UCRL- 97663
PREPRINT

A NEW APPROACH TO EXCLUSIVE DATA ACCESS IN SHARED MEMORY MULTIPROCESSORS

Eric H. Jensen
Gary W. Hagensen
Jeffrey M. Broughton

This paper was prepared for ~~sumibttal~~ submittal to
The 15th Annual International Symposium on
Computer Architecture
Honolulu, Hawaii
May 30 - June 2, 1988

November 13, 1987

Lawrence Livermore National Laboratory

A New Approach to Exclusive Data Access in Shared Memory Multiprocessors

Eric H. Jensen

Gary W. Hagensen†

Jeffrey M. Broughton

S-1 Project

Lawrence Livermore National Laboratory

Livermore, California 94550

Abstract

Optimistic synchronization, a new mechanism to provide exclusive data access in a shared memory multiprocessor has been developed. This mechanism has been implemented in the S-1 AAP multiprocessor. Optimistic synchronization is a generalization of the traditional *conditional_store* instruction. The latency required to determine the ''equality'' condition in the conditional_store is exposed to the compiler. This simplifies pipeline control for RISC-like designs and allows the latency inherent in a complex coherency protocol or a complex processor interconnect to be absorbed by concurrent instruction execution. In addition, optimistic synchronization eliminates the need for auxiliary lock variables in some cases.

Introduction

A shared memory multiprocessor system must provide exclusive data access mechanisms. These mechanisms are used to develop synchronization primitives that can be used by concurrent processes to

†This author is now at MIPS Computer Systems, Sunnyvale, CA.

enforce order and/or consistency when accessing shared data The conventional uniprocessor approach of masking interrupts is not sufficient to ensure exclusive shared data access in a multiprocessor environment. Previous multiprocessor approaches involve locking some global hardware resource. These approaches fall primarily into two classes. The first provides instructions which explicitly lock or unlock a hardware resource. The Am29000 *loadl* and *storel* instructions are an example.[1,2] The second provides indivisible operations such as *test-and-set*[3] or *conditional_store*[4,5] (also called *compare-and-swap*).[6,7] These lock a hardware resource only for the duration of their execution.

The S-1 Advanced Architecture Processor (AAP) introduces a new approach to exclusive access called optimistic synchronization. In its simplest form, an interruptable sequence of instructions attempts an "atomic" modification of a variable under the presumption of exclusive access. The operation completes successfully only if the exclusive access presumption can be verified and no interrupts occur.

This approach is analogous to optimistic concurrency control[8] in database transactions and arises from a generalization of the semantics of the traditional conditional_store instruction.

Conditional_store

The conditional_store instruction typically executes the following code fragment atomically:[5]

```
sample ← MemFetch[ptr];
if sample = old then MemStore[ptr,new];
```

New replaces *old* if and only if the value referenced by *ptr* is the same as *old*. A condition code is set[7] or the value of *sample* is left in a register[6] so that the success or failure of the conditional can be determined by a subsequent operation. This permits a function to read a location, compute a new value for that location and store the new value if the contents of the location have not changed. If the contents of the location have changed then the new value will be discarded and the computation can be repeated. The function presumes exclusive access to the memory location while it is computing a new value.

This use of conditional_store is based on the observation that *the ability to detect the violation of exclusive access to a memory location is sufficient to obtain exclusive access provided that the rate of concurrent access is low.* Concurrent access is defined as any overlap during the time that more than one processor is presuming exclusive access to the same memory location.

Optimistic Synchronization

Optimistic synchronization is accessed through the *sync_load* and *sync_store* instructions. In addition, the S-1 AAP provides a third instruction, *sync_clear*, which is a degenerate form of sync_store. A processor is in a ***presumed exclusive access region*** during the interval between the execution of a sync_load instruction and the execution of either a sync_store or a sync_clear instruction. A processor will trap if two sync_loads are executed without an intervening sync_store or sync_clear. This trap is due to restrictions imposed by the implementation.

The **sync_load** instruction computes *xa-address* (an effective address), declares that the executing processor is presuming exclusive access to a synchronization block containing xa-address and loads the addressed data into a general register. The synchronization block may be larger than the data item referenced by xa-address. Loading the data item referenced by xa-address is merely a convenient side-effect; a separate load could be executed after the sync_load instruction.

The **sync_store** instruction:

1) *conditionally stores an item of data to memory.* Only if the presumption of exclusive access to xa-address is valid does the store occur, in which case the sync_store is said to be successful. *The effective address computed by the sync_store instruction may be distinct from xa-address.* If this address is different than xa-address, it can not be the xa-address of a concurrently executed sync_load.

2) *alters program flow based on whether or not the sync_store is successful.* In the S-1 AAP, the

instruction following the sync_store instruction is skipped if the sync_store is successful. Alternatively a condition code or general register could be set for use by a conditional branch. The ability to alter program flow is necessary in order to retry the sync_load/sync_store sequence when the exclusive access presumption is invalid.

3) *terminates the presumed exclusive access region*

If a sync_store executes outside a presumed exclusive access region (no prior matching sync_load executed), it executes like an unsuccessful sync_store.

Determining the validity of the exclusive access presumption and storing data on the basis of that determination must be performed indivisibly. An instruction that only determines the validity of the exclusive access presumption is useless. If an exception occurs between the validity test and the store, the test may be invalid when the store executes. Thus the sync_store instruction is like a conditional_store, except that the equality computation and possibly the store address are different.

The **sync_clear** instruction terminates a presumed exclusive access region but does not store any data or alter program flow. It is a semantic convenience used when a program decides to abort a presumed exclusive access region. If a sync_clear executes outside a presumed exclusive access region (no prior matching sync_load executed), it has no effect.

Coded Example

The following code sequence illustrates one use of optimistic synchronization. The variable *lock* is locked when non-zero. The jump_q instruction executes the following instruction only if the branch condition is true.

```
            IF (lock = 0) THEN lock := ProcessID          /* indivisible operation */
            ELSE GOTO LockHeld;

Retry:      sync_load       R10, lock                     ; declare x-access presumption on lock
            jump_q          .neq (R10, 0), LockHeld       ; test for zero
                sync_clear                                ; lock non-zero, abort x-access
            load            R10, ProcessID                ; prepare to update lock
            sync_store      R10, lock                     ; update lock if x-access presum. true
                goto        Retry                         ; try the update again
MyLock:
```

Assume *lock* has a non-zero value. The sync_load will load that value into register 10 and declare that this processor is presuming exclusive access to *lock* (xa-address). The jump_q instruction will detect that *lock* has a non-zero value, the sync_clear instruction will be executed and control will transfer to the label LockHeld elsewhere in the program. The sync_clear will declare that exclusive access to xa-address (indicated implicitly) is no longer presumed, terminating the presumed exclusive access region.

Now assume that *lock* has a zero value. The sync_load instruction will load that value into register 10 and declare that this processor is presuming exclusive access to *lock* (xa-address). The condition in the jump_q instruction will be false and *ProcessID* will be loaded. The sync_store instruction is executed next and may stall waiting for the verification of exclusive access to xa-address. If exclusive access to xa-address has been verified, then *lock* is updated to *ProcessID* and execution continues at label MyLock. If exclusive access to xa-address can not be verified then *lock* will not be updated, the goto will be executed transferring control to label Retry, and the sync_load instruction will be reissued. The sync_store will declare that exclusive access to xa-address is no longer presumed. terminating the presumed exclusive access region.

The use of sync_load and sync_store instead of a conditional_store does not create a longer code sequence. If conditional_store were used it would replace the sync_store, and the sync_load would become a normal load instruction.

The programmer or compiler must use the sync_load and sync_store instructions consistently, as they are incompatible with the normal load and store instructions.

Deadlock Avoidance

Allowing a non-privileged process explicitly to lock a hardware resource for an unlimited length of time can have grave consequences. Problems can arise when an external event occurs that suspends a non-privileged process holding an active lock. A process that is required to cope with the external event may need to access the locked resource. For this reason, actions or instructions that lock or unlock a hardware resource (e.g. masking interrupts) have traditionally been placed under the strict control of the operating system kernel.

Because limited sync_load and sync_store hardware resources may be used by an unlimited number of active processes, the processor must execute a sync_clear instruction when switching contexts. This resets the processor state for optimistic synchronization so that the new or resumed context can execute safely. When a process that was suspended in a presumed exclusive access region is resumed, its first execution of sync_store (to terminate the suspended presumed exclusive access region) will fail. However, when the optimistic synchronization code is re-executed without an intervening context switch, it will succeed if the exclusive access presumption is valid. *It is this property that allows non-privileged processes direct access to optimistic synchronization.*

It is unsafe to invoke code which uses optimistic synchronization from within an exclusive access region. This will result in a trap (two consecutive sync_loads) or a livelock[8] condition. A livelock condition occurs when the invoked code executes a sync_clear resetting the processor state for a separate use of optimistic synchronization. The sync_store in the invoking code will always fail. Most kernel calls within a presumed exclusive access region will create a livelock condition because of their use of optimistic synchronization.

Locking Data Instead of Locks

Typically, lock variables are associated with shared data. When a lock variable is obtained through a indivisible operation (e.g. test-and-set) exclusive access to the associated shared data is established by convention. *The permission to sync_store to a different address than xa-address allows the elimination of lock variables in some cases.* If only one data item is to be computed from a set of data, a convention to sync_load one data item to define exclusive access over the entire data set could be established. The sync_store will install the new computed value if the exclusive access presumption is valid. If the sync_store references a non-resident memory page, it will fail because of the necessary context switch. When the memory page is resident and the optimistic synchronization code is re-executed, the sync_store will succeed if the exclusive access presumption is valid.

Violating Exclusive Access

Conditional_store uses a straightforward equality test to determine when exclusive access has been violated. It is more convenient for optimistic synchronization to detect another processor's intention to write xa-address.

This section will address the problem of detecting an intention to write a memory location when there are multiple copies of that location. This problem is evident in the presence of caches, each of which may contain a copy of a shared memory location. In the absence of multiple copies, it is straightforward to detect a write to a memory location.

Most cache coherency protocols rely on a notion of write ownership to insure exclusive access. The protocols range from invalidating other cache copies when a write occurs, to broadcasting updates after *sole write access* has been obtained.[9] When using these protocols, the detection of an invalidation or loss of sole write access during the interval between the execution of sync_load and the execution of sync_store can identify an intention to write.

The S-1 AAP cache coherency protocol does not have a notion of sole write access.[10] Instead, writes are propagated to every interested cache even if they occur simultaneously. A processor's cache is considered interested in a write only if it contains a copy of the written memory location (propagating writes to non-interested caches only wastes cache cycles). The writer knows when all interested caches have been updated.

When a sync_load is executed, a query is sent to all interested caches. The query asks *are-you-presuming-exclusive-access-to-this-synchronization-block?* and a response of *yes* or *no* is returned. A processor only responds *yes* if it is in a presumed exclusive access region that refers to the same synchronization block. Any *yes* response will invalidate the exclusive access presumption and a subsequent sync_store will fail. If any query is outstanding (the S-1 AAP only sends out one query that is propagated from cache to cache) when the sync_store executes, the sync_store will stall until every response is received. If there are no other interested caches, a query is not necessary. The performance gain due to this situation is suggested by the Dragon processor experience.[5]

The querying processor must also service queries from other processors. It responds with *yes* to those queries that refer to the same synchronization block until the exclusive access presumption is known to be false or the presumed exclusive access region is terminated. When two or more processors are simultaneously querying each other about the same synchronization block, their exclusive access presumptions are invalidated. System performance may be enhanced if an arbitrary priority scheme is applied when this situation is detected so that one processor is permitted exclusive access. The S-1 AAP uses the processor ID to resolve this situation.

Writes to xa-address also invalidate the exclusive access presumption. This can occur when a sync_store to xa-address is propagating to interested caches while a sync_load from xa-address is executed on a different processor. The location at xa-address will be written (invalidating the copy in a general register) while the query will get a *no* response. The requirement to detect writes can be dropped if messages between interested caches maintain strict relative ordering and a processor responds *yes* to the appropriate queries until a successful sync_store updates all interested caches.

Simpler Pipelines

The conditional_store has a hidden latency when executed in a sole write access environment. The conditional_store may have to stall while sole write access is obtained. The equivalent latency in optimistic synchronization is due to the sync_load query. However, *this latency can be partially, and sometimes totally, overlapped with the execution of instructions in the presumed exclusive access region.*

The conditional_store can add complexity to a RISC-like pipeline because of the necessary read-modify-write cycle. This is complicated by the internal branch imposed by the conditional *sample = old.* Optimistic synchronization separates the read and write into two instructions and allows the computation of the conditional to be independent of the primary pipeline.

Granularity

Making the synchronization blocks larger than the data referenced by xa-address is equivalent to hashing the units of exclusive access on the high-order bits of xa-address. This will cause collisions when concurrent sync_load instructions reference different items of data in the same synchronization block, although the collisions may have little or no affect on system performance. The choice of synchronization block size is left to a particular implementation. If the temporal concurrency of presumed exclusive access regions is expected to be very low, a single global synchronization block could be provided.

The global address space for most machines is the physical address space. If synchronization blocks are determined by the high-order bits of xa-address, a virtual to physical address translation of xa-address is required to resolve sync_load queries. Because the S-1 AAP has physically addressed caches, this does not represent an additional cost.

For systems in which virtual to physical address translation for sync_loads is difficult, an alternative hashing scheme would be to hash synchronization blocks on the low-order bits of xa-address. If the number of low-order bits used is equal to or less than the number of bits needed to represent a memory page, a virtual to physical address translation can be avoided. This will result in a smaller number of synchronization blocks, but references will be more evenly distributed. This approach can be used when the cache hardware is inaccessible or inappropriate. Providing separate dedicated silicon to manage synchronization blocks would allow the use of existing processors. The Sequent SLIC[11] chip suggests the viability of this approach.

The choice of synchronization block size is equally applicable to both conditional storing and optimistic synchronization.

Conclusion

Optimistic synchronization is a generalization of the *conditional_store* instruction. The latency required to determine the "equality" condition in the conditional_store is exposed to the compiler. This simplifies pipeline control for RISC-like designs and allows the latency inherent in a complex coherency protocol or a complex processor interconnect to be absorbed by concurrent instruction execution. In addition, optimistic synchronization eliminates the need for auxiliary lock variables in some cases.

Acknowledgement

Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract number W–7405–ENG–48 with support from the Office of Naval Technology.

References

1. Brian Case, "Pipelined Processor Pushes Performance," *ESD*, Digital Design Publishing Corp., March 1987.
2. *Am29000 Streamlined Instruction Processor, Advance Information* 09075A, Advanced Micro Devices, February 1987.
3. *IBM System/360 Principles of Operation*, pp. 74-75, IBM Systems Development Division, November 1970.

4. L. C. Widdoes, ''S-1 Multiprocessor Architecture,'' 1979 Annual Report — The S-1 Project, Volume 1: Architecture, Lawrence Livermore National Laboratory Technical Report UCID 18619, 1979.

5. Russel R. Atkinson, Edward M. McCreight, ''The Dragon Processor,'' *ASPLOS II Proceedings*, no. 556870, pp. 65-69, ACM, October 1987.

6. Motorola Inc., *MC68020 32-Bit Microprocessor Users Manual*, pp. B-54, Prentice Hall, 1985.

7. Richard P. Case, Andris Padegs, ''Architecture of the IBM System 370,'' *Comm. ACM*, vol. 21, no. 1, pp. 73-96, January 1978.

8. Jeffery D. Ullman, *Principles of Database Systems, 2nd ed.*, pp. 400-405,439-443, Computer Science Press, 1982.

9. Philip Bitar and Alvin M. Despain, ''Multiprocessor Cache Synchronization — Issues, Innovations, Evolution,'' *Proceedings, 13th Annual Symposium on Computer Architecture*, pp. 424-433, June 1986.

10. S-1 Project, *Cache Coherency on the S-1 AAP*, Lawrence Livermore National Laboratory, November 1987.

11. Bob Beck, Bob Kasten, and Shreekant Thakkar, ''VLSI Assist For A Multiprocessor,'' *ASPLOS II Proceedings*, no. 556870, pp. 10-20, ACM, October 1987.