An Analysis of Three Transaction Processing Architectures

Anupam Bhide
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720.

Abstract

In this paper, we investigate the issues involved in using multiprocessors for high performance transaction processing applications. We use a simulation model to compare the performance of three different architectures, namely, Shared Everything, Shared Nothing and Shared Disks. In Shared Everything, any processor can access any disk and all memory is shared. In Shared Nothing, neither disks nor memory is shared. In Shared Disks, any processor can access any disk, but each has its own private main memory. We first study four different variations of the Shared Disks architecture which attempt to minimize lock request messages. We will then compare the best Shared Disks variation with Shared Nothing and Shared Everything. In addition, we study how intra-query parallelism affects the performance of the architectures.

1. Introduction

Applying multiple processors to database problems has been an active area of research. In the database machine area, several research prototypes as well as a few commercial products have been built. However, most of these systems have attempted to accelerate long running queries such as joins. Less attention has been directed by researchers to efficient transaction processing on multiprocessors. The design of multiprocessors for high speed transaction processing is the main focus of our research. There has been a lot of debate in the transaction processing industry about the suitabaility of various architectures for transaction processing. This debate has focussed on issues such as reliability and performance. We attempt to provide answers about the performance of these architectures by comparing them with uniform assumptions.

In this paper, we will concentrate on three issues:

(1) The main aim of this paper is to compare the performance of three different architectures, namely, Shared Everything(SE), Shared Nothing(SN) and Shared Disks(SD) with uniform assumptions. In a SE system, all disks are directly accessible from all processors with the same access times and all memory is shared. Examples of this architecture include the Sequent Symmetry system[SEQU 86], the Firefly, the IBM 3090 series of machines and SPUR[HILL 86]. In an SN architecture, each disk is connected to a single processor and each processor has its own private memory. A collection of SUNs on an Ethernet exemplifies this architecture. Additionally, a TANDEM TXP system is essentially an SN architecture[BART 81] with extra hardware for high availability. In a SD architecture any processor can access any disk but each processor has its own private memory. IBM provides the "multi-system data sharing facility" [STRI 82] as a feature of its IMS database system which is the ability to couple different systems running IMS in a SD configuration.

(2) In a brute force implementation of a SD architecture almost every lock request needs two messages. Since messages are expensive, the SD architecture is unattractive unless this can be avoided. We will study different variations of SD which try to reduce the number of messages and compare their performance. We will then compare the best SD architecture with SN and SE.

(3) We also study the effect of intra-query parallelism for SE, SN and SD. In a SN system parallelism in a query is structural in nature. Each processor must run the portion of the query plan relevant to the database on its disks. Therefore,

This research was sponsored by the Defense Advanced Research Projects Agency under contract N00039-84-C-0089, the Army Research Office under contract DAAL03-87-G-0041, and the National Science Foundation under contract MIP-8715235.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

the degree of parallelism in a query plan is determined by the distribution of data. The optimal distribution of data is a very hard problem since one needs to balance the load on all the processors and yet not cause too many messages. However, in a SE system, a query can be split into n pieces, for any n > 1 and any processor can run each of the n pieces. There is no cost for switching jobs between processors and the only cost paid for parallelism is the cost of starting up a new process. However, the extensibility of the SE architecture is limited by the bandwidth offered by the latest bus and memory technology. The SD architecture offers an intermediate range of flexibility between SE and SN. Unlike SN, any portion of the query plan can be executed by any processor. However, once a portion of the query plan is assigned to a processor, reassigning it for load balancing reasons involves sending messages. Therefore, we will not consider reassignment for SD. Also, like SN, transmission of query pieces to processors involves sending messages. One of the most interesting design decisions in each of the architectures is the degree of intra-query parallelism one should aim for. For SE and SD, this decision can be made at query optimization time. For SN, the decision must be made at data distribution time.

In [BHID 87] and [BHID 88] we have studied SE and SN under various workload conditions and compared their performance. In [BHID 88], we studied how SN and SE performed as lock contention was varied. In [BHID 87], [BHID 88] we studied different types of workloads and reported that results for both the above issues were substantially different for two different classes of workloads:

(a) either high lock contention or strict response constraints present

(b) low lock contention and no response constraints
The main observation we made was that high lock contention or strict response constraints magnify both the performance gap between SE and SN and the performance gains due to parallelism. In addition, we compared the effect of load imbalances on both architectures and studied alternative file organizations. We showed that a sequential file system creates disk hot spots when intra-query parallelism is present and a parallel file system is necessary. In this paper, we will first compare the performance of different variations of the SD architecture and then compare the best one with SN and SE.

We have built discrete event simulators for all the architectures. In section 2, we will discuss the different variations of the SD architecture. The simulators and the input workloads that we have used are discussed in the section 3. In section 4, we then present a variety of simulation results.

2. The SD Architectures

In this section, we will describe the different variations of the SD architecture which attempt to minimize the cost of lock request messages. The SD architecture bears a distinct resemblance to a fully replicated database system as far as the design of a locking protocol. Locking schemes for SD architectures may be classified based on two criteria: (1) centralized or distributed (2) synchronous or asynchronous.

Another issue for SD architectures is how to keep buffers consistent. Each processor in an SD system has its own buffer pool. Consistency must be maintained between the different buffer pools, and between the buffer pools and the database version on disk. One easy solution is to purge the updates of a transaction from the buffer pool after commit: the transaction is committed when the log is written out, but locks are not released until all the updated pages are written out on disk. Thus the database always contains the latest copy. Pages are always read from disk and there are no buffer hits. We call this the *buffer purge* method. Another alternative would be to design a *buffer invalidation* protocol which marks as invalid all buffer pages on a processor updated by another processor. The problem is that a buffer invalidation protocol can result in a large number of messages, unless it is well-designed.

One method to reduce messages would be to route transactions to processors such that locality of reference is maximized. However, a study of IMS traces [REUT 84] has shown that there is a portion of data (10-15%) which is referenced in 80% of all update lock requests. Updates to such high-traffic data would result in notifications to almost all the processors. Thus it is not clear whether an invalidation protocol would do better than a purge protocol and any such invalidation protocol must be carefully designed.

Buffer invalidation protocols are:

(1) hard to optimize
(2) sensitive to hot spots

Therefore, for the purpose of this paper, we have chosen to use the simpler buffer purge protocol only. We will study the design of buffer invalidation protocols and the tradeoffs between them and buffer purge in a future paper. In this paper, we have considered a database size which is fairly large: 8 Gigabytes (2,000,000 pages of 4Kbytes each). Even with a 80-20 access pattern and 160 Mbytes (40,000 buffers of 4 Kbytes each, 4000 on each processor) of total buffer space the best buffer hit ratio we can hope for is of the order of 0.08. Therefore, the

effect of no buffer hits in the SD architecture variations should have small performance effects.

We will now describe the four SD variations that we intend to study.

(1) Disk Controller Locking (**DIS**): In this distributed scheme, each disk controller maintains a lock table for all the data on the disks connected to it. An example is the Limited Lock Facility for the 3830 IBM disk controller. Locking is performed with the Lock and Proceed channel command word. If the lock is available, the I/O program can continue to read or write the disk. If the lock is not available, the I/O program terminates early. When the lock is released, an interrupt is generated so that the I/O program can be issued again. Other channel command words are supported to release a lock and read the lock table. A spare control unit is needed to recover from a locking unit failure. In this architecture no messages are needed for lock requests. The only extra price is that of communicating with the disk controller for a lock even in the case of a buffer hit. However, specialized hardware in the form of sophisticated disk controllers is needed.

(2) Central Lock Manager (**CLM**): In this centralized scheme, one processor is designated to be the lock manager. Other processors send all lock request messages to this processor. In our simulations, we have used a 10 processor system for all the architectures. In CLM, we designate one of these as the lock manager and use the other 9 as transaction processors. This will ensure a fair comparison with the other architectures. Since sending messages is expensive in terms of CPU instructions, we will attempt to optimize on the number of messages sent by batching many lock requests into one message.

(3) Primary Copy Method: (**PRI**): In this distributed scheme, the database is divided into partitions and each processor is given authority over one partition. This method gets its name from the analogy with the method of the same name proposed for replicated databases. A lock request can be handled locally if it lies in the processor's partition. Otherwise, a message must be sent to the processor which controls the partition. The choice of database partitions and assignment of primary authority need not be static and could be made by global load balancing software on the basis of current load and reference pattern. However, our simulation model does not consider changes in primary copy authority in order to maintain simplicity. In this scheme, one can save on lock messages by routing transactions to processors such that most of the lock requests are local. In section 4.1, we will study how lock request locality affects performance.

(4) Asynchronous Primary Copy Method: (**APRI**): In the PRI scheme, one cannot batch messages because the time interval between a processor sending two messages to another processor is too large and thus there is a degradation in response time. In PRI, lock requests are synchronous. However, if the lock requests are asynchronous, batching would be possible. In the APRI scheme, a processor sends a message containing a lock request and continues processing assuming that it will get the lock. If a negative response is received, then the transaction must be aborted. At commit time, the transaction waits until it gets all responses to its lock requests. It commits successfully if it gets all its locks, otherwise, it is aborted. This is similar to optimistic concurrency control, except that the serializability check is not postponed till commit time; as soon as any negative response is received to a lock request the transaction is aborted.

2.1. Previous Work

[YU 85a], [YU 85b] have studied the performance of SD architectures. [YU 85b] concentrates on a distributed pass-the-buck locking protocol that is used by IMS to synchronize locking in a SD environment. They have shown that this protocol does not work beyond 8 processors, because of increase in contention due to increase in buck cycle time. Our distributed protocols, PRI and APRI are limited only by the bus bandwidth and have no such inherent limitation. [YU 85a] studies a model similar to our CLM, but there is no study of message batching to improve performance. [REUT 84] proposes a range of SD architectures, but there is no performance study. The main contribution of this paper is to compare SE, SN and SD architectures with uniform assumptions. We are not aware of any paper in existing literature which does that.

3. The Simulation Model

In turn, we discuss the architecture, the file system, the workload, the concurrency control algorithm and the buffer cache and logging aspects of our simulation model. In the last sub-section, we will describe the queueing model we used.

3.1. The Machine Architectures

To make a fair comparison between SD, SE and SN, we have used the same system parameters in both environments whenever this assumption is realistic. The parameters are summarized in Table 1. The default value shown in the table has been used in the simulation runs, however, for particular runs, we have used different

values from those in Table 1. These will be noted wherever they occur.

The choice of parameter values has been influenced by the fact that the ultimate goal of our project is to study the viability of using the SPUR architecture (a SE type architecture) being developed here at Berkeley[HILL 86] as a transaction processor. We assume that *page_size* is 4 Kbytes. *Cpu_speed* has been set to 4 MIPs for every processor. We use a uniform distribution for *disk_random-access-time* with a mean of 20 ms. Log pages are written onto the log disks in a sequential manner. The parameters for sequential log writes are *disk_rot_time*, the rotation time for a disk, *disk_page-access-time*, the time required to read/write a page.

Table 1: System Parameters

Machine Parameters	
Parameters	Default Value
page_size	4 Kbytes
cpu_speed	4 MIPs
disk_random_access_time	20 ms mean, uniform distribution
disk_rot_time	16 ms mean, uniform distribution
disk_page_access_time	1.333 ms
network_speed	10 Mb/sec / 30 Mbyte/sec
inst_per_message	5000 inst/message (mean)
number_of_cpus	10
number_of_disks	80
number_of_buffers	40000
System Parameters	
Parameters	Default Value
file_system_type	random/sensibly split
batch_size	1
lock_request_locality	0.1
par_degree	1
Workload Parameters	
Parameters	Default Value
access_pattern	random, 80-20 rule
no_of_pages	10 (mean)
write_access_ratio	0.5
no_of_terminals	4000
think_time	20 sec mean, exp distribution
Dbase_Size	2,000,000 granules
inst_per_page	12,500 (mean)
Concurrency Control Parameters	
Parameters	Default Value
timeout_interval	10 sec
delay	0 sec
mult_degree	400

For the SN system, we have used 10 Mb/sec as the speed of the local area network. For the SD system, we have used 30 Mbyte/sec as the speed of the network which connects all the disks and all the processors. The higher speed is necessary, since all the pages accessed by transactions must be read across the network. We assume that messages are exponentially distributed with a mean size of 1000 bits. Hence, the time each message requires for transmission on the network will be distributed exponentially with mean 1000/*network_speed*. In our simulations, we have observed network utilizations of only 5 - 30% with these parameters. Therefore, the network related parameters are not critical in determining system performance. The number of CPU instructions required to process a message is assumed to be exponentially distributed with mean *inst_per_message*, and this parameter has been set to 5000 instructions/message. Our simulation results are sensitive to this parameter and it is one of the main factors which determines the difference in performance between the SN/SD architectures and SE. We will study the sensitivity to message cost of SD results in section 4.3. We have already studied this for SN and SE in [BHID 88]. We have used a 10 processor system for each of the architectures as the basic configuration in which to run our experiments because SPUR[HILL 86] currently runs out of memory bandwidth with 10 four MIP processors. By limiting the size of the SE system, we avoid having to explicitly model memory and bus contention. For the CLM variation of the SD architecture, we designate one processor out of 10 as the lock manager and use the other 9 to process transactions. Also, we have used a total of 80 database disks. In both the SE and SD architectures these are all attached to the bus and can be accessed from any processor. In SN, 8 are attached to each of the 10 processors. In addition to the database disks, we use 4 to 6 dedicated disks for the SE and SD logs, and one disk per processor for the SN log. The number of SE/SD log disks is chosen based on workload parameters, so that the log disks will not become a bottleneck.

3.2. The Software System Model

For both the SD and SE architecture we used a random file model, in which consecutive disk blocks of a file are distributed randomly over all the disks. This choice is motivated by our desire to explore intra-query parallelism. We wish to maximize the probability that multiple sub-plans can be processed in parallel without bottlenecking on accessing a single disk drive. For the SN architecture we have a "sensibly split" file model where a file is split between the disks of some number of processors (this number would depend on how many files a query accesses, the pattern of access etc.) such that the degree of parallelism achieved for a query is n. It should be noted that achieving a parallelism of exactly

n depends on knowing a lot about the application and the set of pages accessed by a query. The portion of a file on one single processor has disk blocks allocated randomly on all disks attached to it. Note, that if the random file model were to be used for SN a single file would be distributed over all the nodes. Thus a single query would very likely be executed on all nodes and would result in poor performance. For CLM and APRI, the default message batch size used is 1. For PRI and APRI, the default lock request locality used is 0.1. This value represents the case where lock requests are randomly scattered over the entire database. *Par_degree* is the degree of parallelism which is to be used in executing each query and indicates the number of parallel sub-queries into which each query is decomposed. We have varied *Par_degree* in section 4.4, elsewhere we use a value of 1.

3.3. The Workload Model

We have assumed that transactions access pages according to Zipf's Law with 80% of the accesses going to 20% of the database. The 20% frequently accessed pages are distributed uniformly over all the disks and also over all the processors for SN. Table 1 also describes the other workload parameters. *No_of_pages* is the total no of pages that each transaction accesses which we assume to have an exponential distribution with mean 10. This makes our transactions about the same size as the TP1 benchmark[GRAY 84] which does 5 to 10 I/Os. The number of instructions spent in processing a page is exponentially distributed with a mean of *inst_per_page*, which is set to 12,500. *Write/access ratio* is the ratio of number of pages updated to total number of pages accessed. Consequently, each page is updated with this probability. *No_of_terminals* is the number of terminals which are attached to the system from which users enter queries. *Think_Time* is the time a user thinks before submitting a new query after he gets a reply to his previous query. This is assumed to have an exponential distribution with mean 20 sec. *Dbase_Size* is the size of the database in terms of the number of lockable objects. We have set *Dbase_Size* to 2,000,000 granules. This leads to a low lock contention probability. Elsewhere ([BHID 88], [BHID 87]), we have studied workloads with high lock contention. Our results indicate that in high lock contention environments, performance is similar to environments where response constraints are present.

If response time constraints are specified for a particular workload, then the *No_of_terminals* parameter is varied until the maximum number of terminals that can be supported with the given response time constraints is found. Thus getting one data point for a given workload with response time constraints can involve a number of simulation runs.

3.4. The Concurrency Control Model

We use dynamic 2-phase locking as the concurrency control method. We chose to implement timeouts instead of deadlock detection since global deadlock detection is difficult to model in SN. The timeout mechanism seemed to work very well and the number of restarts was negligible. If timeout occurs, all locks are released and the transaction is restarted after a time equal to *delay*. In APRI we use asynchronous locking. This means that transactions do not wait for lock requests to complete and continue processing. If the lock is busy, the transaction must be aborted.

Mult_degree is the number of active transactions in the system at any time.

3.5. The Buffer Cache and Logging Model

For SD, we have used the *buffer purge* scheme described in section 2. For SN and SE, we have implemented a database cache scheme[ELHA 84]. We have assumed a database cache large enough to contain all the updated pages of a transaction. This cache is used to hold all pages that are (1)currently "pinned" for reading by an active transaction (2)"dirty" pages that have been updated by an active transaction and (3) frequently accessed pages which are not in categories (1) and (2). When a transaction commits, its updates are not written to the database. Instead, the buffers are marked as "updated" and the actual database update is deferred until the buffer replacement policy needs to reclaim an "updated" buffer. A list of available buffers (those not in categories (1) and (2) above) is maintained as a FIFO queue. Every time a buffer is accessed it goes to the tail of the FIFO queue. At commit time updated pages are written to the log. This use of page level logging corresponds to the tactic used by some commercial systems, and we plan to experiment with record level logging in the future. In our model of SE/SD we have to multiplex the log among 4 to 6 dedicated disks to prevent the log from becoming a bottleneck. In SN, we keep one log disk per processor. In case of a timeout, the transaction is aborted. In this scheme, aborting a transaction is simple; all its updates are discarded. We have not modelled the recovery from system or media failure. The default cache size is 40000 buffers.

3.6. The Simulation Queueing Model

In this section, we describe the queueing simulation model for both the SE, SN and SD architectures. Figure 1 shows the queueing model for the SE architecture. A transaction waits at the Ready Queue until the number of active transactions in the system falls below the degree of multiprogramming that has been chosen. Then it enters the system and is split into a number of sub-queries equal to the degree of parallelism. Each sub-query is modelled as a list of pages to be processed.

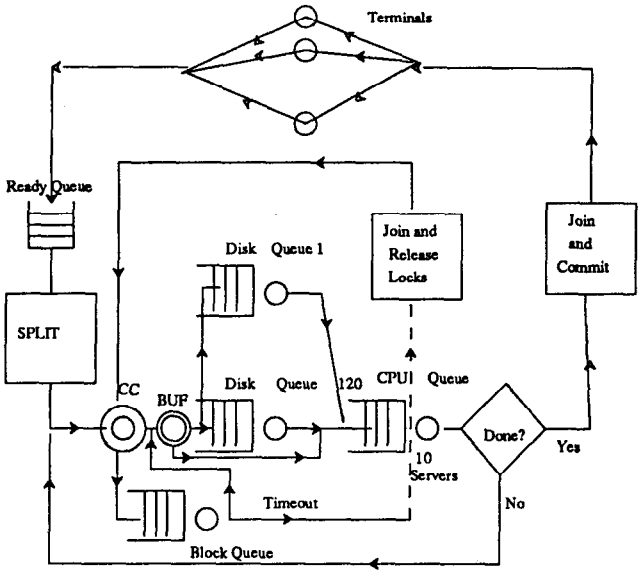


Fig. 1

Shared Everything Queueing Model

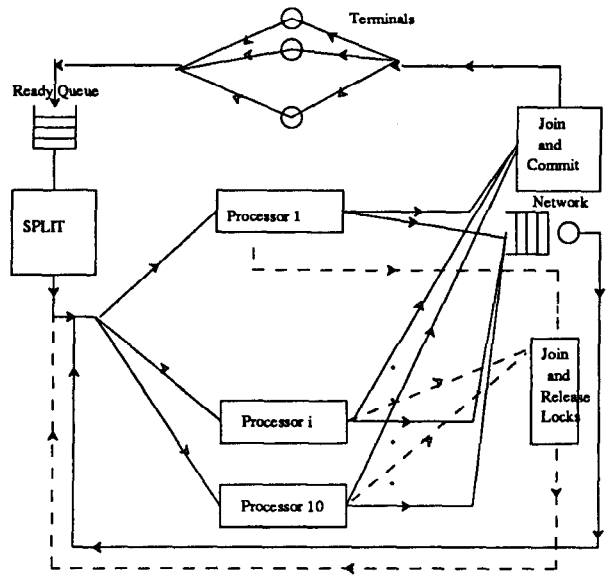


Fig. 2 a
Shared Nothing Queueing Model

The number of pages assigned to each sub-query is exponentially distributed with a mean equal to the average number of pages read by the transaction divided by the degree of parallelism. Each sub-query is an independent job in the queueing network and cycles between the disk queue, using the disk, the CPU queue, using the CPU and the blocked queues until all its pages have been processed. There is a single queue for all CPUs since the scheduler assigns a job to the next available CPU. CC is the node where a sub-query generates concurrency control requests. The blocked queue holds sub-queries which are waiting for locks. For each page in its read or write set, a sub-query first goes to CC node to get a lock and then to the BUF node to get a buffer. If the particular page is already in the cache, the sub-query goes directly to the CPU queue. Otherwise, it goes to the disk queue for a particular disk, then it goes to the CPU queue to process that page. After all its pages have been processed a sub-query job goes to the Join node and waits until all its sibling sub-queries arrive. Then all locks are released, the log record containing all the updated pages is written, and the transaction job goes back to the terminal queue. If a sub-query times out while waiting for a lock, it goes to the *Join and Release Locks* node where it waits for its sibling sub-queries. Each sibling is located and forced to this node. Then, they release all locks and the transaction is restarted after a time *delay*. However, the read and write lists of the transaction are preserved so that the same access pattern is repeated on restart.

Figure 2a shows the queueing model for the SN architecture and a detailed view is shown for one of the 10 identical processors in Figure 2b. The Ready Queue and the SPLIT node have the same functions as in SE. In this case, however, there are 10 separate processors which communicate over a network. Each transaction has a home node which is the node at which it entered the system and the splitting of a transaction into sub-queries takes place at the home node. For each sub-query, a message is sent to its execution site to start up the sub-query. This corresponds to a model of the file-system where files are split such that each query is executed on n processors, where n is the degree of parallelism for that simulation run. Each sub-query joins the CPU queue at the Net Out node and consumes the CPU time needed to send a message. On arriving at its destination, the sub-query joins the CPU queue at the Net In node and consumes the CPU time spent in processing the arriving message. For each page, the order of access is CC, BUF, disk and CPU as in the previous architecture. When all its pages have been processed, a sub-query moves to the Join and Commit node, where it waits for all its siblings to complete. The log record for the query is then written and all locks are released. There are a total of 3 rounds of messages exchanged: (1) a message is sent from the home node to each execution site containing the subquery to be executed (2) a message is sent from each execution site to the home node indicating that subquery processing is done and it is ready to commit (3) after the home node gets a "done" messages from all sites a "commit" message is sent to each of them. Thus, if degree of parallelism is n, the number of messages

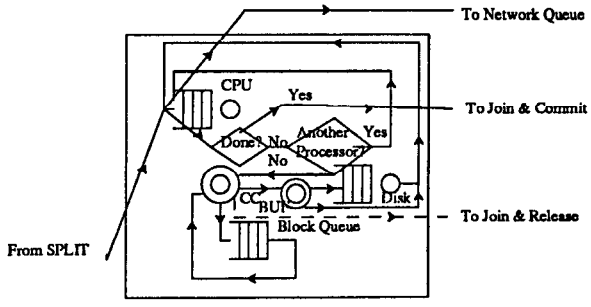


Fig. 2b
A Shared Nothing Processor

exchanged is 3*n. This protocol is appropriate when only one query is performed at each site. If multiple queries were simulated, we would need a general 2-phase commit protocol with additional messages. In case a timeout occurs at any of the execution sites while waiting for locks, an "abort" message is sent to the home node, which in turn sends an "abort" message to each of the other execution sites. Jobs which timeout go to the Join and Release Locks node, where they wait for their siblings, and once all their siblings have been forced to this node, their locks are released and they are restarted after time *delay*.

The SD simulation model resembles the SN model except for lock requests and the fact that disks are shared. In CLM, a job wanting to make a lock request waits at a *batch* node until *batch_size* other jobs collect. Then these jobs are bundled up and the CPU message cost for sending the message is paid. The message goes out over the network and is received by the lock manager processor. The cost of receiving the message is paid and each lock request is processed. The lock manager processor maintains 9 separate *batch* nodes, one for each transaction processor. After waiting at the right *batch* node for *batch_size* other responses, they are packed into a message, the CPU cost of sending is paid and the message goes to the network queue. The lock response is received by the requesting processor after paying the CPU cost of receiving a message. After a transaction commits, a lock release message is sent to the lock manager. PRI and APRI have similar simulation models except that the lock requests go to appropriate transaction processors, instead of a lock manager processor. In APRI, another difference is that a job spins off a lock request and continues processing. If the response received is negative, the transaction gets aborted. Each job waits for all its lock responses to come back after it has finished all its processing and just before it gets to the *Join and Commit* node. After a transaction commits a lock release message is sent to the all processors from which locks were requested. These release messages are batched along with the lock request messages. In DIS, the lock requests go to the disk controller. In all the SD architectures, we route transactions to the processor with the least number of active transactions. This helps in balancing processor loads.

The simulator is written is C and is about 10,000 lines of code. Each data point is obtained by a simulation run lasting for 1,000,000 events. A transition from one queue to another is counted as an event. Results were observed to very stable at this point. Each run lasts 90 minutes on a Sequent Balance machine and simulates 60 - 75 minutes of system time.

4. Simulation Results

Our main emphasis is on transaction throughput in the three architectures. As mentioned in section 2, the two main techniques we will study for improving SD performance are batching of messages and increasing lock request locality. In section 4.1 we will study the effect of lock request locality on PRI and APRI. In section 4.2 we will show the effects of batching lock request messages for CLM and APRI. In section 4.3 we will compare the four SD variations, namely, DIS, CLM, PRI and APRI for different message costs. We will choose the best SD architecture from section 4.3 and compare it with SN and SE for different degrees of intra-query parallelism in section 4.4. The simulations in sections 4.1, 4.2 and 4.3 were performed without response time constraints. In section 4.4, we will use response time constraints to compare SE, SN and SD.

4.1. Effect of Locality

In a SD architecture, a transaction can be executed on any processor unlike SN. An optimal processor allocation policy for transactions must reconcile two sometimes conflicting objectives: (1)In both PRI and APRI, efforts can be made to route transactions to that processor which holds the lock authority for most of the granules accessed by the transaction (assuming that accesses can be predicted with some degree of accuracy). This saves two messages for each local lock request. (2)The second objective is to try to balance the load on all the processors.

In this experiment, we will try to estimate the payoff from trying to maximize lock request locality. This will help a designer in making the correct trade-off between balancing processor loads and maximizing lock locality. Also, if a designer knows roughly how much locality can be obtained from his workload, he can judge whether APRI/PRI is better than CLM which has no concept of locality. Changing the access pattern enables us to change the locality ratio.

Figures 3 and 4 show the throughput and response time respectively against the locality ratio (fraction of

Figure 3
PRI: Effect of Primary Copy Locality

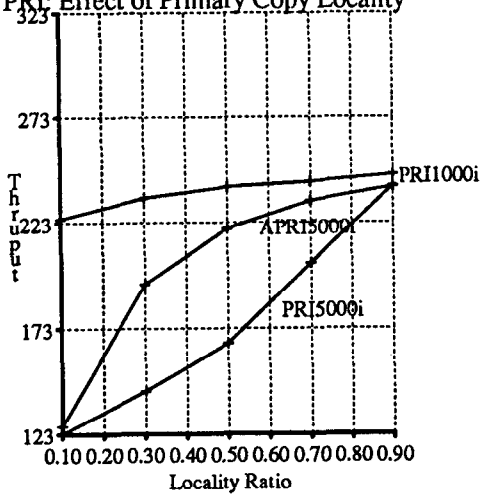


Figure 4
PRI: Effect of Primary Copy Locality

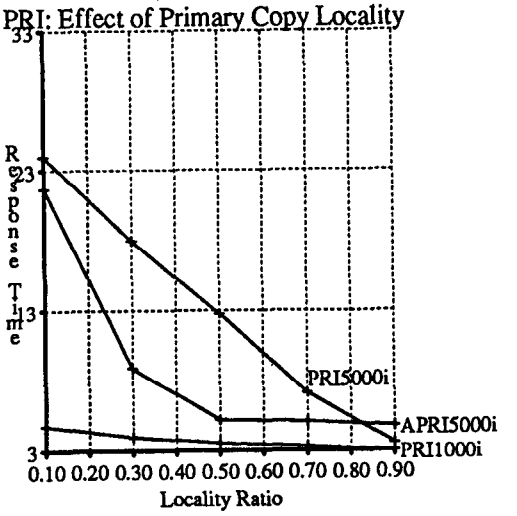


Figure 5
CLM: Effect of Message Batching

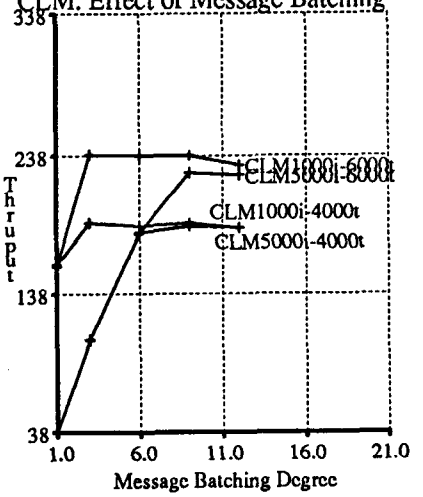


Figure 6
CLM: Effect of Message Batching

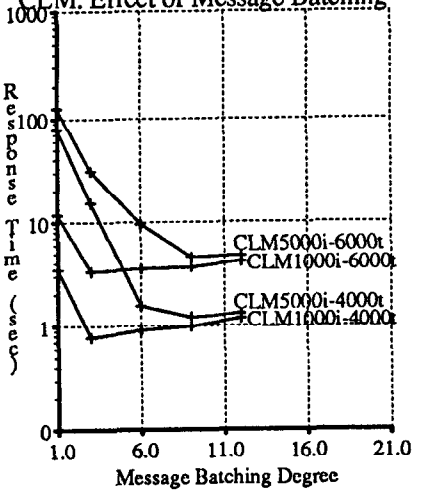


Figure 7
APRI: Effect of Message Batching

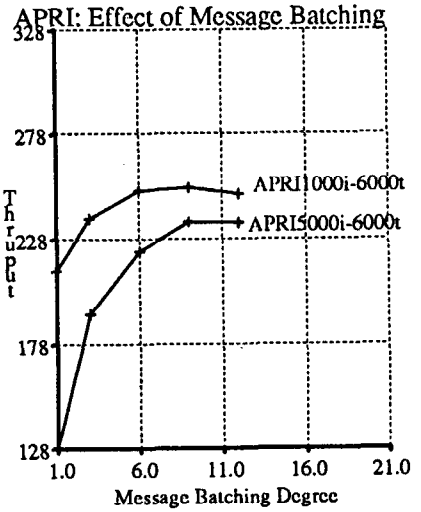
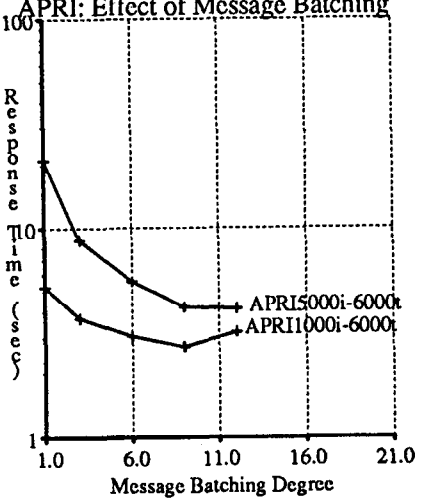


Figure 8
APRI: Effect of Message Batching



local lock requests) for PRI and APRI. The number of terminals is held constant at 6000. PRI1000i stands for the PRI architecture with a message cost of 1000 instructions and PRI5000i stands for PRI with message cost of 5000; similarly for APRI5000i. The gain for the PRI1000i curve is small because 1000 is too small a message cost to cause a substantial overhead. However, the gain for the PRI5000i curve is substantial(almost 100%) as the locality is increased from 0.1 to 0.9. Notice, however, that the PRI5000i curve is concave upwards, which means that most of the gain is between localities 0.5 and 0.9. It might be hard to achieve such high localities. Figure 4 shows that the response time for PRI5000i improves with better locality. The gains for APRI are approximately the same as for PRI. However, the APRI curve is convex upwards which means that most of the gains are in the 0.1 to 0.5 locality region. This degree of locality is easier to achieve and thus APRI shows better locality gains than PRI.

4.2. Effect of Batching

In this experiment, we will study the effects of batching lock messages, for CLM and APRI. Message batching would be a loser for PRI since the interval between lock requests from a processor A to a processor B is of the order of 50 ms. Since lock requests are synchronous, and each transaction makes 10 requests, for a degree of batching of 4 this will add about 750 ms to the response time. This kind of overhead would make it impossible to keep the subsecond response time constraints that are necessary for most transaction systems. In APRI the lock requests are asynchronous and the batching overheads are lower. In CLM, all lock requests go to the central lock manager; therefore, the period between requests is only 5 ms.

Figures 5 and 6 show throughput and response time as a function of degree of batching for four variations of CLM. In the curve labels, 1000i stands for a CLM with message cost of 1000 instructions and 5000i stands for a message cost of 5000 instructions. 6000t and 4000t stand for 6000 and 4000 terminals respectively. As the degree of batching increases all the four curves in Figure 5 show large throughput gains and then flatten out. After batching degree 9 the curves fall slightly showing that the advantage of reduced message costs are overcome by the disadvantage of waiting for more lock requests. At higher batching degrees, the message costs become insignificant, therefore, the curves for 1000i and 5000i merge together. For lower number of terminals, the throughput is smaller but the response time is better.

Figures 7 and 8 show throughput and response time as a function of degree of batching for two variations of APRI. Comparing figures 5 and 7, APRI performs about the same as CLM at higher batching degrees, but performs much better than CLM at lower batching degrees.

Batching has two effects: (1) reducing the message cost for lock requests by a factor equal to the batching degree. (2) increasing the response time due to waiting for a batch to form. We performed an experiment to isolate and observe the effect of (2). In figure 9 CLM1 stands for CLM with degree of batching 1. CLM9 stands for CLM with degree of batching 9. However, the CLM1 curve represents an architecture where the message costs are 1/9th of those of CLM9. For example, the CLM1 point at X coordinate = 2000 was taken by using a message cost of 2000/9 = 222 instructions. Thus, both CLM1 and CLM9 have the same message costs; however, CLM9 has to pay the penalty for batch waiting times. Thus CLM9 has a throughput about 20 transactions/sec below CLM1 and a response time which is about 2 seconds higher.

4.3. Effect of Message Cost

In this section, we compare the effect of CPU message cost on the different SD architectures. Figures 10, 11 and 12 show the throughput, response time and the CPU utilization for various SD variations against the CPU cost of sending a message. This is the number of instructions it takes a processor to send a message. We assume that it takes the same number of instructions for the receiving processor to receive a message. For DIS, however, the "CPU cost of messages" represents the extra number of CPU instructions needed to communicate lock requests to the disk controller. This is the only extra overhead in DIS, since, all the architectures need to communicate with the disk controller to read/write pages. This number is typically small. That is why the DIS curve is plotted only between 500 and 2500. CLM9 stands for the CLM variation with a batch size = 9 and CLM1 stands for batch size = 1. Similarly for APRI1 and 9. In Figure 12, the CPU utilizations of the central lock manager and those of the other processors are different; for example, CLM1-clm curve shows the lock manager utilization of the CLM1 architecture and similarly the CLM1-all curve shows the utilization of the other processors. Figure 10 shows that CLM1 is very sensitive to message cost. Figure 12 explains why: since the lock manager processor spends most of its time processing messages, the CLM1-clm curve rises very fast and is very near saturation. Thus the central lock manager becomes the bottleneck in this system. This shows that it is worthwhile to keep the CPU utilization of the central lock processor low, perhaps by using a faster processor. For CLM9, the CLM9-all curve rises very fast also, but is farther away from saturation, so CLM9 shows very good performance. If the workload is heterogenous, with few infrequently running transactions which make many lock requests and most transactions which make few, the presence of the lock intensive transactions will degrade the rsponse time of the short

Figure 9

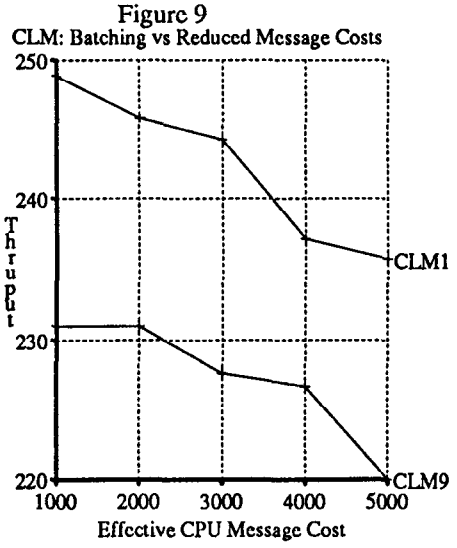


Figure 10

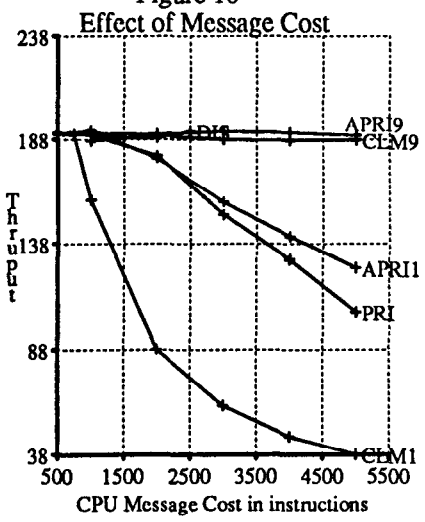


Figure 11

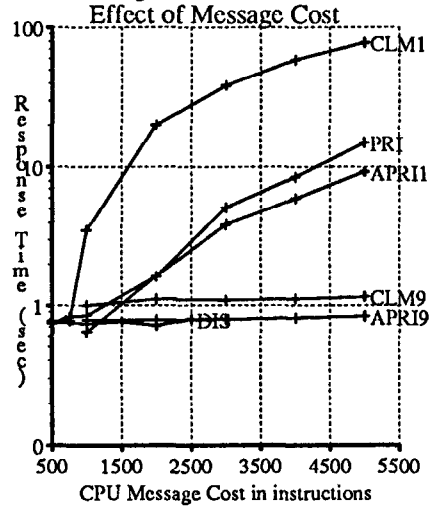


Figure 12

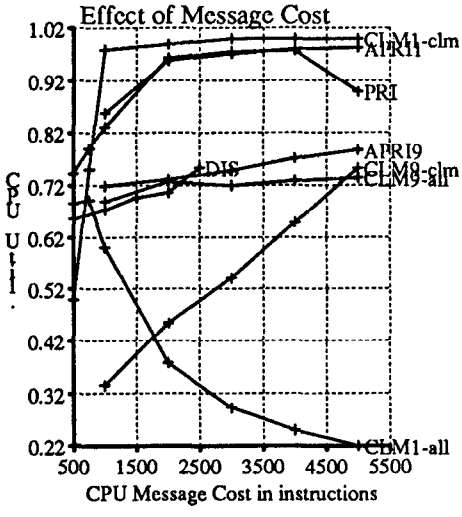


Figure 13

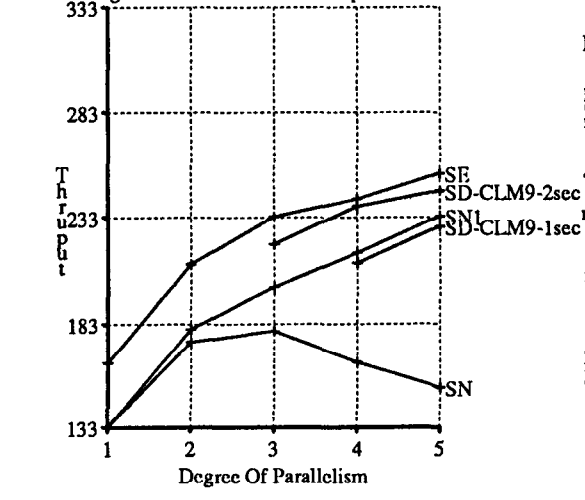
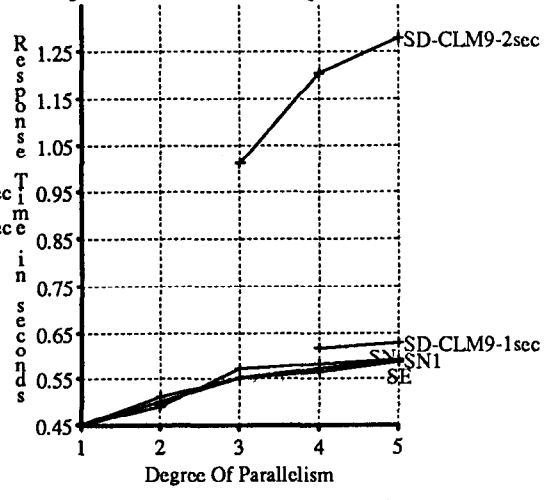


Figure 14



transactions. CLM thus is an unstable design because of the presence of a potential hot spot.

The best performers are DIS, APRI9 and CLM9. The good performance of DIS is intuitively expected. However, DIS requires extra hardware in the form of a disk controller which is able to keep track of locks. Therefore, it is reassuring to know that both CLM9 and APRI9 can match the performance of DIS very closely. APRI9 is better than CLM9 in terms of response time and almost the same in terms of throughput. PRI does worse than both APRI9 and CLM9 but better than CLM1. As explained before, the lock messages in PRI cannot be batched because the degradation in response time would be too high. Also notice that APRI1 does better than PRI specially at higher message costs. The gap between APRI1 and PRI is the gain of asynchronous locking over synchronous locking at low lock contentions (The probability of conflict on making a lock request is approximately 0.0007 with these parameters). For higher lock contention workloads, APRI should perform worse; we have not investigated this effect yet.

4.4. Comparison of SE, SN and SD

In this section, we compare the effect of parallelism on SD, SE and SN. From the previous section (figures 10 and 11), we see that DIS, CLM9 and APRI9 have approximately similar performance. Both DIS and APRI9 have the same throughputs as CLM9 but the response times are about 20% better. Since DIS needs special hardware it would be unfair to use it to compare SD against SN and SE. The APRI architecture has an additional locality parameter which is hard to estimate. Therefore we chose the CLM9 variation (central lock manager with message cost of 5000 and batching degree 9) for this experiment. We asssume the same message cost for SN: 5000 instructions. The default parameters from Table 1 were used for this set of simulations. Figures 13 and 14 show throughput and response time against the degree of parallelism for the different architectures in a response time constrained environment. The response time constraint is that 90 % of the transactions must have a response time of less than 1 second. This is a fairly tight constraint since the average total stand-alone execution time of a single transaction is 0.233 seconds. For SD, we have also plotted another curve with the response constraint changed to 90% transactions respond within 2 seconds. This curve is labelled as SD-CLM9-2sec and the one with the 1 second constraint is labelled as SD-CLM9-1sec. SN and SE curves are plotted with a 1 second constraint only and are labelled SN, SN1 and SE. We have plotted two variations of SN. For the curve labelled SN, sub-queries belonging to the same transaction were executed on different processors and hence had to pay a message cost for 3*n messages if n is the degree of parallelism. For the curve labelled SN1, all the subqueries belonging to a transaction were executed on one processor and thus no message cost had to be paid. To obtain each data point in figures 13 and 14, a number of simulation runs were required. Each simulation run was taken with a given number of terminals and depending on whether the constraint is achieved or not the number of terminals is increased or decreased for the next run. A binary search type technique was used to find the maximum number of terminals that can be supported with the given constraint.

At low degrees of parallelism, SD-CLM9 has response times of the order of 2 seconds. Therefore, almost no transactions can meet the 1 and 2 second deadlines. The points not shown for the SD curves should be taken to be almost 0. Thus intra-query parallelism is necessary for good SD performance when tight response constraints are present. Note that the SD curves lie in between the curves for SN and SE. In fact if the response constraint is relaxed to 2 seconds, SD (the SD-CLM9-2sec curve) performs almost as well as SE for high degrees of parallelism.

In Figure 13, the SN curve goes up initially as parallelism helps to increase the number of transactions making the response time constraint. After parallelism degree 3, however, the cost of messages starts to drive the curve down. SN1 does much better than SN at higher degrees of parallelism because no message cost has to be paid. However, in a real SN architecture, it would be hard to achieve SN1, since it involves localizing the data of every transaction on one processor. Therefore the performance of SN would lie somewhere between SN and SN1. Note that SE beats both SN and SD. In Figure 16, SE and SN are about 30 transactions/sec apart at a degree of parallelism of 1 and this gap increases at higher degrees of parallelism. Both SE and SD (with 1 second response time) achieve their best performances of 253 transactions/sec and 228 transactions/sec at a degree of parallelism of 5, while SN achieves its best performance of 179 transactions/sec at a parallelism degree of 3. Thus, there is a 41 % performance gap between SE and SN and a 22% gap between SN and SD. SN1 and SE are about 30 transactions/sec apart at all the degrees of parallelism.

From Figure 13 it is clear that parallelism improves performance substantially for all three architectures. For SE, the performance jumps 55 % between parallelism degrees 1 and 5. For SN, the jump is smaller, about 35 % between degrees 1 and 3 and then throughput falls between degrees 3 and 5. Parallelism reduces the response time and hence more transactions satisfy the response time constraint of 1 second. It also helps to balance the loads on the CPUs and the disks, and to achieve higher utilizations.

Figure 14 shows the average transaction response time. Response times increase as the degree of parallelism is increased, because, at higher degrees of

parallelism more terminals are supported and throughput increases. The fact that the throughput increases shows that even though average response time increases, the variation in response time decreases (more transactions make the 1 second deadline) for higher degrees of intra-query parallelism.

5. Conclusions

We have described a study of the performance of three multiprocessor architectures, namely, Shared Nothing(SN), Shared Everything(SE) and Shared Disks(SD), in transaction processing. We first studied four different variations of SD which attempted to optimize on the number of lock request messages sent. We studied the performance effects of two mechanisms for doing this: batching for CLM and APRI and increasing lock request locality for PRI and APRI. We observed that DIS, APRI with batch size 9 and CLM with batch size 9 had roughly comparable performance and were better than the other variations. We used CLM with batch size 9 to compare SD against SN and SE and to study the effect of the degree of intra-transaction parallelism. We observed that SD performs as well as the optimistic SN version (which assumes that each transaction is local to a single processor) and much better than a reasonably pessimistic version. Thus SD is a viable architecture in cases where it is not possible to partition the database so that the data for most transactions is local to one processor. However, we also observed that CLM with batch size 9 needs high degrees of parallelism to meet response time constraints.

We observed that SE outperformed SN and SD by a fairly wide margin. However, SE has two disadvantages:

(1) It is limited by bus and memory bandwidths.
(2) It has a potential single point of failure: the shared memory.

Therefore, SE is the architecture of choice for environments where the transaction throughput requirements are within the bus/memory technological barrier and failures can be handled in other ways. One way to do this would be to connect up SE nodes in a SN configuration. SD would be the architecture of choice if throughput requirements were too large to use SE. Designing a SD transaction processing system is easier than designing a SN system because the database partitioning problem does not arise. SD, of course, has its own technological limitations, namely bus bandwidth. However, this bus is less loaded as compared to an SE bus, since, the SE bus must handle memory traffic in addition to disk traffic. Thus, the choice of an architecture for a transaction processing system depends on throughput requirements, availability requirements, workload considerations and hardware technologies available.

References.

[BART 81] Bartlett, J.F., "A Non-Stop Kernel," Proc. 8th Symposium on Operating Systems Principles, 1981.

[BHID 87] Bhide, A., and Stonebraker, M., "Performance Issues in High Performance Transaction Processing Architectures" Proc. 2nd International Workshop on High Performance Transaction Processing, Oct 1987.

[BHID 88] Bhide, A., and Stonebraker, M., "A Performance Comparison of Two Architectures for Transaction Processing" Proc. 4th Intl. Conference on Data Engineering, Feb 1988.

[ELHA 84] Elhard, K. and Bayer, R., "A Database Cache for High Performance and Fast Restart in Database Systems," ACM Trans. on Database Systems, Dec 1984.

[GRAY 84] Gray, J., et. al. " A Measure of Transaction Processing Power," Datamation 1984.

[HILL 86] Hill, M., et. al., "Design Decisions in SPUR," IEEE Computer, Nov. 1986.

[REUT 84] Reuter, A. and Shoens, K., "Synchronization in a Data Sharing Environment," unpublished manuscript.

[SEQU 86] Sequent Computers, "Balance Guide to Parallel Programming," June. 1986.

[STRI 82] Strickland, J., et. al., "IMS/VS: An Evolving System", IBM Systems Journal 21, 4(1982).

[YU 85a] Yu, P., et. al., "Modelling of Centralized Concurrency Control in a Multi-System Environment," Proceedings of the 1985 SIGMETRICS Conference on Measurement and Modelling of Computer Systems, Aug 1985.

[YU 85b] Yu, P., et. al., "Distributed Concurrency Control Analysis for Data Sharing," Proceedings of the Intl. Conference on Management and Performance Evaluation of Computer Systems, 1985.