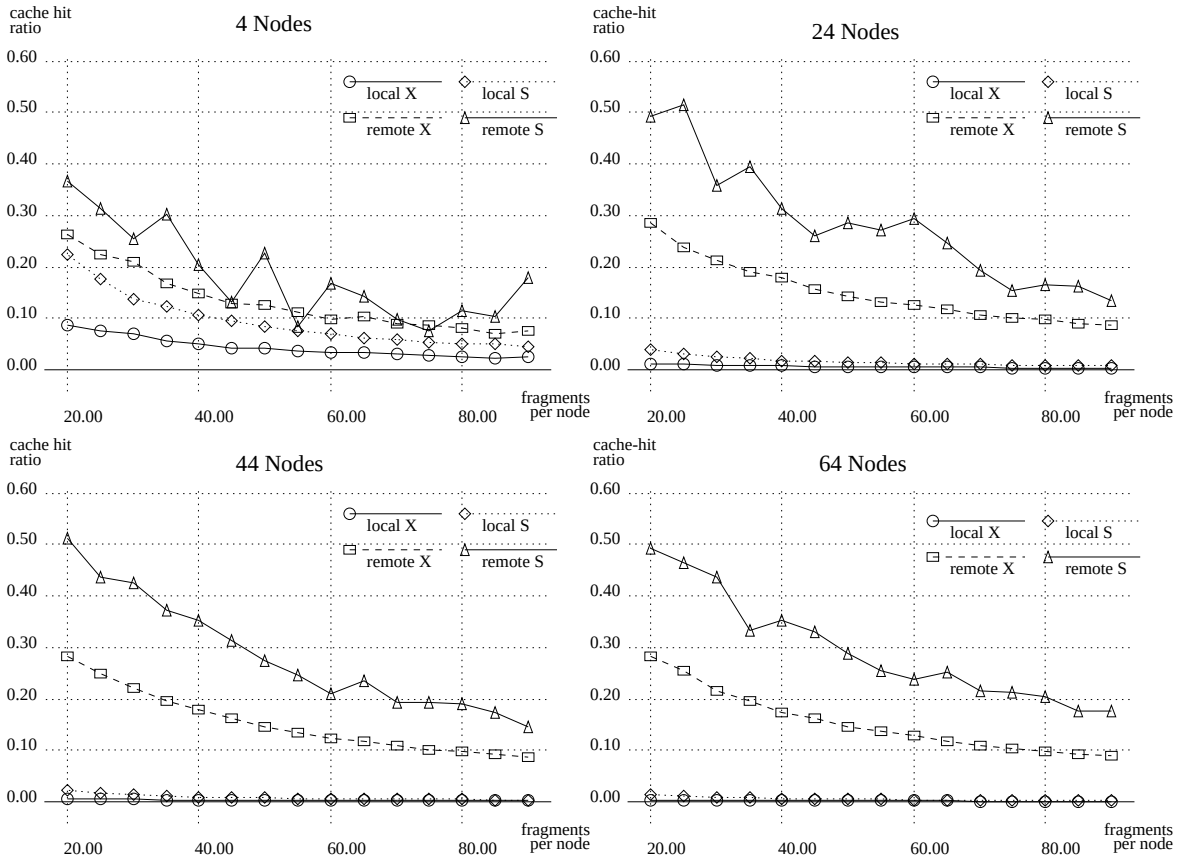

Fig. 2. The four cache hit ratios.

PEs and all fragments - estimated by the cache memory model when the number of PEs were 4, 24, 44 and 64 and the number of fragments per PE was varied from 20 to 90 in steps of 5. Fig. 3 shows the mean values of the overall cache hit ratios for a particular PE, the mean values of $h_{m,n,d,f}$ over all PEs and all fragments, when the number of PEs were varied among 4, 14, 24, 34, 44, 54 and 64 and the number of fragments per PE was varied from 20 to 90.

It can be seen from Fig. 2 that when the number of PEs is fixed and the number of fragments per PE increases, all four cache hit ratios decrease. This is due to the fact that the size of data on each PE increases while the cache size on each PE remains unchanged and therefore the probability of a page being in the cache decreases. It can also be observed that when the number of PEs increases, the probabilities that a page is found in the local buffer of a PE holding either X lock or S lock ($h^{l,X}_{m,n,d,f}$ or $h^{l,S}_{m,n,d,f}$) decreases. The reason for this behaviour is that as the number of DBMS instances increases, the probability of

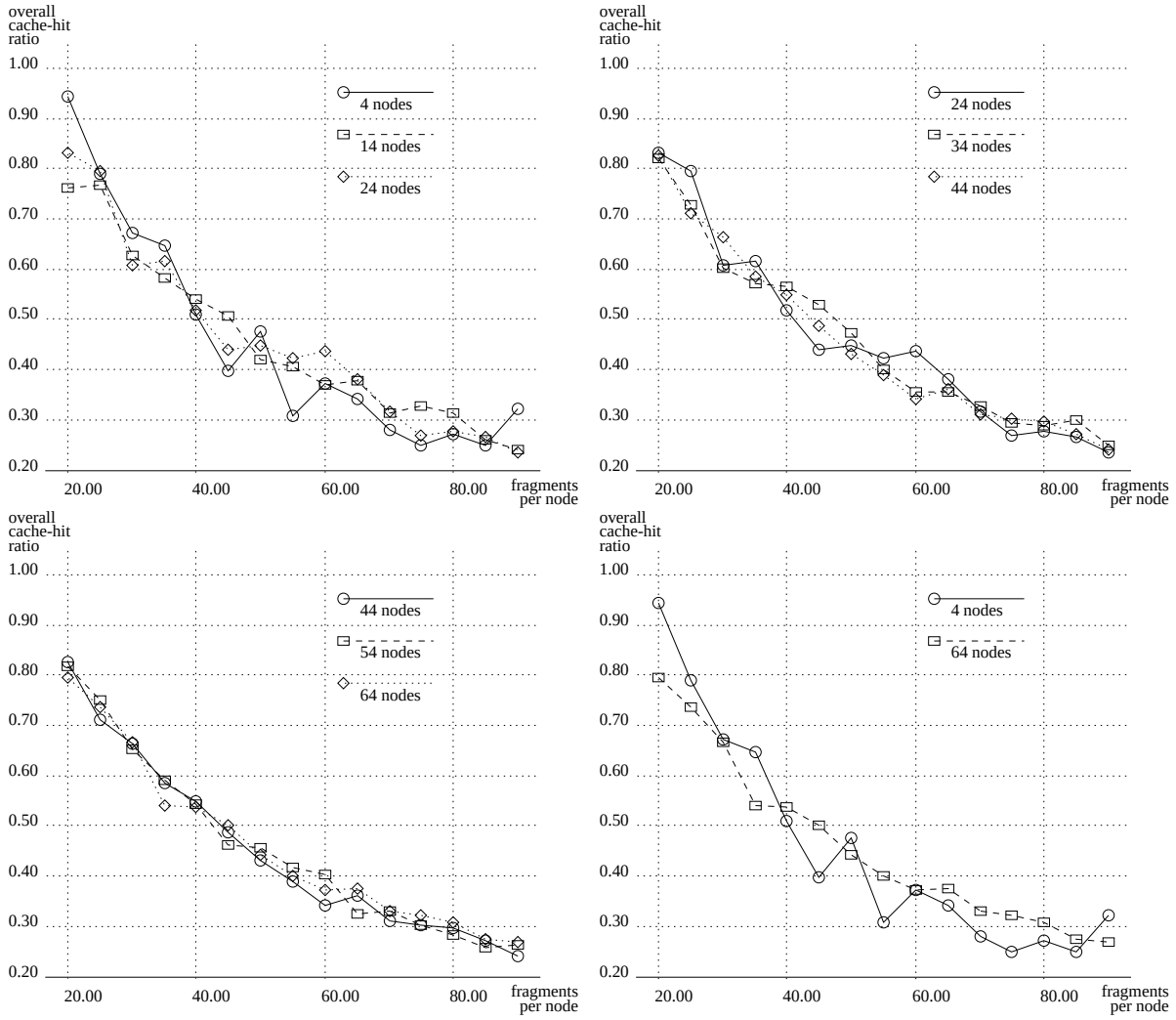

Fig. 3. Overall cache hit ratio.

a page being held by other instances also increases which means the probability of a page being held locally decreases. This is also evident in the increase of the probability that a page is found in one of the remote buffers holding an S lock, $h^{r,S}_{m,n,d,f}$, when the number of PEs increases. The probability that a page is found in one of the remote buffers holding an X lock is stable with increasing number of PEs since only one instance can hold an X lock on a page at any time no matter how many PEs are used.

In Fig. 3, the overall cache hit ratio generally decreases as the number of fragments per PE increases, although there are some irregularities in the process. The cache hit ratio curve becomes smoother as the number of PEs increases. However, the value of the overall

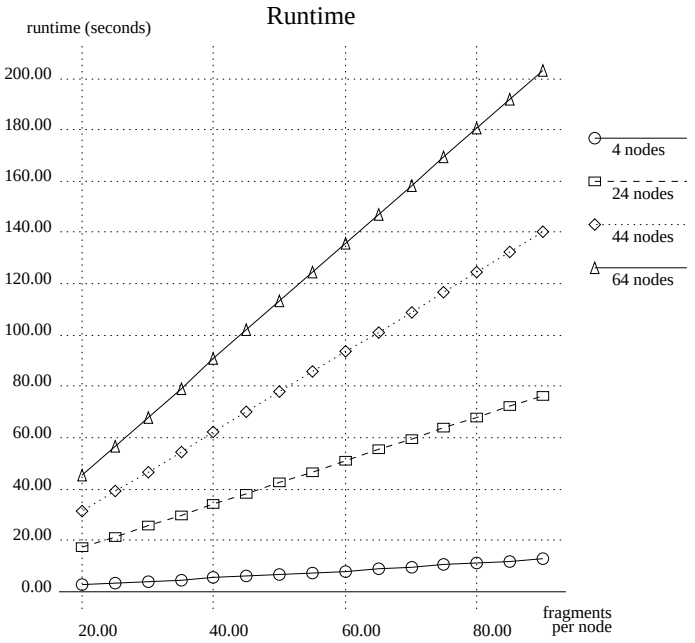

Fig. 4. The run times of the algorithm.

cache hit ratio is relatively stable as the number of PEs is varied. This is due to the fact that when the number of fragments per PE is fixed, the ratio between the overall data size (i.e. the amount of data over all PEs) and the overall cache memory size (i.e. the sum of the sizes of cache memories over all PEs) remains unchanged as the number of PEs changes.

Fig. 4 shows the run times of the cache memory model algorithm implemented in GNU C++ and executed on a Pentium 133 with 16MB RAM. The execution time increases linearly as the number of fragments increases. The larger the number of PEs, the faster the increase in the execution time. The longest time obtained in our experiment is just over 200 seconds for 60 PEs with 90 fragments per PE.

VI. Conclusions

Methods for estimating the performance of DBMSs can aid the sizing and tuning of database applications by identifying potential performance bottlenecks or by predicting the relative performance of different designs. Performance estimation is critical in parallel database systems with distributed memory where an effective overall performance depends on a good choice among a wide range of ways of placing data. To develop an analytical

performance evaluator to perform performance estimations for applications running on parallel DBMSs, a cache model for predicting cache hit ratios is critical in producing accurate estimations. This paper described a cache model used in an analytical performance estimation tool STEADY, which was developed for the cache management in Oracle7 Parallel Server. A prototype of this cache model has been developed for estimating the overall cache hit ratios for applications executed by Oracle7 Parallel Server running on the ICL GoldRush MegaServer. Some preliminary results have been obtained by using the cache model to predict cache hit ratios under varying database size and varying number of participating processing elements in a parallel DBMS.

Work is currently being extended to use the cache model for estimating the number of lock requests made by an Oracle instance through the GoldRush Distributed Lock Manager for buffer pages held by remote instances. This will be coupled with other models currently being developed within STEADY to produce an analytical performance evaluation tool capable of estimating maximum throughput values for applications running on GoldRush Oracle7 Parallel Server.

Acknowledgements

The authors acknowledge the support received from the Commission of the European Union under the Framework IV programme for the Mercury project (ESPRIT IV 20089) and from the Engineering and Physical Sciences Research Council under the PSTPA programme. They also wish to thank Mr Phil Broughton, Mr Arthur Fitzjohn, Mr John Hayley and Mr Ben Thornton of ICL for their assistance and support.

References

[1] Oracle Corporation. Oracle 7 parallel server administrator's guide, December 1992.

[2] A. Dan and P. S. Yu. Performance analysis of coherency control policies through lock retention. In *Proceedings of ACM SIGMOD Conference*, pages 114–123, California, U.S.A, June 1992.

[3] A. Dan and P. S. Yu. Performance analysis of buffer coherency policies in a multisystem data sharing environment. *IEEE Trans. on Parallel and Distributed Systems*, 4(3):289–305, March 1993.

[4] D. M. Dias, B. R. Iyer, J. T. Robinson, and P. S. Yu. Integrated concurrency-coherency controls for multisystem data sharing. *IEEE Trans. on Software Engineering*, 15(4):437–448, April 1989.

[5] C. Mohan and I. Narang. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In *Proceedings of the 17th VLDB Conference*, pages 193–207, Barcelona, Spain, September 1991.

[6] Y. Wang and L. A. Rowe. Cache consistency and concurrency control in a client/server DBMS architecture. In *Proceedings of ACM SIGMOD Conference*, pages 367–376, Denver, Colorado, U.S.A, May 1991.

[7] P. Watson and G. Catlow. The architecture of the ICL GoldRush MegaSERVER. In *Proceedings of the 13th British National Conference on Databases (BNCOD 13)*, pages 250–262, Manchester, U.K., July 1995.

[8] K. Wilkinson and M. A. Neimat. Maintaining consistency of client-cached data. In *Proceedings of the 16th VLDB Conference*, pages 122–133, Brisbane, Australia, August 1990.

[9] M. H. Williams, S. Zhou, H. Taylor, and N. Tomov. Decision support for management of parallel database systems. In *Proceedings of HPCN Europe-96*, Brussels, April 1996.

[10] S. Zhou, M. H. Williams, and H. Taylor. Practical throughput estimation for parallel databases. *To appear in the July 1996 Issue of Software Engineering Journal*, 1996.