A Fast General-Purpose Hardware Synchronization Mechanism

John T Robinson
IBM T J Watson Research Center
P O Box 218
Yorktown Heights, New York 10598

Abstract. One way to reduce the overhead of concurrency control in a multiprocessor transaction processing system is to implement an underlying synchronization mechanism or a simple global concurrency control directly in hardware The problem with this approach is that a strong commitment may then be made to a particular synchronization protocol, and so the resulting hardware mechanism may be useful in only a very narrow range of systems A solution is possible using a table-driven approach However, a straightforward table-driven approach is impractical due to the extremely large table sizes required for many protocols It is shown here that this problem can be solved by reducing the table sizes required by making use of the processor symmetry that occurs in most systems The resulting algorithm for a table-driven synchronization mechanism is not only general-purpose but also extremely fast An example hardware implementation of this algorithm is presented, and practical experience using this approach is described

1. Introduction

The effect of concurrency control overhead in transaction processing systems and the trade-off between this overhead and the granularity of concurrency control has been widely studied (e g , see [Ries and Stonebraker 77,79], [Thomasian and Ryu 83], [Carey 83], [Tay 84]) In addition to attempting to choose the best granularity, the approach of trying to reduce concurrency control overhead as much as possible can in practice be quite valuable For example, in the case of System R, Gray noted that a careful re-design of the synchronization mechanism required to enter the lock manager reduced the number of instructions for a latch-unlatch pair from 120 to 10 [Gray 78]

This second approach to the problem of concurrency control overhead is the subject of the current work Since a transaction processing system's concurrency control is typically implemented using some underlying synchronization mechanism (e g the latch-unlatch mechanism mentioned above, or the send-receive mechanism in a message-based system), there are essentially two techniques that can be used to reduce concurrency control overhead (1) reduce the overhead of the underlying synchronization mechanism, and (2) reduce the number of instructions executed for the most common concurrency control invocations Here the primary emphasis is on the former technique, although it will be seen that the mechanism presented is also directly applicable to certain relatively simple concurrency control protocols In particular, in multiprocessor systems one technique for concurrency control is to use a two-level protocol in which (1) a global concurrency control grants access to data on a processor basis as requested by local concurrency controls running on each processor, and (2) each local concurrency control then grants access to data as requested by transactions running on the same processor In a variety of cases the mechanism presented here is applicable at the global concurrency control level when using such a two-level protocol

The obvious strategy for reducing synchronization overhead, or the overhead of global concurrency control in a two-level protocol, is to directly implement the synchronization mechanism in hardware The problem with this strategy is that since there are a variety of synchronization protocols in use, and since one can reasonably expect different protocols to be developed in the future, the resulting hardware synchronization mechanism may be applicable only to a very narrow range of systems, and the expense of hardware development may not be justified in such a case As an illustration of the variety of protocols that might be considered in an actual system, four example synchronization protocols are described in Section 2

Here, the problem of designing a general-purpose hardware synchronization mechanism is addressed by using a table-driven approach, the idea being that once

the mechanism is implemented in hardware, in order to implement any particular protocol it is only necessary to load the appropriate table Furthermore, determining state changes of synchronization entities by table-lookup is extremely fast However, a straightforward table-driven approach, as described in Section 3, is impractical for all but the simplest types of protocols as a result of the extremely large table sizes required A solution to this problem is possible due to the symmetry of the processors executing transactions that occurs in most systems Using this symmetry, the table size can be reduced to reasonable values for a wide variety of protocols This reduction method is presented in Section 4 The resulting algorithm for a table-driven synchronization mechanism can easily be implemented in hardware, as shown in Section 5 Finally, practical experience in using this approach is described in Section 6 and conclusions are drawn

2. Example Synchronization Protocols

In order to illustrate the variety of synchronization protocols for which hardware support could prove valuable, in this section four methods for concurrency control in a multiprocessor system will briefly be described (the first two methods are well known, and the third method is currently in use) Since these are only examples, details such as techniques for global deadlock detection when using a concurrency control method that permits it have been omitted

2.1. Token Passing

In a multiprocessor system with both shared and local memories, one technique for implementing a global concurrency control is to store the data structure containing the state of each currently accessed data object in shared memory, and to synchronize accesses to this data structure made by local concurrency controls running on each processor In order to provide increased concurrency, this data structure may be split into an arbitrary number of partitions (for example, if object IDs are integers, a request for the object with ID i would be processed by accessing partition i mod n, where n is the number of partitions) Also, partitions may be stored in separate memory modules that can be accessed in parallel in order to avoid processor-memory interference

In such a system one of the simplest synchronization protocols that provides a degree of fairness is token passing a token is associated with each partition, and only the processor holding the token for a given partition can access that partition Additionally, each processor has a "next" processor in a ring of processors If a processor currently holds the token for a partition, then when all outstanding requests that map to the partition have been processed, or when a fixed time interval has expired, the token is "passed" to the next processor in the ring Passing a token means signalling the next processor in some fashion and transferring the ID of the partition

2.2. Exclusive Locks with Queueing

Again assuming a multiprocessor system with shared and local memories, and with a shared global concurrency control data structure that has been partitioned, rather than accumulating requests for a partition which are processed when a token is received, an exclusive lock can be associated with each partition that controls access to that partition Using exclusive locks, one synchronization protocol is the following (1) a local concurrency control processes a request that maps to a given partition by first requesting the lock for the partition, accessing the shared data structure when the lock is granted, and then releasing the lock, (2) to provide fairness and prevent starvation, processors making conflicting requests for a given lock are queued on the lock in FIFO order, and (3) when a lock is granted to a waiting processor, the processor is signalled in some fashion and the ID of the partition transferred

The above synchronization protocol is similar to the use of binary semaphores to control access to data structures by multiple processes in a multiprogrammed system Here the term "lock" has been used because if the hardware synchronization mechanism has enough memory for lock states and uses an associative lookup mechanism for lock names, an option is to use the synchronization mechanism directly to exclusively lock data objects on a processor-ownership basis This is the general approach used by the facility described next

2.3. The Limited Lock Facility used by ACP

In order to support concurrent access to databases on shared disks by multiple ACP systems (ACP is widely used in airlines reservation systems -- see [Siwiec 77]), a "limited lock facility" (LLF) was developed (see [Behman et al 79]) With the LLF installed in a disk control unit, four additional channel commands are available that can be used to lock data, read the lock table, etc

The LLF provides exclusive locks with processor ownership, supports 5 byte lock names, and has memory for 512 lock states (in each control unit) However, rather than using a FIFO queue for processors waiting on a lock and automatically granting the lock to a waiting processor when it is unlocked by the current owner, the designers of this facility chose to use the following synchronization protocol when a lock with waiting processors is unlocked by the current owner, the lock state becomes "unlocked-pending", *all* waiting processors are interrupted, and the first processor to re-request the lock becomes the owner The special unlocked-

pending state is used to prevent the memory allocated for the lock state in the control unit being freed The advantage of this approach is that it gives priority to lightly loaded processors, which probably outweighs the disadvantage of possible starvation in the high throughput environment for which ACP is intended

2.4. Shared-Exclusive Locks with Queueing and Conflict Notification

Instead of only supporting exclusive access to data structures or data objects as in the previous examples, it may be desired to imbed more of the system's concurrency control directly in the hardware synchronization mechanism For example, in a system using locking with a share mode for readers and an exclusive mode for updaters, a local concurrency control could use a protocol based on the following two principles to request locks on a processor-ownership basis (1) in order to grant a lock locally to a transaction in share mode, it must be globally owned by the processor in share or exclusive mode, and (2) in order to grant a lock locally in exclusive mode, it must be globally owned in exclusive mode

Using these two principles, it is possible to develop various algorithms for local concurrency control To provide fairness, processors waiting for a lock can be queued in the hardware synchronization mechanism in FIFO order in a series-parallel graph structure, with parallel waiting used for share mode and series waiting used for exclusive mode However, due to the two levels of ownership, there may be some problems that do not occur when using locking with a single level of ownership First, assume that processor P_1 owns a lock in share mode, and that processor P_2 is waiting for the lock in exclusive mode Then if local transactions keep requesting the lock in share mode on processor P_1 so that the lock is always locally owned in share mode by at least one transaction (this could actually happen, for example, if the lock was for the root node of a tree-structured index that is used by every transaction), processor P_2 is subject to starvation Second, suppose data objects are locally cached in each processor In order to prevent a data object that is in the cache of processor P_1 being invalidated due to an update made to the object on processor P_2, it may be desired to have the local concurrency control of P_1 continue to hold a lock on the object as long as it is in the cache, even after all local transactions accessing the object have completed

Both of these problems can be solved by using a *conflict notification* protocol in the hardware synchronization mechanism That is, in addition to signalling a processor and transferring the lock name when it is granted a lock for which it has been waiting, the processor may be signalled and the lock name and conflict information transferred when another processor requests the lock in a conflicting mode By providing a conflict notification mechanism, the previously described problems can be solved as follows (1) if a global conflict occurs on a lock held by P_1 that is currently locally held by one or more transactions, all future local requests for the lock are (locally) queued, and the lock is released by P_1 as soon as all local transactions currently holding the lock have released it (the lock would then be re-requested by P_1 if the local queue for the lock was non-empty), and (2) if a conflict occurs on a lock held by P_1 that is not currently locally held by a transaction but rather is held because it is a lock on a data object in the local cache, then the object is flushed from the cache and the lock is released

3. A Straightforward Approach

The synchronization protocols described in the previous section are only examples, it is easy to design many other protocols that could conceivably be useful in practice For example, it could be useful to have hardware support for synchronization protocols designed specifically for particular data structures such as stacks or queues, or for particular functions such as buffer management or commit processing The general case can be formalized as follows

1 There are p processors, $P_1, P_2, \ldots, P_p$

2 There is a collection of *synchronization entities*, each referred to by some name E (a synchronization entity could be used as a token, a semaphore, a lock, etc , depending on the synchronization protocol)

3 Each processor can make one of r requests, $R_1, R_2, \ldots, R_r$, against any synchronization entity E

4 Each synchronization entity is initially in state S_1, and as a result of the sequence of requests made against it in the past, is at any point in time in one of s states $S_1, S_2, \ldots, S_s$

Using this formalism, any synchronization protocol can be specified using two functions, a *transition* function T $\{P_i\} \times \{R_j\} \times \{S_k\} \rightarrow \{S_k\}$ and a *notification* function N $\{P_i\} \times \{R_j\} \times \{S_k\} \rightarrow 2^{\{P_i\}}$, as follows if processor P_i makes request R_j against a synchronization entity E currently in state S_k, then the new state of E is given by $T(P_i, R_j, S_k)$, and each processor in the set $N(P_i, R_j, S_k)$ is signalled and the name E and new state $T(P_i, R_j, S_k)$ transferred to the processor (An alternative is to assume that there is a set of request results, and to transfer names and request results, however, the approach of transferring the new state is equally powerful, but simpler for the purposes of presentation)

The straightforward way to implement a general synchronization mechanism of this type is to encode the transition and notification functions in two tables T and N such that

i	j	k	T[i,j,k]	N[i,j,k]
1	1	1	-	-
1	1	2	1	1,0
1	1	3	-	-
1	1	4	3	1,1
1	1	5	-	-
1	2	1	2	1,0
1	2	2	-	-
1	2	3	5	1,0
1	2	4	-	-
1	2	5	-	-
2	1	1	-	-
2	1	2	-	-
2	1	3	1	0,1
2	1	4	-	-
2	1	5	2	1,1
2	2	1	3	0,1
2	2	2	4	0,1
2	2	3	-	-
2	2	4	-	-
2	2	5	-	-

Figure 1 Transition and Notification Tables

$$T[i,j,k] = k' \text{ if and only if } T(P_i, R_j, S_k) = S_{k'},$$

and where $N[i,j,k]$ is a bit-vector of length p such that

$$N[i,j,k][i'] = 1 \text{ if and only if } P_{i'} \in N(P_i, R_j, S_k)$$

A very simple example is exclusive locks with queueing for two processors In this case the requests could be $R_1 = unlock$ and $R_2 = lock$, the states could be $S_1 = unlocked$, $S_2 = locked\ by\ P_1$, $S_3 = locked\ by\ P_2$, $S_4 = P_2 \rightarrow P_1$, and $S_5 = P_1 \rightarrow P_2$ (the notation $P_2 \rightarrow P_1$ represents the case in which P_1 holds the lock and P_2 is waiting, and similarly for $P_1 \rightarrow P_2$), and the transition and notification tables could be as shown in Figure 1 For the notification table it has been assumed that a processor should always be notified of the result of its own request State transitions that are not supposed to occur in this protocol (like unlocking an unlocked lock or a lock that is not owned) have been left blank in the tables If it is desired to detect errors in the use of the synchronization mechanism, this can be done by adding additional "error" states, or by associating request results with each state transition (such as "OK", "wait", or "error") as described above Note that conflict notification (as described in Section 2 4) can be added to the above protocol by changing the entries for N[1,2,3] and N[2,2,2] to "1,1"

It is also necessary to implement some method for storing and finding the state of each synchronization entity There are several well known alternatives for doing this, and the method used is independent of the synchronization mechanism developed here Some examples are as follows

1 If there is a relatively small fixed set of synchronization entities (such as in the examples of Sections 2 1 and 2 2), the state of all synchronization entities can be kept in fast memory at all times, and the offset in memory of the state of a synchronization entity can be used as its name

2 If it is desired to use the synchronization mechanism directly to control access to data objects, in most cases it is not feasible to store the state of all objects in fast memory However, the technique of hashing object names into a smaller space may give acceptable performance if it is unlikely for objects with the same hash value to be in use concurrently Using this approach, all objects with a given hash value are "lumped together" into a hash class, access to each hash class is directly controlled by the synchronization mechanism, the state of each hash class is stored in fast memory at all times, and the offset in memory of the state of a hash class can be used as its name

3 A third approach is to use some type of associative memory, either partially or completely implemented in hardware Using this approach, one of the states (such as the initial state S_1) must be specified as a special *empty state* Given the name of a synchronization entity E, if no state for E is found in the associative memory, then E is by default in the empty state, and whenever the state of a synchronization entity becomes the empty state, it can be removed from the associative memory

Unfortunately, the above table-driven approach for implementing a fast general-purpose synchronization mechanism is impractical for other than the simplest synchronization protocols, or for more than a few processors when using a more complex protocol For example, consider using this approach for exclusive locks with queueing in an eight-processor system Then for the case in which one processor holds a lock and $n - 1$ processors are waiting, there are 8 processors that could hold the lock, 7 processors that could be queued after the owner, etc , for a total of $8!/(8-n)!$ states The total number of states would be

$$1 + 8!/7! + 8!/6! + \quad + 8!/0! = 109601,$$

and the transition and notification tables would each be of size

$$8 \times 2 \times 109601 = 1753616$$

Using several megabytes of ROM to store these tables is of course not impossible, but it is somewhat expensive, and the situation becomes much worse for more processors or for more complex protocols A solution to this problem is presented next

4. Reducing Table Size Using Symmetry

All of the example synchronization protocols of Section 2, and most synchronization protocols of practical interest, have the following feature the global synchronization mechanism "looks the same" to every processor, that is, there are no preferred processors In terms of the specification developed in the previous section using transition and notification functions, this property can be formally defined as follows

A synchronization protocol is *symmetric among the processors* if given any permutation of the processors π, there exists a permutation of the states Σ_π such that

$$T(\pi(P_i), R_j, \Sigma_\pi(S_k)) = \Sigma_\pi(T(P_i, R_j, S_k)),$$

and

$$N(\pi(P_i), R_j, \Sigma_\pi(S_k)) = \pi(N(P_i, R_j, S_k)),$$

where $\pi(\{x\})$ is defined as $\{\pi(x)\}$

In other words, given any re-ordering of processors in the specification, it is possible to re-order the states so that the new specification is identical to the original specification

It easy to show that given two permutations of processors π and π' and two permutations of states Σ_π and $\Sigma_{\pi'}$ that satisfy the above definition, the permutation of states $\Sigma_{\pi'} \bullet \Sigma_\pi$ (where " $\bullet$ " denotes function composition) satisfies the above definition for the permutation $\pi' \bullet \pi$ The symmetry property and this composition property can be used to collapse states in the following fashion For each permutation π of the processors, choose a permutation of states Σ_π to satisfy the above definition, in such a fashion that given any two permutations π and π', $\Sigma_{\pi' \bullet \pi} = \Sigma_{\pi'} \bullet \Sigma_\pi$ (this can always be done by selecting a set of generators for the group of permutations of processors, finding a permutation of states for each generator that satisfies the symmetry definition, and then using this set of state permutations to generate a state permutation for any processor permutation) Next, define an equivalence relation among states by $S_k \equiv S_{k'}$ if for any permutation π of the processors $S_k = \Sigma_\pi(S_{k'})$ Let there be c equivalence classes $C_1, C_2, \ldots, C_c$ For example, for the synchronization protocol used for Figure 1 in the previous section, there are three equivalence classes, $\{S_1\}$, $\{S_2, S_3\}$, and $\{S_4, S_5\}$ Next, choose one state $X(C_l)$ from each equivalence class Each of the c states $X(C_l)$ will be called a *canonical state*, and the full set of states S_k will be called the *actual states* Then any state S_k is in some equivalence class C_l, and can be represented as the pair $(X(C_l), \pi)$, where π is a permutation that makes S_k equivalent to $X(C_l)$, i e , $S_k = \Sigma_\pi(X(C_l))$ In other words, any actual state can be represented as a canonical state, processor permutation pair

Next, suppose it is desired to find $T(P_i, R_j, S_k)$ and $N(P_i, R_j, S_k)$, where the actual state S_k is represented as the canonical state, processor permutation pair $(X(C_l), \pi)$ Then from the definition of symmetry,

$$\begin{aligned} T(P_i, R_j, S_k) &= T(\pi(\pi^{-1}(P_i)), R_j, \Sigma_\pi(X(C_l))) \\ &= \Sigma_\pi(T(\pi^{-1}(P_i), R_j, X(C_l))), \end{aligned}$$

and

$$\begin{aligned} N(P_i, R_j, S_k) &= N(\pi(\pi^{-1}(P_i)), R_j, \Sigma_\pi(X(C_l))) \\ &= \pi(N(\pi^{-1}(P_i), R_j, X(C_l))) \end{aligned}$$

However, $T(\pi^{-1}(P_i), R_j, X(C_l))$ is not necessarily a canonical state Let the actual state $T(\pi^{-1}(P_i), R_j, X(C_l))$ be represented as the canonical state, processor permutation pair $(X(C_{l'}), \pi')$ Then

$$T(P_i, R_j, S_k) = \Sigma_\pi(\Sigma_{\pi'}(X(C_{l'}))) = \Sigma_{\pi \bullet \pi'}(X(C_{l'}))$$

This shows that $T(P_i, R_j, S_k)$ can be represented as the canonical state, processor permutation pair $(X(C_{l'}), \pi \bullet \pi')$ This provides the key for the reduction of the transition and notification tables in a table-driven approach Assuming that the states of synchronization entities are stored in a canonical state, processor permutation pair representation, instead of having entries in the tables for every (P_i, R_j, S_k) triple, it is only necessary to have entries for the canonical states Furthermore, for each entry in the transition table that gives a new canonical state number there must also be an associated processor permutation π', where π' is as defined above

In more detail, define transition, permutation, and notification tables T, Q, and N as follows

1 T[i,j,l] = l' if and only if $T(P_i, R_j, X(C_l)) \in C_{l'}$,
2 Q[i,j,l] is a vector of processor numbers of length p, and Q[i,j,l][i'] = i'' if and only if $\pi(P_{i'}) = P_{i''}$, where π is defined by $T(P_i, R_j, X(C_l)) = \Sigma_\pi(X(C_{l'}))$, and
3 N[i,j,l] is a bit-vector of length p, and N[i,j,l][i'] = 1 if and only if $P_{i'} \in N(P_i, R_j, X(C_l))$

Using these tables, the algorithm for processing a request R_j from processor P_i for a synchronization entity E is the following

1 Find the state of E, say (l, q), where l is an integer giving a canonical state number and q is a vector of processor numbers of length p representing a permutation of processors

	STRAIGHTFORWARD APPROACH		USING CANONICAL STATES	
	#states	table size	#states	table size
TOKEN PASSING	8	64	1	8
LLF PROTOCOL	1026	16416	10	160
XCL LOCK/QUEUES	109601	1753616	9	144
SHR/XCL LOCKING/ SER-PAR QUEUEING	7472808	179347392	511	12264

Figure 2 Table Size Reductions for the Eight-Processor Case

2 Set the new canonical state number l' to $T[q^{-1}[\imath], j, l]$, where $q^{-1}[\imath]$ is defined by $q[q^{-1}[\imath]] = \imath$

3 Compute the new processor permutation q' by setting $q'[\imath']$ to $q[Q[q^{-1}[\imath], j, l][\imath']]$ for $\imath' = 1,2, \quad ,p$

4 Store the new state of E as (l', q')

5 For $\imath' = 1,2, \quad ,p$, if $N[q^{-1}[\imath], j, l][\imath'] = 1$, then signal processor number $q[\imath']$ and transfer the name E and new state (l', q') to the processor

The same alternatives for storing and finding the states of synchronization entities as described in the previous section also apply when using the above approach based on canonical states In particular, when using an associative lookup mechanism, it will be necessary to identify one canonical state as a special empty state

It is worthwhile at this point to examine the reduction in table size that can be achieved when using the above approach Numbers of states and table sizes (for each table) are given in Figure 2 for the straightforward and canonical-state-based approaches for each of the example synchronization protocols of Section 2, and for an eight-processor system, under the following assumptions (1) for token passing there is only one request, "pass", (2) for the LLF type protocol and also for exclusive locks with queues there are only two requests, "lock" and "unlock", and (3) for shared/exclusive locks with series-parallel queues there are three requests, "lock-share", "lock-exclusive", and "unlock" (note that a "lock-share" request for a lock currently owned exclusively could be interpreted, if desired by the designer of the synchronization protocol, as a demotion request, and similarly an exclusive lock request for a lock currently owned in share mode could be interpreted as a promotion request) The extra canonical state for the LLF protocol as compared to exclusive locking is due to the "unlocked-pending" state described earlier The numbers of states in each case are derived using simple combinatorial techniques (with the help of a short APL program for straightforward shared/exclusive locking), and the derivations are omitted here

Finally, in order to use this approach in practice it is necessary to develop software tools to generate transition, permutation, and notification tables for the synchronization protocol that is to be used Since in practice it seems that equivalence classes of states can always easily be recognized by the designer of a synchronization protocol using the semantics of the states, it has been the author's experience that the development of such tools does not present significant problems

5. A Hardware Implementation

The algorithm developed for a canonical-state-based table-driven synchronization mechanism in the previous section can easily be implemented directly in hardware, since it is straightforward to implement the various steps of the algorithm using comparators, multiplexors, and de-multiplexors An example implementation is shown in Figures 3-6 For simplicity, the four-processor case has been used, and it is assumed that canonical states can be represented with six bits and request numbers with two bits, for a maximum of 64 canonical states and four requests

Figure 3 presents a block diagram of the mechanism In the following, it should be understood that the origin of all integers, including array indexes, should be rebased at 0 for the hardware version of the algorithm Parenthesized numbers in the figures indicate bit numbers, with bit 0 the low-order bit Referring to the algorithm of the previous section, the block labelled q^{-1} in Figure 3 finds $q^{-1}[\imath]$, and is shown in detail in Figure 4 The tables T, Q, and N are implemented as a 1024x18 ROM, as shown The block labelled $q \bullet Q$ finds $q[Q[\imath']]$ for $\imath' = 0,1,2,3$, and is shown in detail in Figure 5 Finally, there are four outputs from the block labelled N', one for each of the four processors These outputs indicate which of the four processors should be notified of the state transition, thus imple-

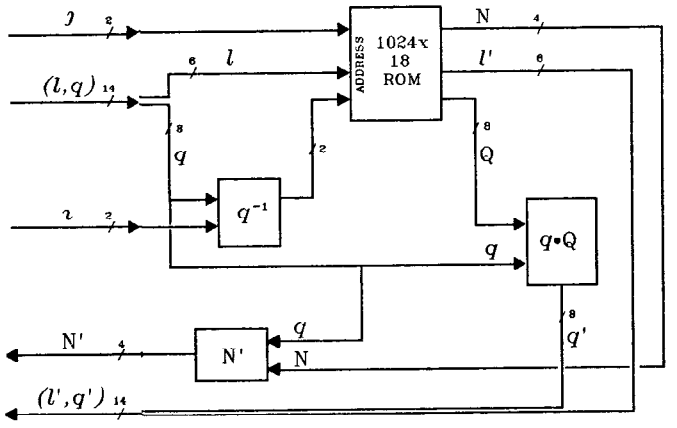

Figure 3 Block Diagram of Hardware Synchronization Mechanism

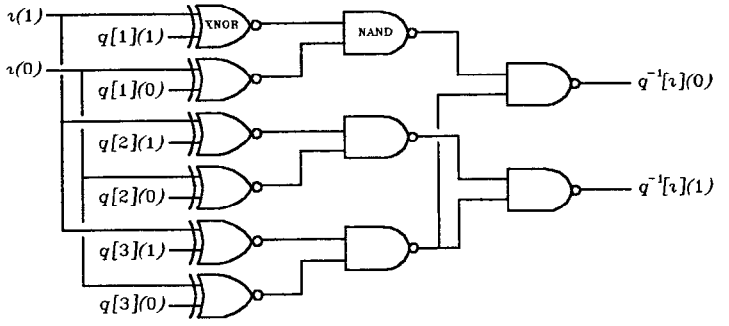

Figure 4 Circuit to Compute $q^{-1}[i]$

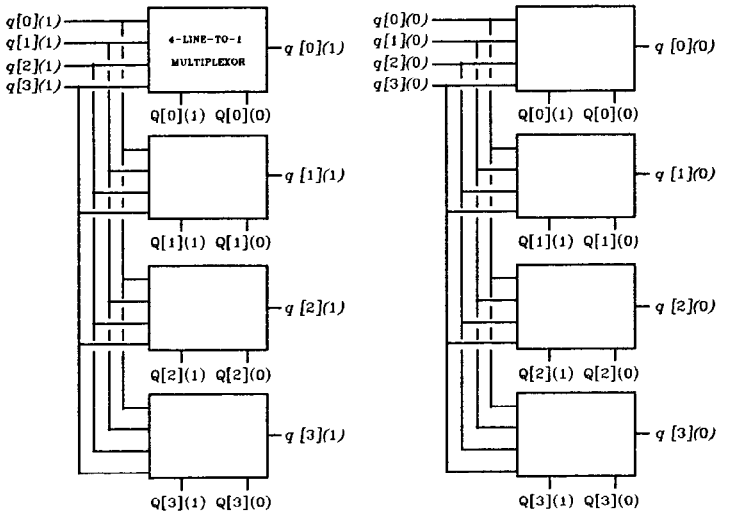

Figure 5 Circuit to Compute $q \bullet Q$

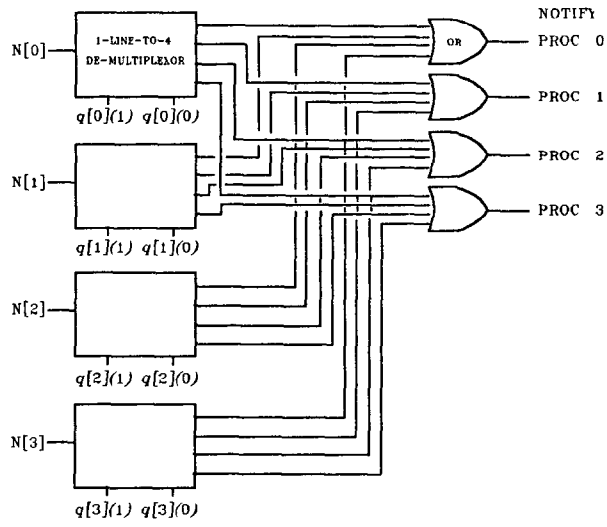

Figure 6 Circuit to Compute Set of Processors to Notify

menting step 5 of the algorithm This block is shown in detail in Figure 6

To use this mechanism in an actual system, it would be necessary to choose one of the alternatives described earlier for synchronization entity state memory, and to develop additional hardware to interface the state transition mechanism to this memory and to the processors in the system However, note that the time required to compute a state transition and the set of processors to be notified of the transition is extremely small, essentially consisting of one ROM access time and a fairly small number of gate delays

6. Conclusion

Another way to implement the canonical-state-based algorithm of Section 4 is to implement the algorithm in microcode This approach was used to develop a prototype general-purpose hardware synchronization mechanism for up to eight 370 processors using the 3880 model 11 storage control unit as the hardware base (see [IBM 81] for a short description of this control unit) This control unit, in addition to microcode control store, has available eight megabytes of RAM, normally used as a cache for pages on DASD For the synchronization mechanism prototype, though, all eight megabytes were used to store the states of synchronization entities Using eight-byte states, this provided storage for more than one million (2^{20}) synchronization entities

In addition to channel commands for processing requests against synchronization entities, a channel command was developed that allowed the tables for arbitrary synchronization protocols to be dynamically loaded into control store Also, various other channel commands included commands to read and write the states of synchronization entities directly, to disable or enable processor interfaces (for envisioned fault-tolerant applications), etc Some simple software tools for specifying and generating tables for various types of synchronization protocols were also developed

To date, this synchronization mechanism prototype has actually been used in only one experimental database system, using the current IMS multi-system data sharing protocol as a starting point (see [Strickland et al 82] for an introduction to IMS multi-system data sharing) IMS uses a distributed algorithm to implement a multi-system lock manager However, in order to reduce inter-system communication, each lockable resource is mapped into one of a number of hash classes, and a system is defined to have "interest" in a hash class if any transaction on the system currently holds or is waiting for a lock in the hash class By keeping track of which systems have unique interest in any hash class, the following optimization can be made if a system has unique interest in a hash class, then all lock requests that map to that hash class can be granted locally with no inter-system communication

Using hash classes as synchronization entities, a synchronization protocol was developed that allowed the control-unit-based synchronization mechanism to replace the distributed algorithm currently used by IMS for changing hash class states A description of the details of this experimental system is beyond the scope of this paper, other than to emphasize that the design and implementation of this synchronization protocol took

place *after* all microcode development was complete, and that no microcode changes were necessary all that was required was the generation and loading of the tables for this protocol Of course this is only one example, but the experience so far has been that a table-driven general-purpose synchronization mechanism, in addition to providing an extremely fast algorithm for computing state transitions and the set of processors to be notified for each transition, is quite valuable for decreasing the time needed for system development

Acknowledgments. John Christian provided significant aid in microcode development for the 3880-11-based synchronization mechanism prototype All IMS-related design and IMS modification for the experimental database system that used the prototype was done by George Wang

References

[Behman et al 79] Behman, S B , DeNatale, T A , and Shomler, R W Limited lock facility in a DASD control unit Tech report TR 02 859, IBM General Products Division, San Jose, California, October 1979

[Carey 83] Carey, M J Modelling and evaluation of database concurrency control algorithms Ph D Thesis (UCB/ERL 83/56), Electronics Research Laboratory, University of California, Berkeley, September 1983

[Gray 78] Gray, J Notes on database operating systems In *Notes in Computer Science 60 Operating Systems*, ed R Bayer, R M Graham, and G Seegmuller, Springer-Verlag, Berlin, 1978, 393-481

[IBM 81] Introduction to IBM 3880 storage control unit model 11 (GA32-0060), IBM General Products Division, Tucson, Arizona, September 1981

[Ries and Stonebraker 77] Ries, D R , and Stonebraker, M Effects of locking granularity in a database management system *ACM Trans Database Systems 2*, 3 (Sept 1977), 233-246

[Ries and Stonebraker 79] Ries, D R , and Stonebraker, M Locking granularity revisited *ACM Trans Database Systems 4*, 2 (June 1979), 210-227

[Siwiec 77] Siwiec, J E A high-performance DB/DC system *IBM Systems Journal 16*, 2 (1977), 169-195

[Strickland et al 82] Strickland, J P , Uhrowczik, P P , and Watts, V L IMS/VS An evolving system *IBM Systems Journal 21*, 4 (1982), 490-510

[Tay 84] Tay, Y C A mean value performance model for locking in databases Ph D Thesis (TR-04-84), Center for Research in Computing Technology, Harvard University, February 1984

[Thomasian and Ryu 83] Thomasian, A , and Ryu, I K A decomposition solution to the queueing network model of the centralized DBMS with static locking *Proc ACM SIGMETRICS Conf Measurement and Modeling of Computer Systems*, August 1983, 82-92