WHITHER HUNDREDS OF PROCESSORS IN A DATABASE MACHINE?

by

Rakesh Agrawal
David J. DeWitt

Computer Sciences Technical Report #531

January 1984

Whither Hundreds of Processors in a Database Machine?

Rakesh Agrawal
Bell Laboratories, Murray Hill

David J. DeWitt
University of Wisconsin, Madison

Abstract

In the past, a number of database machine designs have been advanced that propose using hundreds of query processors to process a database query in parallel. In this paper, we argue that the performance of these designs is severely limited by the I/O bandwidth provided by the disk drives that store the database, and unless the problem of the I/O bandwidth is tackled, the use of hundreds of query processors is not justified.

1. Introduction

During the past decade, database machines have been the subject of intense research activity, and a number of database machine designs have been proposed (see the surveys in [10,18]). The basic approach in a large class of these designs has been to use the conventional mass storage devices but employ hundreds of processors to process a query in parallel in order to gain performance improvements. Examples of database machines in this class include DIRECT [8], RAP.2 [17], INFOPLEX [14], RDBM [11], and DBMAC [15]. In [9], these designs have been classified as the *multiprocessor-cache* class of database machines as all data to be processed is first moved from disk to a cache, and once the data is there, it is processed by multiple processors in parallel. Furthermore, intermediate results in processing a query are placed in the cache from where they are quickly accessible to the processors for subsequent operations in the query.

In this paper, we present the results of our simulation experiments that show that the performance of the multiprocessor-cache class of database machines is severely limited by the I/O bandwidth provided by the disk drives that store the database, and unless the problem of I/O bandwidth is tackled, use of hundreds of processors is not justified.

The organization of the rest of the paper is as follows. In Section 2, we present the basic architecture of the multiprocessor-cache class of database machines. Section 3 contains examples of query processing that illustrate the motivation behind using a large number of processors in this architecture. Section 4 describes the structure of our simulation model and Section 5 specifies the characteristics of the simulation model. Section 6 presents the results of the first set of experiments that assume that the conventional disks are the mass storage medium. Section 7 describes the results of another set of experiments where the database is assumed to reside on parallel-readout disks. Section 8 describes the sensitivity experiments. Section 9 contains our conclusions and suggestions for future research.

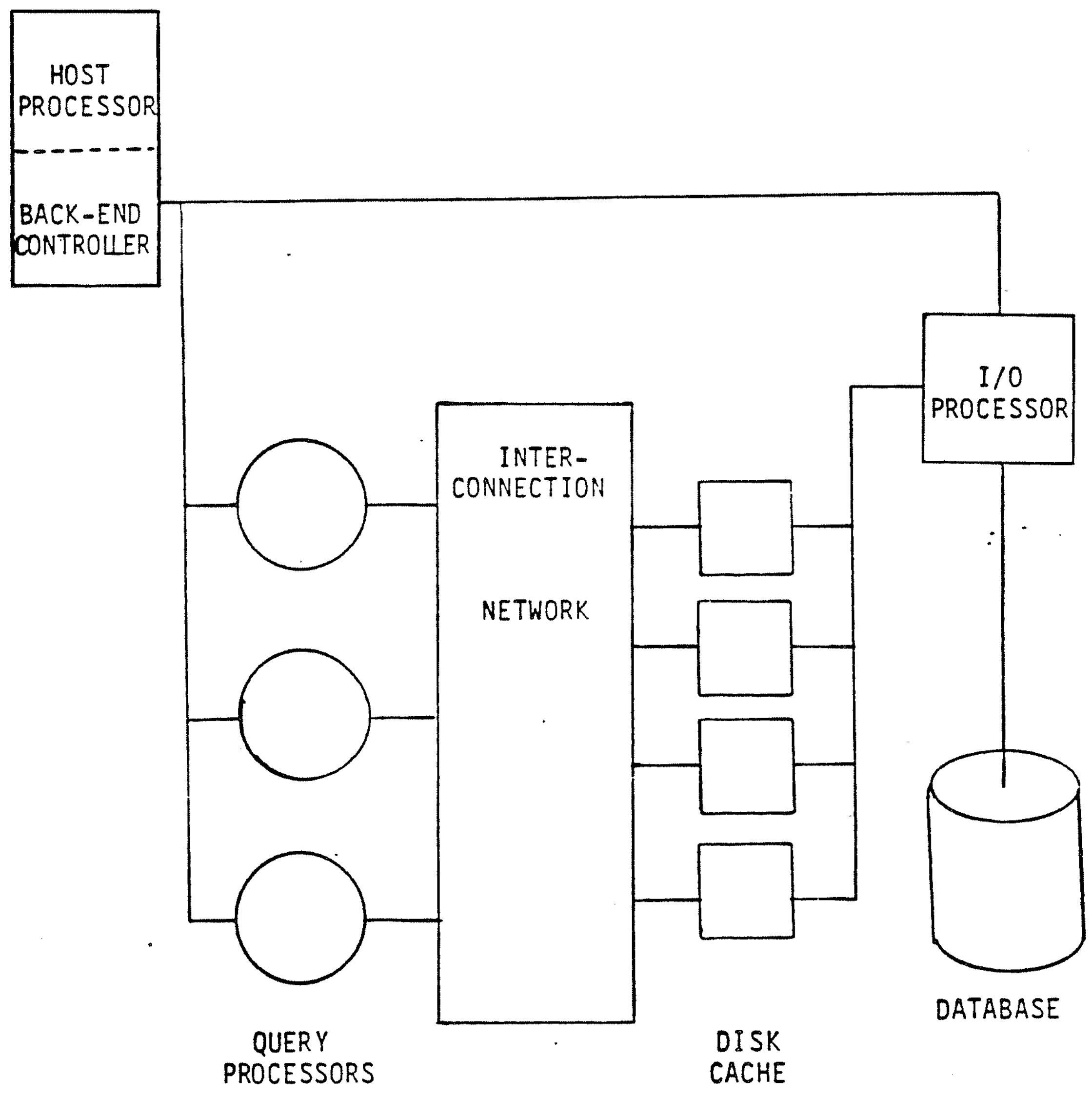

Figure 1 Multiprocessor Cache Architecture

2. Multiprocessor-Cache Database Machine Architecture

The multiprocessor-cache database machine architecture (Figure 1) consists of a set of processors, a multi-level memory hierarchy, and an interconnection device.

Some of the processors, designated query processors, execute user queries and operate asynchronously with respect to each other. One of the processors, designated the back-end controller, acts as an interface to the host processor (the processor with which a user interacts) and coordinates the activities of the other processors. After a user submits a query for execution, the host compiles the query and sends it to the back-end controller for execution on the database machine.

We assume that the memory hierarchy consists of three levels. The top level consists of the internal memories of the query processors. Each processor's local memory is assumed to be large enough to hold both a compiled query and several data pages. Mass storage devices (disks) make up the bottom level and the middle level is a disk cache that is addressable by pages.

The bottom two levels of the memory hierarchy are connected in a way that allows for data transfers between each of the disks and any page frame in the cache. A processor, designated the I/O processor, transfers pages between the disks and the cache. The top two levels of the hierarchy are so connected that each processor can read and write a different page of the cache simultaneously and all processors can simultaneously read the same page of the cache. See [4] for a discussion of the interconnection schemes that may satisfy these requirements.

3. Query Processing

In this section, we will describe the processing of some of the relational operators to motivate the suggestion for using hundreds of query processors in a multiprocessor-cache database machine. This section is in no way intended to be an exposition of the query processing algorithms in this class of database machines. See [4] for such a discussion.

Selection

To perform selection on a relation R, a different page of R is provided to every query processor. Each query processor then scans its page and extracts tuples that satisfy the search criteria. If there are N query processors and $|R|$ is the size of the relation R, then in the best case, this operation can be performed in $|R|/N$ time units, where one time unit is the time required to perform the operation on one page. In particular, if $N = |R|$ then the selection can be performed in 1 time unit, and hence the need for large N.

Join

To join an outer relation R with an inner relation S, first a different page of R is provided to each of the query processors. Then each page of S is broadcast one at a time to all of these query processors. Query processors join different pages of R with the same page of S in parallel. Thus, in the best case, the join can be performed in $(|R|/N) * |S|$ time units. If $N = |R|$, then the join will need only $|S|$ time units.

A Pitfall

The realization of the parallelism described above is predicated on the assumption that a query processor is never blocked because there are no data pages to process. We designed a set of simulation experiments to test the validity of this assumption.

4. Structure of the Simulation Model

The structure of the database machine simulator is shown in Figure 2. When a transaction arrives for processing, its size in terms of number of pages accessed and the reference string is generated. For each page accessed by the transaction, it is determined whether the page will be updated, whether the page will be found in the cache (e.g., a page of an intermediate result) or will have to be read from a disk, and how much query-processor time will be required to process this page[1].

1. These parameters are determined for each page at the time of the arrival of a transaction so that a page has identical routing and processing time across different simulation experiments.

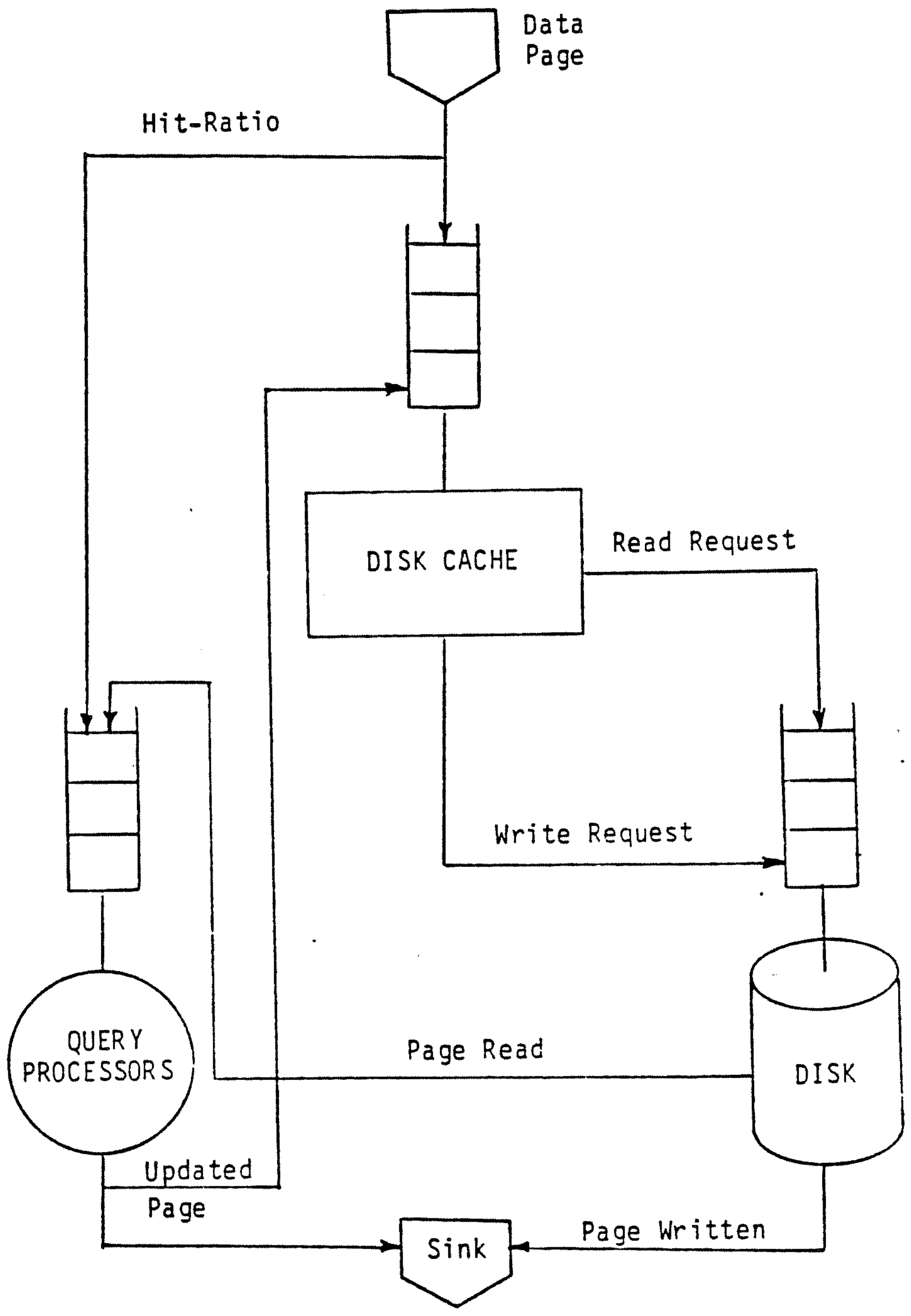

Figure 2 Data Base Machine Simulator

Pages accessed by the transaction are put in the queue for the disk cache. When a page is allocated a cache frame, it is put in the disk queue. After the page has been read from disk, the page can be assigned to a free query processor and it is put in the query-processor queue[2]. If a page is determined to be already in the cache, then it is immediately put in the query-processor queue.

When a processor becomes free, it is assigned the page which is first in the processor queue. The processor reads this page into its local memory and releases the cache frame that was holding this page. If the page is read-only, the processor becomes free after processing the page. If the page is updated, the query processor requests a cache frame to output the updated page. The cache request for sending an updated page from a processor's local memory has priority over the request for prepaging data pages from disk. When a cache frame is allocated to the updated page, the processor writes the page to the cache. The processor is now available to process the next page.

The updated page is put in the disk queue for writing it to disk. In the disk queue, updated pages have priority over the pages to be read. After all the pages of a transaction have been processed and the updated pages have been written to disk, the transaction terminates.

Observe that the query execution in our simulation model is page driven which is similar to the data-flow approach proposed in [5]. We also model anticipatory paging or prepaging [8] of data pages.

5. Model Characteristics

Transactions

A transaction is modeled by the number of pages it accesses. The size of a transaction is a uniform discrete random variable in the range 1 to 250. The size of a data page is 4096 bytes.

The reference string of a transaction can either be random or sequential. To generate a random

2. There is one queue for all the processors as all the query processors are identical and a page can be processed by any one of them.

reference string, the page number and the disk on which it resides are randomly selected for each page. The page number is between 1 and the maximum number of pages on the disk. For a sequential reference string, the first page is randomly determined. Subsequent pages are determined by incrementing the page number of the first page. All pages accessed by a sequential transaction reside on one disk which is selected in round-robin fashion. In one simulation experiment, all the transactions are assumed to be either random or sequential to isolate the effect of access pattern on performance.

In database machines, all the pages processed in response to a query are not necessarily read from disk. Some of the pages may be found in cache. For example, if a query consists of restricting a relation followed by a join of the resulting relation with another relation, then it is likely that at the time of the join operation some fraction of the pages produced by the selection operation will be present in the disk cache. The size of a transaction is the total number of pages processed by the transaction. To estimate the percentage of pages of a transaction that will be found in the cache, we instrumented the simulation of the database machine DIRECT [5], and found the average cache-hit ratio to be 20%.

The write set of a transaction is a random subset of its read set. The percentage of pages updated by a transaction was taken to be 20% of the total number of pages accessed by the transaction.

We used separate random number streams for generating different parameters to ensure that the size and the reference string of a transaction were the same across different simulations.

Query Processors

Query execution in a database machine is characterized by a mix of simple operations like restrict and complex operations like join. To estimate the average time to process a page, we again instrumented the simulation of DIRECT [5], and determined the average times for the selection and the join operations on a page. The average processing time for a page was then estimated by weighting these timings with the corresponding number of pages for each operation. Thus, 36.65 ms. was determined to be the average time to process a page. In a recent benchmark of the commercial version of INGRES running on a VAX 11/750 [3], the average cpu time per page for a selection operation was measured to be 27 ms. and the time per page for a join operation was found to be 46 ms. Thus, 36.65 ms. seems to be a reasonable estimate of the average time required to process a page. The time taken by a query processor to process a particular page is assumed to be normally distributed with the standard deviation equal to 1/3 of the average processing time.

The transfer rate between a processor's memory and the disk cache was assumed to be 1/2 megabyte per second [6].

We assumed a database machine configuration consisting of 25 query processors.

Mass Storage Devices

The mass storage devices were modeled after the IBM 3350 disk [12]. This disk has 30 recording surfaces, 555 cylinders and 4 blocks of 4096 bytes each on every track. The revolution time is 16.7 ms. and it takes 10 + 0.0772*N ms. to move the head by N cylinders.

To determine the access time for a disk page, the cylinder number on which the page resides is first computed. The access time is then computed by accounting for the seek from the current head position to this cylinder plus latency and the transfer time. Disk requests were serviced on first-come first-serve basis. We assumed 2 disk drives for our experiments.

Disk Cache

The disk cache is addressable at page boundaries of 4096 bytes pages. The unit of access is a full page. We assumed a disk cache of 100 page frames. We further assumed that the transfer rate between the cache and the processors, and the rate between the cache and the disk drives are limited only by the bandwidth of the processor's bus and the disk transfer rates respectively. Thus, the disk cache has been modeled as a *passive* resource [16]. The disk cache does not have any service time associated with it; the size of the disk cache limits the number of pages that may be active in the database machine.

6. Experiments with Conventional Disks

In the first set of simulation experiments, we assumed that the database resides on conventional disks and the access pattern of all the transactions is either random or sequential. Each simulation run was stopped after executing 500 transactions. (See [1] for a discussion on the stability of the simulation results).

Table 1 shows the average execution times per page[3] for the two types of access patterns. The *execution time per page* is defined to be the time taken by the database machine to execute a given transaction load divided by the total number of pages processed by the machine, and is a measure of the *throughput* of the machine. Total pages processed is given by $\Sigma|T_i|$ where $|T_i|$ is the size of the transaction T_i in pages.

Tables 2 and 3 contain performance statistics related to the disk drives and the query processors respectively. In Table 2, *Q-length* is the sum of the average number of disk requests pending at the two disk drives, and *Q-time* is the average waiting time before a disk request is taken up for servicing. *Access time* is the average time for transferring a page between the disk and the cache and consists of seek, latency and transfer times. *Total I/Os* is the total number of disk requests serviced by the disk drives.

In Table 3, *Q-length* is the average number of data pages resident in the disk cache waiting to be assigned to a query processor, and *Q-time* is the average time that a page stays in cache before it is assigned to a free processor. *Max QPs Used* is the maximum number of query processors in use at any time. *Max Utilization* is the maximum utilization of any one of the query processors. For computing the utilization of a query processor, in addition to the time a query processor is performing an operation on a page of data, the time during which a query processor reads a page into its local memory or writes a page to cache, the processor is also considered to be in use. *Effective QPs* is the sum of the utilizations of all the query processors.

3. unit of time throughout this paper is milliseconds

Access Pattern	Execution Time per Page
Random	18.00
Sequential	11.01

Table 1. Execution Time per Page (Conventional Disks)

Access Pattern	Utilization	Q length	Q time	Total I/Os	Access time
Random	.99	95.80	1711.22	59170	35.53
Sequential	.75	94.98	1043.40	59170	16.52

Table 2. Disk Characteristics (Conventional Disks)

Access Pattern	Max QPs Used	Max Utilization	Effective QPs	Q length	Q time
Random	9	.15	.51	0	0
Sequential	17	.16	.82	0	0

Table 3. Processor Characteristics (Conventional Disks)

It may be observed that the parallelism in the database machine is severely constrained by the I/O bandwidth. Although there were 25 processors, 16 of them were never used when the access pattern was random. Out of the 9 processors that were used, none of them was in use for more than 15% of the time and the sum of the utilization of these processors was only 51% of a single processor. When the accesses were sequential, 8 processors were never used, none of the processors was used more than 16% of the time, and the sum of the utilization of the 17 processors was only 81% of a single processor. There was never an instance when a data page was available in the disk cache and no query processor was available to process it.

In contrast to processor utilization, the disk utilization was nearly 100% for random accesses, and there were long disk queues with large waiting times. For sequential transactions, the disk utilizations were also very high[4].

4. The lower numbers for disk utilization in the case of sequential access compared to the random access is explained by the following. We assumed that a sequential transaction accesses all the pages from the same disk and the cache frames are allocated to the transactions on first-come first-serve basis. Thus, if there is a sequence of large transactions that all access, say, Disk 1 interspersed with small transactions that access Disk 2, then Disk 2 becomes idle whereas there is a queue at Disk 1.

Table 4 shows the results of another experiment. This experiment was designed to investigate the number of query processors that will be required to achieve the same performance as 25 query processors, while the other parameters (number of cache frames, number of data disks etc.) remained invariant. When the access pattern of the transactions was random, the same average execution time per page was achieved using only 3 query processors. Even when all the accesses were sequential, performance gains from using more than 6 query processors were marginal. This experiment clearly demonstrates that if the conventional disk drives are used for storing the database, then a few rather than hundreds of query processors are adequate in a multiprocessor-cache class of database machine.

Access	Number of Processors								
Pattern	25	1	2	3	4	5	6	7	8
Random	18.00	47.18	23.59	17.99					
Seq.	11.01	47.18	23.60	16.13	13.20	11.97	11.38	11.11	11.02

Table 4. Execution Time per Page (Conventional Disks)

7. Experiments with Parallel-Readout Disks

One way of increasing the I/O bandwidth is to use parallel-readout disks as proposed by the SURE [13] and DBC [2] database machine projects. On a parallel-readout disk, pages on different tracks of the same cylinder may be read and written in parallel in one disk access.

Another set of experiments were designed to investigate the effect of using parallel-readout disks on database machine performance. To simulate the parallel-readout disks, disk requests were serviced by determining the cylinder number of the page that was first in the disk queue and then examining the whole queue to find if there was another page in the queue belonging to the same cylinder. All such pages were then accessed in parallel. The rest of the simulation model and the simulation parameters were kept identical to the conventional disk experiments. The simulation results are summarized in Tables 5 through 7.

Even the parallel-readout disks did not alleviate the I/O bandwidth problem when the accesses were

Access Pattern	Execution Time per Page
Random	16.62
Sequential	1.92

Table 5. Execution Time per Page (Parallel-Readout Disks)

Access Pattern	Utilization	Q length	Q time	Total I/Os	Access time
Random	1.00	95.90	1580.80	55236	35.49
Sequential	0.92	27.42	52.65	13991	15.01

Table 6. Disk Characteristics (Parallel-Readout Disks)

Access Pattern	Max QPs Used	Max Utilization	Effective QPs	Q length	Q time
Random	10	.15	0.56	0	0
Sequential	25	.91	22.38	54.47	104.77

Table 7. Processor Characteristics (Parallel-Readout Disks)

random. 15 processors remain unused, the maximum utilization of any one of these processors was 15%, and the sum of the utilization of the 10 processors was 56%. On the other hand, disk utilization continued to be 100%. The slight reduction in the average execution time per page when compared to the conventional disk case is due to the reduction in the number of disk I/Os. A total of 55,236 disk I/Os are made with the parallel-readout disks compared to 59,170 I/Os required with the conventional disks. Table 8 shows that even with the parallel-readout disks, 3 processors were adequate to process data at the rate provided by the disk drives, and there was no further improvement in the performance if a larger number of processors were used.

Number of Processors			
25	1	2	3
16.62	47.18	23.59	16.61

Table 8. Execution Time per Page (Random Access, Parallel Disks)

The execution time, however, improves dramatically when parallel-readout disks are used for sequential accesses. The improvement is mainly the result of significantly fewer total I/Os (only

13,991 I/Os were required). Not only all the 25 query processors were used, utilization of all the processors was also very high. Consequently, there were instances when the data pages were available in the cache for processing but no processor was free. Utilization of the disk drives was also very high. Table 9 shows that the performance of the database machine can be further improved by using even a higher number of query processors and a larger disk cache in this configuration.

QPs=	25	25	75	75	75	100	250
Cache=	100	150	150	200	250	250	500
	1.92	1.92	.91	.81	.74	.70	.60

Table 9. Execution Time per Page (Sequential Access, Parallel Disks)

8. Sensitivity Experiments

We performed two sets of sensitivity experiments. First, the cache hit ratio was increased to 50% from 20% keeping the other parameters invariant (25 query processors, 100 cache frames, 2 data disks). Next, the size of a transaction was randomly determined from the range 1 to 10 instead of 1 to 250 (cache hit = 20%). The average execution time per page for these experiments is summarized in Table 10.

Configuration	Cache Hit 50%	Transaction Size 1-10
Conventional disks-Random access	17.94	12.55
Parallel disks-Random access	11.57	16.70
Conventional disks-Sequential access	6.98	11.20
Parallel disks-Sequential access	1.89	5.52

Table 10. Average Execution Time per Page

As expected, for 50% cache hit, the average execution time per page decreased when compared to 20% cache hit. However, the performance continued to be limited by the I/O bandwidth provided by the disk drives except when the access pattern was sequential and the parallel-readout disks were used. In this configuration, even with 20% cache hits, all the processors were almost fully utilized. With 50% cache hits, the processors became the bottleneck.

For small transactions (access to 1-10 pages), simulation results exhibited the same pattern as with the mix of transactions of size 1 to 250 pages. However, for sequential access pattern, the improvement in performance with the parallel-readout disks was less dramatic. With a shorter sequential reference string, fewer pages were accessed from the parallel-readout disk in one disk I/O, and consequently, there was less reduction in the total number of disk I/Os.

9. Conclusions

We have shown that the performance of the multiprocessor-cache class of database machines is severely limited by the I/O bandwidth provided by the disk drives. It has been suggested that in such machines hundreds query processors may be used to process a database query in parallel. We demonstrated that if the current state-of-art disk drives (like IBM 3350 disks) are used for storing the database, then for 2 disk drives, not even 10 query processors are adequately utilized even when the access pattern of all the transactions is assumed to be sequential. Highly parallel database machines will become viable only if the problem of I/O bandwidth is solved.

One way of increasing the I/O bandwidth is to use parallel-readout disks. For sequential transactions, when parallel-readout disks were used, the throughput of the database machine increased considerably. However, parallel-readout disks do not necessarily solve the I/O bandwidth problem. If the accesses are random or the transactions access small number of pages, so that there are not many pages belonging to the same cylinder in the disk queue, the parallel accessing capability of a parallel-readout disk becomes redundant. Similar results are expected if the disk controller is augmented with a large internal cache so that it reads the whole cylinder at a time instead of reading one block at a time. Currently, there is no empirical evidence that real life databases exhibit temporal locality.

What then is the solution to the I/O bandwidth problem? One solution may be to construct a database machine so that the whole database, or a very large part of it, is always memory resident.

A primary issue in such an architecture would be how to make updates to the database permanent. In [1], a parallel logging algorithm has been proposed to log changes to the database. As many log disks as necessary may be used to avoid degradation in the throughput due to the logging activity. This will, in addition, require efficient algorithms for incrementally saving the image of the database on stable storage while the database is in operation so that the database may be reconstructed in acceptable time after a system crash. The feasibility and the details of the architecture, the incremental dumping and the database reconstruction algorithms, the query processing strategies in such an architecture, all appear to be promising subjects for future research.

The conventional wisdom is that, for best results, if a device is free and there is a task to be performed, let the device start working on the task immediately. With parallel-readout disks, there are situations where the forced idleness may be a better choice. For example, suppose that two adjacent pages P1 and P2 are to be read from the same cylinder of a parallel-readout disk. Further, suppose that two cache frames become free at time t and $t+\Delta t$ respectively where Δt is very small compared to the disk access time, and the disk becomes free at time t. If the disk begins accessing P1 at time t, then a separate access will be required to access P2. However, if the disk is kept idle for Δt time units, then both P1 and P2 may be read in one disk access. It will be interesting to explore further the scheduling of parallel-readout disks from this point of view.

In our experiments, for servicing the disk requests, the service discipline was always assumed to be first-come first-serve. It will be worth investigating the effect of other disk-scheduling strategies on database machine performance.

10. Acknowledgements

We wish to thank Ray Bryant for developing SIMPAS [7] that saved us considerable amount of programming effort. Kevin Wilkinson and Bill Roome made helpful comments on an earlier draft of this paper. Thanks are also due to Doug Blewett and Charles Wetherell for helping us not become a prey to "mm" and "bib", and to Rudd Canaday for constant encouragement. This research was partially supported by the National Science Foundation under grant MCS82-01870.

References

1. R. Agrawal, Concurrency Control and Recovery in Multiprocessor Database Machines: Design and Performance Evaluation, Computer Sciences Tech. Rep. #510, Univ. Wisconsin, Madison, Sept. 1983. Ph.D. Thesis.

2. J. Banerjee, R. I. Baum and D. K. Hsiao, Concepts and Capabilities of a Database Computer, *ACM Trans. Database Syst. 3*, 4 (Dec. 1978), 347-384.

3. D. Bitton, D. J. DeWitt and C. Turbyfill, Can Database Machines Do Better? A Comparative Performance Evaluation, *Proc. 9th Int'l Conf. on Very Large Data Bases*, Oct. 1983.

4. D. Bitton, H. Boral, D. J. DeWitt and W. K. Wilkinson, Parallel Algorithms for the Execution of Relational Database Operations, *ACM Trans. Database Syst. 8*, 3 (Sept. 1983), 324-353.

5. H. Boral and D. J. DeWitt, Processor Allocation Strategies for Multiprocessor Database Machines, *ACM Trans. Database Syst. 6*, 2 (June 1981), 227-254.

6. H. Boral, D. J. DeWitt, D. Friedland, N. F. Jarrell and W. K. Wilkinson, Implementation of the Database Machine DIRECT, *IEEE Trans. Software Eng. SE-8*, 6 (Nov. 1982), 533-543.

7. R. M. Bryant, SIMPAS User Manual, Computer Sciences Tech. Rep. #391, Univ. Wisconsin, Madison, June 1980.

8. D. J. DeWitt, DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems, *IEEE Trans. Computers C-28*, 6 (June 1979), 395-406.

9. D. J. DeWitt and P. Hawthorn, A Performance Evaluation of Database Machine Architectures, *Proc. 7th Int'l Conf. on Very Large Data Bases*, Sept. 1981.

10. P. Hawthorn and D. J. DeWitt, Performance Analysis of Alternative Database Machine Architectures, *IEEE Trans. Software Eng. SE-8*, 1 (Jan. 1982), 61-75.

11. W. Hell, RDBM - A Relational Database Machine: Architecture and Hardware Design, *Proc. 6th Workshop on Computer Architecture for Non-Numeric Processing*, June 1981.

12. IBM, Reference Manual for IBM 3350 Direct Access Storage, GA26-1638-2, File No. S370-07, IBM General Products Division, San Jose, California, April 1977.

13. H. O. Leilich, G. Stiege and H. C. Zeidler, A Search Processor for Data Base Management Systems, *Proc. 4th Int'l Conf. on Very Large Data Bases*, Sept. 1978, 280-287.

14. S. E. Madnick, The INFOPLEX Database Computer: Concepts and Directions, *Proc. IEEE Computer Conf.*, Feb. 1979.

15. M. Missikoff, An Overview of the Project DBMAC for a relational machine, *Proc. 6th Workshop on Computer Architecture for Non-Numeric Processing*, June 1981.

16. C. H. Sauer and K. M. Chandy, Computing Systems Performance Modeling, Prentice-Hall, Englewood Cliffs, N.J., 1981.

17. S. A. Schuster, H. B. Nguyen, E. A. Ozkarahan and K. C. Smith, RAP.2 - An Associative Processor for Data Bases and its Applications, *IEEE Trans. Computers C-28*, 6 (June 1979), .

18. S. W. Song, A Survey and Taxonomy of Database Machines, *IEEE Database Engineering Bulletin 4*, 2 (Dec. 1981), 3-13.