Design and Analysis of Integrated Concurrency-Coherency Controls

Daniel M. Dias, Balakrishna R. Iyer, John T. Robinson, and Philip S. Yu

IBM Thomas J. Watson Research Center
P. O. Box 218, Yorktown Heights, NY 10598

Abstract

In a multi-system data sharing complex, the systems have direct access to all data, with sharing typically at the disk level. This necessitates global concurrency control and coherency control of local buffers in each system. We propose an integrated controller for handling both global concurrency and coherency control, and show that this leads to a significant performance gain. The multi-system performance can be enhanced by use of an intermediate shared semiconductor memory. This gives rise to additional read-write synchronization and disk write serialization problems. We show these can be handled efficiently by the integrated controller, while allowing for early transaction commit. Significant transaction speedup and reduction in lock contention among transactions are obtained. The decrease in lock contention allows the multiple systems to sustain a higher transaction throughput. A queueing model is used to quantify the performance improvement. Although intermediate memory can be employed as a buffering device our analysis shows that substantial performance gains can be realized when combined with the integrated concurrency-coherency control.

1. Introduction

Two general approaches have been used in designing multi-system transaction processing complexes, which we will refer to here as *data sharing* and *partitioning* [SEKI84, YU86]. Using a data sharing approach, all systems have direct access to all data by means of an I/O network, bus, or switch, whereas using a partitioning approach, database function requests are sent via an inter-system network to the system "owning" the data. Although a detailed comparison of these two approaches is beyond the scope of this paper, it is clear that in the case of a central-site transaction processing complex data sharing has several strong advantages: (1) since all systems have access to all data, the complex as a whole can be made more available; (2) commit protocols are less complex; (3) load balancing problems can be more easily solved; and (4) migration from a single system is much simpler, since it is unnecessary to construct a mapping of parts of the global database to the systems in the complex. In fact, with regard to the last two points, a "real-world" example is described in [CORN86] in which it is impossible to partition more than three ways and satisfy load balancing constraints without restructuring databases and applications.

On the other hand, using data sharing two problems arise that do not occur when using partitioning: *global concurrency control* and *coherency control*. First consider concurrency control. In a partitioned complex, since each file, relation, etc., is owned by a unique system, all locking can be done locally within each system (if global deadlocks are possible, the only remaining concurrency control problem is their detection, for which a variety of techniques are known). Using data sharing, though, any data block can potentially be accessed by any system, and so it is necessary for systems to acquire global locks. Next, the coherency control problem (this term is used here in analogy with the problem of cache coherence in multiprocessor systems) is the following. One or more levels of the memory hierarchy will be private to each system (in particular, the primary memory of each system will be private). If copies of data blocks are maintained in private memory buffers past the end of transactions that access them (using LRU replacement, for example), it is necessary to invalidate these private copies if they are modified by another system before being replaced (a copy of a data block is said to be *valid* if it is the most up-to-date copy, that is, a copy of the block written by the most recently committed transaction that modified the block). Since in a partitioned complex all memory is private to some system, this problem does not occur.

In the past, these two problems have been addressed using separate mechanisms. For example, in IMS multi-system data sharing a distributed algorithm is used to implement global locking, and invalidation messages are broadcast in order to implement coherency control [STRI82] (in other data sharing complexes, the need for coherency control is avoided simply by purging the database buffers used by each transaction when the transaction completes). Here, we investigate the integration of these two mechanisms within a single global *concurrency-coherency controller*. The key observations that lead to studying integrated controllers are the following: (1) since blocks are the unit of data transfer between private and shared memories, block granularity locking is the most natural choice (coarser granularity may not provide sufficient concurrency, and finer granularity will not prevent one system from overwriting changes made within a block by another system); (2) if private memory buffers are maintained past the end of transactions that access them, block granularity buffers are also the most natural choice; (3) therefore, it should be possible to reduce the overhead associated with global control by integrating the data structures and algorithms for finding and changing all global control information associated with data blocks.

Two basic types of data sharing complexes will be considered. In the first type, examples of which exist today, there are two levels in the memory hierarchy: (1) private primary memories for each system, and (2) shared secondary memory (disk storage including log disks). For this type of system we investigate the addition of a controller that not only provides concurrency control functions but also coherency control functions: the controller will maintain information on the valid copies of data blocks in all private memory buffers. The overall structure of such a system is shown in Figure 1.1.

The second type of system we consider is motivated by the rapidly decreasing cost of semiconductor memory. Due to this continuing decrease, it has become cost-effective to add an intermediate level of semiconductor memory to the traditional primary-secondary memory hierarchy (examples are cached-disk devices [SMIT85] and the expanded storage of the IBM 3090 system). Here, we assume that the intermediate level of memory is shared by all systems in the complex, as shown in Figure 1.2. Also, we will make two more assumptions regarding the use of intermediate memory. First, we assume that the management of the memory is partitioned among the systems, that is, each system is responsible for allocating, writing, and freeing blocks in disjoint regions of intermediate memory (however, each system can read from any intermediate memory partition). Second, it is assumed that even though the intermediate memory may be volatile, each transaction can commit as soon as all blocks modified by the transaction have been written to intermediate memory (but only *after* log records have been written to non-volatile memory in the case that the writes were to volatile memory). Given these two assumptions, there are some additional problems of global control as compared to the previous two-level memory hierarchy system. For example, one system should not be allowed to free and then overwrite a block in intermediate memory while it is being read by another system. Similarly, writes to secondary memory by multiple systems must be coordinated. For this type of system we present a design for a global controller that, in addition to concurrency-coherency control functions, provides functions that enable the systems to cooperate in order to effectively make use of the shared intermediate memory.

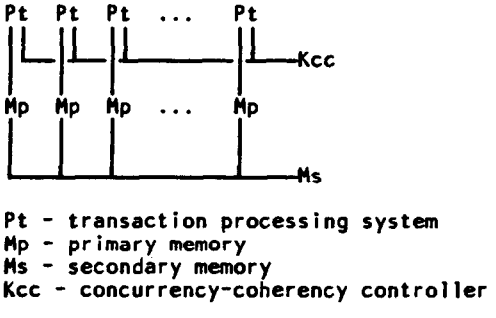

Figure 1.1. Transaction Processing Complex with Shared Secondary Memory

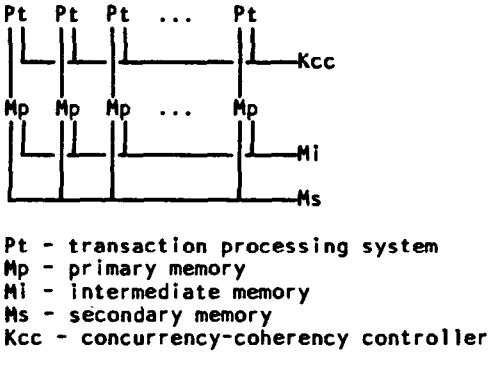

Fig. 1.2. Shared Intermediate and Secondary Memories

Section 2 contains an overview of the system-level transaction processing protocols that we assume are being used for each of these types of systems. Designs for the integrated concurrency-coherency controllers for each type of system are presented in Section 3. In Section 4 a model to evaluate the performance of such systems is described, and modelling results are presented for (1) systems without intermediate memory or an integrated controller, (2) systems with an integrated controller but no intermediate memory, and (3) systems with both an intermediate memory and an integrated controller. Finally, Section 5 contains conclusions.

2. Overview

In this section an overview is given of the system-level transaction processing protocols that we assume are being used in the various types of multi-system data sharing complexes that were described in the Introduction. In all cases we assume that global concurrency control is implemented by block granularity locking. We will take the details of the global concurrency control protocol for granted, with the exception of its interaction with coherency control. First we consider a complex with shared secondary memory only, and in which invalidation messages are broadcast in order to implement coherency control.

2.1. Shared Secondary Memory with Broadcast Invalidation

For this type of complex there must be some kind of inter-system communication mechanism that can be used for system-to-system and broadcast messages. As far as global concurrency control is concerned, there are a variety of implementation techniques: using the inter-system communication mechanism a distributed algorithm based on either partitioning or replicating the global lock table can be used, or alternatively a central controller can be used. The implementation technique will not influence our description of the transaction processing protocols at the level of this section (however, in order to isolate the effect of integrating coherency control, in Section 4 a central controller implementation is assumed).

Each system acquires global locks on blocks in various modes on behalf of the transactions running in that system in order to maintain the consistency of the shared database (finer granularity local locking may be used for intra-system concurrency control). With respect to the interaction between global concurrency control and coherency control we make two assumptions: (1) a global share-mode lock must be acquired for any transaction that, due to the semantics of the database operation being performed, must read the most up-to-date copy of a block (or, if it is known in advance that the transaction will modify the block, a global exclusive-mode lock should be acquired); and (2) global exclusive-mode locks must be acquired for all blocks modified by any given transaction (promoting share-mode locks to exclusive-mode if necessary), and all exclusive-mode locks for modified blocks must be held until the commit point of the transaction. Before a transaction can commit, though, not only must all modified blocks be written to shared disk (in order to make the most up-to-date copy available to the other systems in the complex), but also any private copy of any such block in the buffer of any other system must be marked as invalid (otherwise transactions running on the other system that require the most up-to-date copy would read the obsolete copy). Since for this type of system no global information is maintained on private buffer contents, a message containing a list of the modified blocks must be broadcast to all other systems. Only when all disk writes are complete and all other systems have acknowledged the message can the transaction commit and the global exclusive locks be released.

2.2. Shared Secondary Memory with an Integrated Controller

For this type of complex we assume that a central controller is being used to implement global concurrency control. It is also assumed that share and exclusive mode locks must be acquired as previously described, and that exclusive-mode locks must be held until the modified blocks have been written to secondary memory. Now note the following: since a global share or exclusive mode lock must be acquired for any transaction that requires the most up-to-date copy of a block *before* the block is read by the transaction, if the central controller had information on the validity of all blocks in all buffers in the system,

then the lock request sent to the controller could be combined with a request to check the validity of a private copy of the block. If the lock is acquired but the private copy is invalid then the private copy must be replaced by reading the up-to-date copy from disk. Thus, by having the central controller keep track of buffer contents, the broadcasting of invalidation messages can be avoided.

Now each system must follow protocols in which, in addition to concurrency control requests, requests are sent to the controller that allow global information on the current state of all private buffers to be maintained. The details of these protocols are described in Section 3. It will be seen that coherency control requests can always be combined with concurrency control requests.

2.3. *Shared Intermediate and Secondary Memories*

As shown earlier in Figure 1.2, for this type of complex the existence of a shared intermediate level of memory is assumed. Here, we investigate the use of this memory to speed up the commit process in multi-system data sharing: transactions will be allowed to commit and exclusive locks will be released as soon as all modified blocks are written to intermediate memory. If the memory is volatile, though, it is still required that exclusive locks be held until log records are written to non-volatile memory. However, there are several techniques that can be used to speed up log writes. For example, since log records can temporarily be written to any free disk location, seek times and rotational delay can be minimized (this technique is used in IMS [STRI82]). Another technique is to use a small non-volatile buffer for log records. Therefore, removing the requirement that a transaction cannot commit until all modified blocks are written to disk should greatly improve performance, as will be seen in Section 4.

As described in the Introduction, it is assumed that the management of intermediate memory is partitioned: for each system, there is a separate region of intermediate memory called a partition associated with that system, and an intermediate memory manager running on the system is responsible for allocating, writing, and freeing blocks in the partition. However, since we allow any system to read from any partition, several problems arise. The most obvious problem is locating data blocks in intermediate memory: when a transaction requires the most up-to-date copy of a block, as before the lock request and a check of the validity of a private copy can be combined, but if the private copy is invalid or if there were no private copy, the most up-to-date copy could be in intermediate memory. Therefore, even though the management of intermediate memory is partitioned, it is necessary to maintain global information on intermediate memory contents. As described in the next section, the integrated controller can be extended to keep track of this information. With this extension, when a transaction requires the most up-to-date copy of a block, three requests to the integrated controller can be combined: (1) a lock request, (2) a check of the validity of a private copy, and (3) a request to give the location of a valid copy in intermediate memory if there is one.

Two additional problems have to do with the systems cooperating in their use of intermediate memory. First, one system cannot free and overwrite a block in its partition while other systems are reading the block. Second, when a valid block in one system's partition is modified by a transaction that commits on another system, the fact that the block in the first system's partition is now invalid must be recorded. That is, there is also a coherency control problem for intermediate memory (this is due to the partitioned management).

There is one final problem, but first it is necessary to describe our assumptions regarding recovery protocols and availability. Since the examples of an intermediate level in a memory hierarchy previously mentioned use volatile memory today, for the sake of exactness we assume volatile intermediate memory (however, the basic techniques described here can be applied to the non-volatile intermediate memory case). Therefore, we assume that after modified blocks are written to intermediate memory and a transaction commits, the system immediately schedules writes of all modified blocks to disk, since in the event of a power failure this allows a long recovery scenario to be avoided. In the case that recovery is necessary, log records must be processed only for the transactions that were in progress at the time of the failure and for the transactions that had committed but for which disk writes were not complete. Also, we note that even if intermediate memory is non-volatile it may be less reliable than disk storage, and so it may desirable to immediately schedule disk writes after a transaction commits in order to provide continued operation even if part or all of intermediate memory fails. Another availability concern is that since the central controller is necessary for continuous operation, it should be designed so as to be highly available. In this regard we note that one technique that can be used is to provide a backup controller: if one controller fails, a recovery protocol can be designed in which the information necessary to reconstruct the global state of the complex is down-loaded from all the systems in the complex to the backup controller.

Given that the blocks modified by a transaction are first written to intermediate memory, that next exclusive locks are released, and last that the blocks are scheduled to be written to disk, the final problem can now be described. Suppose transaction T1 modifies a block and issues a commit request. The system will write the block to intermediate memory, release the exclusive lock, and schedule the block to be written to disk. However, before the disk write is initiated, transaction T2 on another system may obtain an exclusive lock on the block, read the copy from intermediate memory, modify it, and issue a commit request. Now the other system will write the block back to intermediate memory, release the exclusive lock, and also schedule the block to be written to disk. If the disk write of the second system is initiated before that of the first system, when the disk write of the first system finally occurs the update of T2 will be undone. This illustrates the final problem: it is necessary to serialize disk writes of the same block by multiple systems so that they are in the order in which the updates occurred (the alternative of somehow "cancelling" previously scheduled disk writes on other systems presents significant distributed control problems).

In summary, for the shared intermediate memory type of complex, in addition to concurrency control and private memory coherency control, the controller must provide functions and the systems must follow protocols that allow blocks to be located in intermediate memory, that prevent a system from overwriting a block in intermediate memory while it is being read by other systems, that invalidate obsolete copies of blocks in intermediate memory, and that serialize disk writes.

3. Design

In this section we present designs for the integrated concurrency-coherency controllers in the last two types of complexes described in the previous section. In order to present the controller designs, it is convenient to consider a typical concurrency control implementation as a starting point, and then describe extensions. For example, Figure 3.1 shows the hash table access structure and linked list block state representation in a typical concurrency control using share and exclusive lock modes: the state shown indicates that transactions T1 and T2 own the block in share mode and that transaction T3 is waiting for the block in exclusive mode. The queued request information shown in the figure refers to the information required for the controller to send a response to the system that issued the request for transaction T3 when the request is eventually granted or rejected. In order to extend this access structure to contain additional control information, we can simply add pointers in the hash table entry for other types of global information associated with blocks. This method will be used in the descriptions below. Such extensions generalize straightforwardly to other kinds of access structures and state representations that might be used in typical concurrency controls. Optimizations designed to minimize overhead such as using combined concurrency-coherency states are possible but somewhat implementation dependent, and will not be described here.

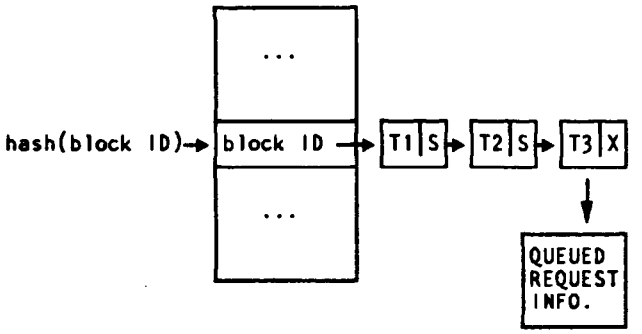

Figure 3.1. Typical Concurrency Control Structure

3.1. Private Memory Coherency Control

The key idea here is that validity of blocks in the private memory buffer needs to be checked at the time the access request for concurrency control is made. A mechanism and protocol is developed to track the validity of the buffer. For the type of complex that has shared secondary memory only, the state of a block in the controller has two parts: (1) an access state (global concurrency control state), and (2) a buffer state (global coherency control state). The buffer state consists simply of a list of the IDs of systems that have a valid copy of the block in their private memory buffer. An example block state is shown in Figure 3.2: this state represents the case in which the access state is as before in Figure 3.1, and the buffer state is that systems S1, S3, and S4 have a valid copy of the block in their respective buffers. The null buffer state is the empty list. When a block's access state and buffer state are both null, the hash table entry for the block can be removed.

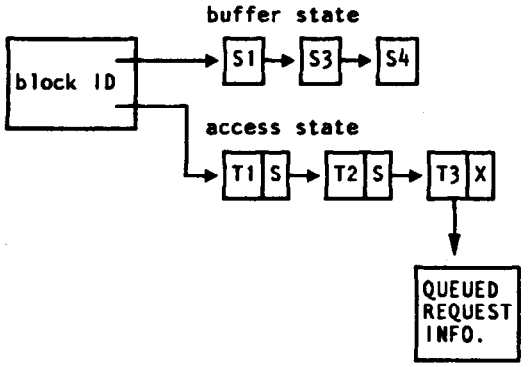

Figure 3.2. Example Access and Buffer States

In addition to access requests, the controller accepts the following buffer requests for blocks from system buffer managers: (1) check and add - if the buffer manager system ID is in the list of systems having the block in their buffer, respond *valid* (indicating that the private copy is valid); otherwise add it to the list and respond *invalid* (indicating that if the system had a copy of the block, it is invalid, and reflecting the fact that the system will now read in a valid copy of the block), (2) remove - remove the system ID from this list; (3) invalidate - remove all system IDs except that of the system making the request from the buffer state list.

The concurrency-coherency controller also accepts combined requests, consisting of an access request followed by one or two buffer requests. In such a case the combined request must be processed atomically (i.e., there can be no interleaving of other requests for the same block). If the access request is queued, the following buffer requests are processed only when the access request is re-processed and granted, and if the access request is rejected then the following buffer requests are not processed.

Each system must use the following protocols.

1. When a transaction accesses a block that is in the local buffer, issue the appropriate lock request combined with a check and add request. Then, if and when the lock request is granted, if the result of the check and add request is *valid* use the local buffer copy, otherwise delete the local buffer copy and read the block from disk.
2. When a transaction accesses a block B1 that is not in the local buffer, in general some other block B2 that is in the buffer must be selected to replace. In this case issue the appropriate lock request combined with a remove request for B2 and a check and add request for B1. Then, if and when the the lock request is granted, read B1 from disk into the buffer replacing B2. If there is free buffer space that is not occupied by any block (for example after system start-up), issue only a lock and a check and add request. (Note that for transactions requiring the most up-to-date copy of a block, the copy is guaranteed to remain valid while the transaction is accessing it due to the global share or exclusive mode lock that is held.)
3. When releasing an exclusive lock for a modified block that has been written to disk, issue an unlock request combined with an invalidate request.

3.2. Intermediate Memory Control

Next, consider the complex with both shared intermediate and secondary memories. In addition to controlling concurrency and maintaining validity information on the blocks in private buffers, the controller will be used to maintain global control information on blocks in intermediate memory.

Each block has an access state and a buffer state as previously described, and each valid copy of a block in intermediate memory also has a non-null intermediate memory state containing the following kinds of information: (1) the address of the block in intermediate memory; (2) a list of systems currently reading the block from intermediate memory; (3) whether the system owning the partition has requested that the slot the block occupies be freed (it cannot be freed while reads from other systems are in progress); and (4) information to serialize writes to disk.

Disregarding the intermediate memory location part, the intermediate memory state can be represented as a "lock" held in various modes by systems (we will see below that the state is really not used at all like traditional lock states, but this is a convenient starting point). These modes are as follows.

W write in progress - held in this mode by the system managing the intermediate memory partition in which the block resides prior to the completion of the write to disk, and requested in this mode by another system that has updated the block

D write to disk complete - converted to this mode by the system owning the partition after the completion of the write to disk

R read in progress - held in this mode by each system reading the intermediate memory copy of the block while the read is in progress

U pending release - this is a pseudo-mode used when the system owning the partition makes a "release" request on the intermediate memory state of the block (see below) in order to free the intermediate memory slot, but there are reads by other systems in progress

We first summarize the protocol for using the integrated controller. Before a transaction can update a data block, an exclusive lock

request for concurrency control is issued with check requests on buffer status in private memory and intermediate memory. Assuming the private memory copy is invalid and there is a valid copy in the intermediate memory (say in D mode), the address in the intermediate memory will be returned, and the mode of the block in intermediate memory is changed to read-in-progress (R) mode to prevent the block being overwritten. After the read is completed, the controller will be notified and will switch back to the previous mode. At commit time, the system first allocates a free slot in the intermediate memory for the updated block and writes the block out to that slot. Then the system issues an unlock access request for the exclusive lock, combined with an invalidate buffer request and a request to change to write-in-progress mode. To serialize disk writes, the change to W-mode request can be granted only after any ongoing disk write for the block has completed. To insure availability of free slots in the intermediate memory, each system periodically frees up slots via release requests, and each release request is either granted immediately if there are no reads in progress to the slot or else granted as soon as all reads currently in progress to the slot have completed.

Let us consider in detail how the controller responds to each type of request. The response depends on the current state of the block.

1. A W-lock request includes the address in intermediate memory into which the block has already been written by the requesting system. The W-lock request will be queued if any write or read is in progress on the block, i.e. the block is in W or R mode. Otherwise, the request is granted, and the address part of the intermediate memory state of the block is updated. When a W-lock request is granted, if another system owns the lock in D-mode, the controller automatically performs an unlock on behalf of the D-mode owner, effectively invalidating the copy of the block in the partition owned by that system.
2. When an R-lock request is received for a block with a null intermediate memory state, the request is rejected, thus indicating that the block must be read from disk. Similarly an R-lock request is rejected if there is a queued release request for the block (in order to avoid possible starvation of the release request). Otherwise, when the request is granted, the result contains the intermediate memory location of the block. When an R-lock request is received for a block from the system owning the block in D-mode, the location is returned as in the previous case, but the mode remains D, thus indicating to the system that the copy of the block in its partition is still valid.
3. When a D-lock request is issued upon completion of the disk write, the W mode will be converted into D-mode. Any other system owning the lock in R-mode will not be affected.
4. When a release request is received from a system that is not currently an owner (because the intermediate memory copy in the system's partition was previously invalidated by means of a W-lock request from another system as described above), the request is granted. When a release request is received for a block from a system owning the block in D-mode, if there are any R-mode owners, the controller treats this as a U-conversion request, thus queueing the release request.

The control algorithm can be described by using a "standard" locking-based algorithm using the mode-compatibility matrix shown in Figure 3.3 (certain cases such as the way mode-conversion requests are handled will not be described in detail, but will be indicated by the example at the end of this section).

Similarly to the previous design, the controller can accept combined requests consisting of intermediate memory, access, and buffer requests, and combined requests must be processed atomically. In combined requests, in some cases an access request should be processed first and in other cases an intermediate memory request should be first. This can be controlled by having the system issuing the combined request order the sub-requests appropriately. If the first request in a combined request is queued the remaining requests should also be queued, and if the first request is rejected the remaining requests should be rejected, with the following exception: if an R-lock request is rejected, associated access and buffer requests *are* processed as before. Also, a block's global state is null if each of its access, buffer, and intermediate memory states are null.

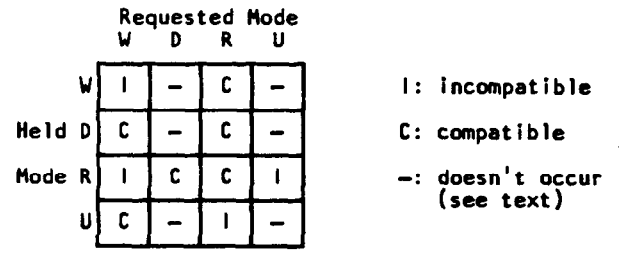

Figure 3.3. Intermediate Buffer State Mode Compatibility Matrix

An outline of the protocols that must be used by the intermediate memory managers running on each system is as follows.

1. To free an intermediate memory slot, issue an intermediate memory release request. When the request is granted, the slot can be used.
2. To check if a block is in intermediate memory, issue an R-lock request. If the request is granted, the result contains the location.
3. If an R-lock request is issued for a block that was previously written to intermediate memory by the intermediate memory manager and the location returned is different than that previously allocated, or if the request is rejected, the copy is now invalid, and the intermediate memory slot is freed.
4. After allocating a slot and writing a block to intermediate memory, issue a W-lock request. When the request is granted, any write to disk of this block by another system has completed, and a write to disk can be initiated by this intermediate memory manager, thus guaranteeing correct disk write serialization.
5. After a disk write of a block completes, issue a D-conversion request.

Analyzing the algorithms used by controller and the intermediate memory manager protocols, it can be seen that certain combinations of held and requested modes never occur, as shown in Figure 3.3.

We conclude this section with an example illustrating some of the above protocols and global block states.

1. S1 allocates an intermediate memory buffer slot by issuing a successful intermediate memory release request, writes the block to intermediate memory, issues a successful W-lock combined with an access state unlock and invalidate requests (the access state of the block was that it was held exclusive by some transaction on S1), and initiates a write to disk. The block state is shown in Figure 3.4(a).
2. Prior to the completion of the disk write, transactions T1 and T2 on S2 and S3 respectively read the block, first issuing successful R-lock combined with add and access share lock requests, resulting in the state shown in Figure 3.4(b).
3. The read from intermediate memory by S3 completes, and S3 issues an intermediate memory release request, resulting in the state shown in Figure 3.4(c).
4. The write to disk completes, and S1 issues a D-conversion request, resulting in the state shown in Figure 3.4(d).

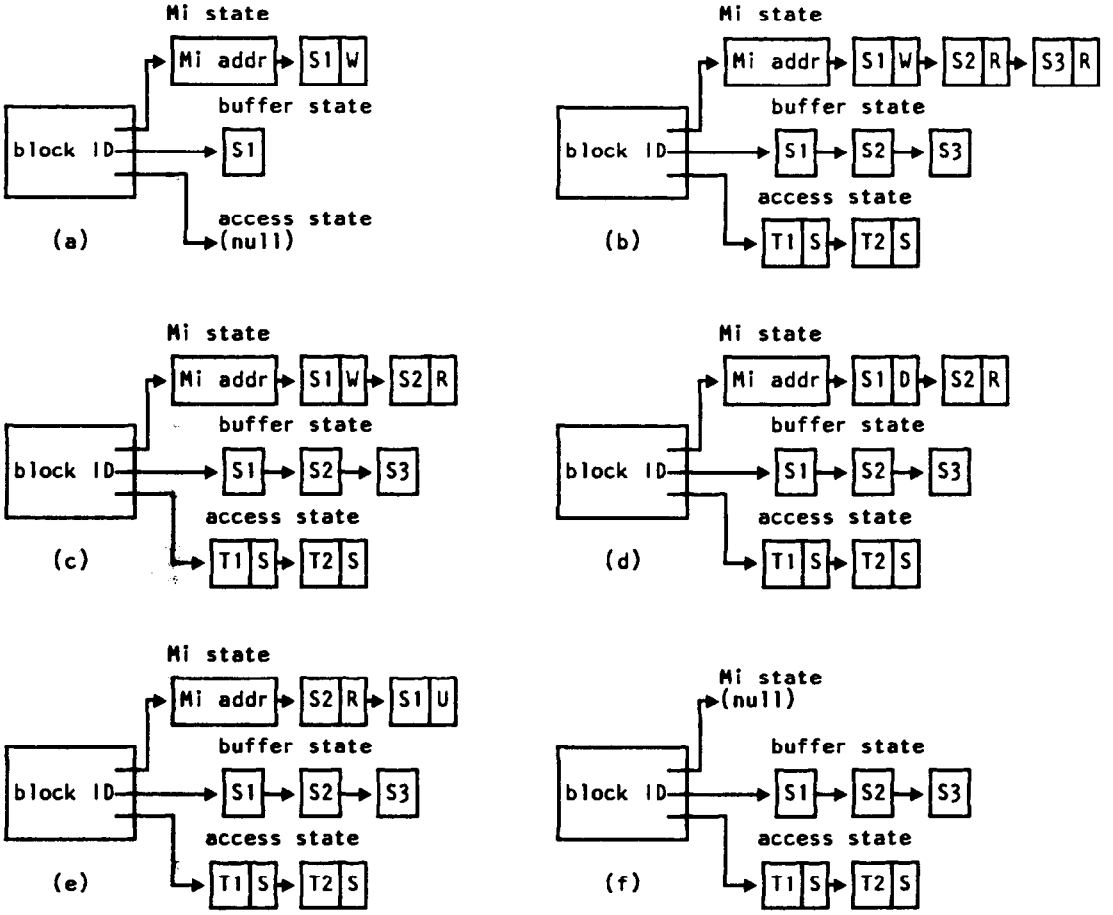

Figure 3.4. Example Access, Buffer, and Mi States

5. To free the buffer slot, S1 issues an intermediate buffer release request, but because there is a read in progress the request is queued, as shown in Figure 3.4(e).

6. The read from intermediate memory by S2 completes, S2 issues an intermediate memory release request, S1's queued release request is then processed, resulting in the null intermediate memory state as shown in Figure 3.4(f).

4. Performance

This section outlines a model to estimate the perfomance of the integrated concurrency-coherency control schemes presented in Section 3, and illustrates the benefits of this technique. Qualitatively, the following effects occur. Using intermediate memory with the integrated concurrency-coherency protocol of Section 3.2 permits the early release of locks. This leads to shorter lock holding times, and lower lock contention. Thus, when the transaction rate is increased, either by increasing the number of coupled systems or by using more powerful processors, significant improvement in response time is expected. Further, the integrated concurrency-coherency scheme eliminates the increase in protocol overhead with number of systems that occurs with broadcast invalidation. This is particularly important as the number of coupled systems increases. Finally, blocks that are frequently updated will be found in the intermediate memory, further reducing I/O time. In particular, updated blocks that are contended for will be found in the intermediate memory, leading to significant improvement at high contention levels. These effects are quantified in this section.

4.1. The Model

The model uses parameters derived from traces taken from two large intallations running IBM's IMS database management system [YU85B]. The first was an inventory parts database traced for 15 minutes, and the second was an on-line materials planning database system traced for 60 minutes. Both traces lead to similar results. The results presented in this paper are for the first trace. The traces from the single system environment were used to derive traces for a multi-system environment as described in [YU87]. In a trace driven simulation of two coupled 14 MIPS systems with an average workload of 20 transactions per second per system, the probability of lock conflict was found to be about 0.007 per lock request. This lock conflict probability is projected to other system configurations by the approximate analytical model presented below. A database block reference trace was derived from the multi-system lock traces. The block reference trace was used to drive a simulation of the database buffer manager under different coherency control protocols. The increase in database I/O caused by this phenomenon was captured. The broadcast invalidation and integrated concurrency coherency protocols were found to have about the same increase in I/O, although buffer space occupied by obsolete blocks is reused earlier by the buffer invalidation

protocol in comparison with the integrated concurrency coherency protocol. Thus, the major difference of the two protocols is in overhead.

In the model, transaction arrivals are modelled by a Poisson process. A front-end server is assumed to assign transactions randomly to the different systems. A balanced load on all systems is assumed. The average transaction response time R is expressed in terms of its components as follows [YU85A] :

$$R = R_{CPU} + R_{IO} + R_{CONT} \qquad (4.1)$$

Each component will be described separately.

R_{CPU} is the total time the transaction spends at the CPU. This includes both the CPU service and queueing times, and the time the CPU waits for a lock from the integrated concurrency coherency controller. From the trace analysis the average number of instructions that need to be executed per transaction was found to be 430,000. Lock requests and database I/Os are assumed to occur uniformly over this transaction path length, breaking the transaction into many small segments of equal size. In addition, each lock request entails sending the lock request to the integrated controller, and waiting for the reponse. For a given transaction rate, the rate of lock and unlock requests at the integrated controller is known. Assuming Poisson lock request arrivals at the integrated controller, the average service and queueing time is computed [YU85A]. The Poisson approximation is reasonable for a large number of concurrent transactions. This time is added to the CPU time and the total CPU utilization is computed. Then R_{CPU} is computed by modeling the (dyadic) CPU as and M/M/2 queue.

In addition to the concurrency control protocol overhead, the coherency control protocol overhead is modelled as follows. The model for the broadcast invalidation coherency control protocol assumes that 1K instructions of local processing is incurred by the system initiating the invalidate message, before incurring the overhead of broadcasting the message to the remaining $n - 1$ systems. The overhead O_{com} for sending a message between systems is assumed to be 3K instructions equally split between the sending and receiving systems. The broadcast overhead is assumed to be the same as the overhead for sending a message, $O_{com}/2$. Each of the $n - 1$ remaining systems incurs an $O_{com}/2$ overhead for receiving the invalidate message, a 1K instructions overhead each for processing the invalidate message, and an overhead of $O_{com}/2$ for sending an acknowledgement in reply. The model for the integrated concurrency coherency protocol assumes that every block referenced is locked before being accessed. The validity of the data block being accessed is verified by the integrated controller at the time that the lock request is processed. We assume a 50% extra overhead for lock processing in order to perform this validity check.

R_{IO} is the amount of time spent during the transaction, waiting for I/O to occur. Note that for each I/O the processor is modelled to task switch to process another transaction, while suspending the executing transaction, until the completion of the I/O. Thus,

$$R_{IO} = t_{IO} n_{IO}$$

where t_{IO} is the average time per I/O and n_{IO} is the average number of I/Os per transaction. Sufficient I/O bandwidth is assumed to enable the modelling of the I/O server as an infinite server with a load independent service time of 35 mS. n_{IO} consists of two kinds of I/Os: n_{IOPL} and n_{IODB} . n_{IOPL} is the average number of non-database I/Os per transaction. Typically, these are the I/Os needed to load the application program and the other constructs into the main memory from disk. Trace analysis yielded a value of 5 for n_{IOPL}. n_{IODB} is the average number of I/Os that occur during the execution of the transaction. These are required to read and write data from disk resident databases into and from the main memory buffer, respectively. In the trace analysis, the average transaction performed 11 database I/O in the single system environment. In the model for the coupled systems, the average number of extra I/O's incurred under the different coherency control protocol is added to 11 to arrive at a number for n_{IODB}. The extra I/O due to invalidation is modeled as proportional to the probability that transactions running external to a system cause the invalidation of a block in the buffer. Let α be the probability of an invalidation caused in a system's buffer over a fixed time period due to transactions executing at a rate of one transaction per second, external to the system. Thus, the probability of buffer invalidation due to n_i such independent streams of transactions is $1 - (1 - \alpha)^{n_i}$. The increase in I/O, Δ_{IO}, is modelled as

$$\Delta_{IO}(n_i) = \Delta_{max}(1 - (1 - \alpha)^{n_i}), \qquad (4.2)$$

where Δ_{max} is the constant of proportionality. The increase in I/O can reach a maximum value corresponding to the increase in I/O if blocks used by a transaction are purged at commit time; this gives Δ_{max}, and was found by trace driven simulation to be 4 I/Os. From trace analysis, we found that $\Delta_{IO}(20) = 0.11$. Substituting in equation (4.2), $\alpha = 0.001393$. The model for predicting the increase in database I/O, described by equation (4.2) was validated for the coupling of two, three, and four systems, through simulation of the buffer manager, driven by the derived multi-systems traces, described earlier. Hence,

$$n_{IO} = n_{IOPL} + n_{IODB},$$

$$R_{IOPL} = t_{IO} n_{IOPL}, \qquad R_{IODB} = t_{IO} n_{IODB}$$

$$R_{IO} = R_{IOPL} + R_{IODB}. \qquad (4.3)$$

R_{CONT} is the time spent in contention wait for a lock that is held by another transaction. The contention wait is estimated as,

$$R_{CONT} = L \; P_{CONT} \; \overline{W} \qquad (4.4)$$

where L is the average number of lock requests per transaction derived from trace analysis and is found to be 15. P_{CONT} is the probability of contention on a lock request, and the product $L \; P_{CONT}$ is the average number of contention waits per transaction. The manner in which P_{CONT} is projected from the value measured in the trace driven simulation is described later in this section. $\overline{W}$ is the average time a transaction waits, if it contends with another transaction for a lock, and is a function of the transaction response time. This wait time is estimated as $(R - R_{IOPL}) / F$, where F is estimated below. The reason R_{IOPL} is subtracted from the reponse time is because the transaction does not hold any locks during the time these n_{IOPL} I/Os are being carried out. For shorthand notation we define $\overline{R} = R - R_{IOPL}$. Substituting this estimate for $\overline{W}$ in (4.4) , and using (4.4) with (4.1) and (4.3) gives,

$$R = R_{IOPL} + \frac{R_{CPU} + R_{IODB}}{1 - \left(\frac{P_{CONT} \; L}{F}\right)} \qquad (4.5)$$

The reciprocal of the denominator in (4.5) indicates the expansion in the response time due to lock contention wait.

The factor F for the fraction of the response time that represents the expected waiting time, is estimated as follows. We neglect the probability of restart due to deadlocks. The mean wait time of a transaction that contends for lock is the mean remaining time of the transaction that it contends with. It is assumed that the transaction makes lock requests evenly over its execution. Thus, there are an equal number of transactions (including waiting and running transactions) that hold different numbers of locks. The probability of contention with a transaction is proportional to the number of locks that a transaction holds. Using a continuous time approximation, the mean remaining time, r, is

$$r = \int_0^{\overline{R}} P(x) \, (\overline{R} - x) \, dx \; ,$$

where P(x) is the probability of contention with a transaction that has run for time x, and $\overline{R}$ is the transaction response time, during which the transaction holds locks. Using the continuous approximation that a

transaction holds a linearly increasing number of locks with time, P(x) equals $x / \int_0^{\bar{R}} x\ dx$, which gives,

$$r = \frac{\int_0^{\bar{R}} x\ (\bar{R} - x)\ dx}{\int_0^{\bar{R}} x\ dx} = \frac{\bar{R}}{3} . \qquad (4.6)$$

Thus, F in equation (4.5) is estimated as 3.

The response time for the two coupled 14 MIPS systems at 20 transactions per second and contention probability of 0.007 (as measured from the trace) was derived to be 0.65 seconds, using Equation (4.5). It is assumed that the contention probability grows as the product of the transaction rate and response time. Using this assumption the lock contention observed in the trace is projected to higher transaction rates and more coupled systems. The assumption is consistent with the asymptotic results in [GRAY81] [LAVE84]. The response time is affected by the size of the processor. The *feedback* effect of the contention probability varying with response time is estimated using an iteration. The transaction response time is first derived for the initial estimate on contention probability. Then the resulting response time is used to compute a new contention probability by assuming that the contention probability grows as the product of the transaction rate and response time. The approximate model is then run again with the new contention probability. The iteration is repeated until convergence is obtained. Only a few iterations are required in the contention range considered. This approximate model for the average response time in the data sharing environment is validated to within 5% of the results from simulation, described in [YU85A], for two systems coupling over a wide range of lock contention levels.

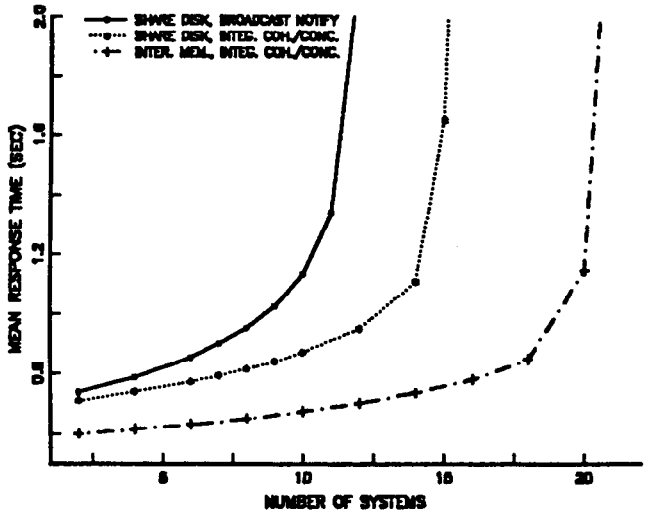

Figure 4.1. Performance Advantage of Integrated Concurrency-Coherency Control

4.2 Performance Projections

We now examine some performance projections from this model. We first consider the case where the intermediate memory is used only to store updated blocks. That is, it is assumed that blocks are held in intermediate memory only till the write of the block to disk is completed. This allows for early transaction commit, and for any transactions waiting to access updated blocks to obtain these blocks directly from intermediate memory. A relatively small intermediate memory should be sufficient for this purpose. Figure 4.1 shows the projected response time as a function of the number of coupled systems for shared disk with broadcast invalidation, and for the integrated concurrency-coherency control scheme with and without intermediate memory. In this graph, the transaction rate per system is kept constant at 20 transactions per second per system, as the number of systems in the complex increases. Thus, the contention grows with the number of coupled systems. With the broadcast block invalidation scheme, the processing overhead for the broadcast and receipt of acknowledges for the invalidation from the other systems, also grows with number of systems. Both these effects combine to restrict the number of coupled systems to 10 if sharing is only at the disk level and broadcast invalidation is used. For the integrated concurrency-coherency scheme, the overhead for the coherency control does not increase with the number of systems; this results in significant improvement as the number of coupled systems increases. With this scheme, the limitation on the number of systems is due to contention rather than processing overheads. Using intermediate memory with the integrated concurrency-coherency control scheme results in a large improvement in response time and number of coupled systems. Both I/O time and lock contention level are reduced by the integrated protocol. When a small number of systems are coupled, it is only the reduction in the I/O time that affects the transaction response time. When a large number of systems are coupled the combined effect of reduced I/O and contention become apparent through a large reduction in transaction response time.

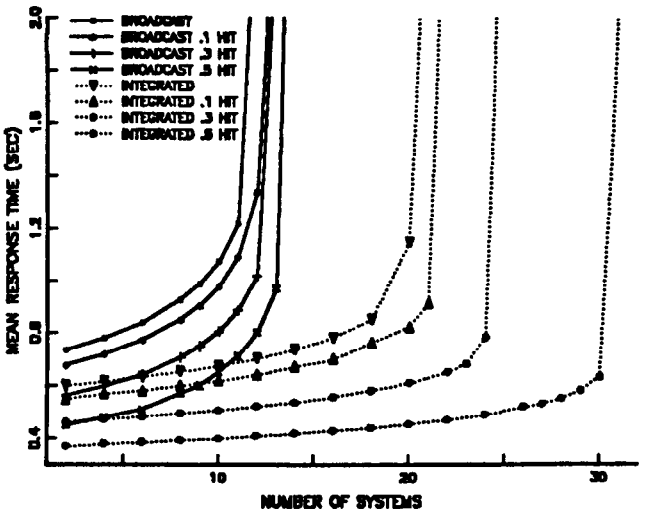

Figure 4.2. Effect of Buffer Hits in Intermediate Memory

We now consider the case where updated blocks are retained in the intermediate memory after the write to disk is completed. For this case frequently updated blocks will often be found in the intermediate memory. We observe that the intermediate memory can be used in a straighforward manner for broadcast invalidation with no early lock release; updated blocks are written both to the intermediate memory and to disk, locks are retained until the disk writes are complete, and the intermediate memory address of the block is broadcast along with the block invalidate. The effect of this retention of updated blocks in intermediate memory is modeled as read hits in Figure 4.2. The curves shown with solid lines are for broadcast invalidation, and the dashed lines are for the integrated concurrency-coherency control scheme. For both cases, the reduction in reads results in a smaller transaction response time for a small number of coupled systems. However, the intermediate memory hits do not substantially enhance the number of coupled systems with the broadcast invalidation, because it is the processing overhead that is the limitation. With the integrated concurrency-control scheme, read hits in the intermediate memory have a significant effect in enhancing the number of systems that may be coupled.

5. Conclusion

In a multi-system data sharing complex, inter-system interference due to global concurrency control and coherency control of local buffers in each system can lead to degraded performance. We observed that while globally locking an item for concurrency control, a check could simultaneously be made to determine if the system requesting the lock had a valid copy of the block being locked. This was the basis for handling combined requests for concurrency and coherency in an integrated controller. The integrated controller maintains a list of systems that hold a valid copy of the block, and when a system releases an update lock with an update, all other systems are deleted from this list. This method leads to a large reduction in overhead for coherency control as compared to broadcast buffer invalidation.

Next we considered the use of a shared intermediate memory to enhance performance. The intermediate memory can be employed as a shared buffering device to reduce disk IO's. Further, updated blocks that are invalidated at other system's local buffers can now reside in the shared intermediate memory. Thus, blocks that are frequently updated, and therefore lead to lock contention, will reside in intermediate memory, leading to a reduced holding time for these contended blocks. Finally, with proper controls, the intermediate memory can be used for early transaction commit, i.e. before disk writes are completed, by allowing release of locks after updates have been written to the shared memory. Control of such an intermediate memory requires handling of read-write synchronization and disk write serialization, which we show can be done by the integrated controller. A queueing model developed to evaluate the system performance indicates that a significant transaction speedup and reduction in lock contention between transactions can be obtained. Even without intermediate memory, our analysis shows that the special case of the integrated concurrency-coherency control can improve the performance over broadcast invalidation by reducing the protocol overhead. With limited intermediate memory for early commit, the integrated control can significantly enhance the performance. When intermediate memory is employed for buffering, substantial performance gains can be realized only through use of the integrated concurrency-coherency control protocol. The reduced contention and overhead imply that a larger number of systems can be coupled together using this integrated control than without it.

References

[CORN86] Cornell, D.W., Dias, D.M., and Yu, P.S., "On Multisystem Coupling Through Function Request Shipping", IEEE Trans. on Software Engrg., 12,10 (Oct. 1986) 1006-1017.

[GRAY81] Gray, J., Homan, P., Obermarck, R. and Korth, H., "A Straw Man Analysis of Probability of Waiting and Deadlock", IBM Research Report RJ 3066, San Jose, California (1981).

[LAVE84] Lavenberg, S., "A Simple Analysis of Exclusive and Shared Lock Contention in a Database System", *Performance Evaluation Review 12,3* (Proc. 1984 ACM SIGMETRICS Conference), 143-148.

[SEKI84] Sekino, A., Moritani, K., Masai, T., Tasaki, T., Goto, K., "The DCS - A New Approach to Multisystem Data-Sharing," Proc. National Computer Conference 1984, Las Vegas, NV (July 1984).

[SMIT85] Smith, A.J., "Disk Cache Miss Ratio Analysis and Design Considerations", ACM Trans. on Computer Systems, 3,3 (1985), 161-203.

[STRI82] Strickland, J. P., Uhrowczik, P. P., and Watts, V. L., "IMS/VS: An Evolving System", *IBM Systems Journal 21*, 4 (1982), 490-510.

[YU85A] Yu, P.S., Dias, D.M., Robinson, J.T., Iyer, B.R. and Cornell, D., "Modelling of Centralized Concurrency Control in a Multi-system Environment", *Performance Evaluation Review 13*, 2 (Proc. 1985 ACM SIGMETRICS Conference), 183-191.

[YU85B] Yu, P.S., Dias, D.M., Robinson, J.T., Iyer, B.R. and Cornell, D., "Distributed Concurrency Control Analysis for Data Sharing", *Proc. 16th Computer Measurement Group Conference*, Dallas, TX (Dec. 1985), 13-20.

[YU86] Yu, P.S., Cornell, D.W., Dias, D.M., and Thomasian, A., "On Coupling Partitioned Database Systems", Proc. 6th International Symposium on Distributed Computing, Boston, MA (May 1986), 148-157.

[YU87] Yu, P.S., Dias, D.M., Robinson, J.T., Iyer, B.R. and Cornell, D.W., "On Coupling Multi-Systems Through Data Sharing", Proceedings of the IEEE, May 1987.