Abstract—This goal of this project is to implement fine-grained synchronization on a Chip Multiprocessor. Synchronization of multiple threads is an important aspect of parallel programs. Over the years researchers have proposed several algorithms and hardware implementations that attempt to understand the trade-offs in terms of latency, bandwidth, implementation complexity etc; nonetheless synchronization remains a significant concern for parallel applications. This study shows that with simple hardware implementation the cost of synchronizing multiple threads can be significantly reduced.

I. INTRODUCTION

Synchronization mechanisms (locks) have to be applied to shared data structures to ensure serialized accesses, leading to a potential bottleneck if threads spends significant amount of time in acquiring exclusive permissions to get the access to shared data structure. The original idea for fine-grained hardware based synchronization was proposed for simultaneous multithreading processor (SMT) as threads in those processors compete for fetch and execution resources each cycle. The threads that consume any shared resources without making forward progress can potentially impede forward progress of other threads. Lockbox is a simple hardware mechanism that enables the transfer of memory-based locks between threads on the same processor in just few cycles. In this implementation execution of program instructions is halted while the program is waiting for certain release event to occur on different thread. The goals of this synchronization scheme are high performance, which implies both high throughput and low latency, resource conservative, as spin locks consume processor resources while waiting for a lock, stalling those threads till any the release of shared resource will free up resources, deadlock free as the threads are waiting for a release instruction to execute on other thread forward progress has to be guaranteed.

Previous similar work in the area of managing synchronization using hardware mechanisms includes QOSB that introduced the idea of queuing requests for locks in hardware. Followed by MCS proposal that implemented the idea in software. QOLB (Queue on Lock Bit), I-QOLB with extra hardware support and support for speculating critical section in a parallel application were proposed.

The rest of the report is organized as follows. Algorithm for identifying and tracking lock operations is presented in Section II. Details of hardware implementation are presented in Section III. Learning’s and issues are discussed in Section IV. Some partial results and conclusion are summarized in Section V.
rarely in cases of I/O operations or instructions not modeled in Opal, then opal detects discrepancy and roll back to recover. Details of Lock identification algorithm are as follows. As SPARCv9 does not provide acquire and release instructions so to infer these instructions dynamic instruction stream is examined. To achieve this lock address table is maintained to track all memory operations that have been previously accessed by an atomic instruction. Every atomic instruction looks up in this table and makes an entry if an address is not already present. Each table entry tracks the status of the lock. Acquire and release operations are inferred based on the lock status maintained in the table and by the values read and written by the memory instructions to these addresses. For LDSTUB based locks following algorithm is used for acquire and release identification:

- An atomic operation writing a non-zero value (0xFF) and reading a zero is considered a successful acquire. The thread Id of this owner is marked to keep track of later release operation.
- An atomic operation that reads non-zero value while storing a non-zero is considered a failed acquire attempt. A contention is marked with an entry in the global lockbox table for this entry and the thread goes to sleep.
- Normal store operations to the similar physical address as previous atomic operation are considered as silent release operations and are used as events to wake up any sleeping thread which have an entry marked in the global lockbox table.

The algorithm fairly tracks the mutex lock accesses that contain a value of zero at the initialization time. Due to time restrictions the other classes of synchronization e.g. non-mutex are not considered in this implementation. Three hardware primitives of Solaris ISA are considered for identifying mutex locks ldstub, casxa, swap and their variants. Ldstub (load store unsigned byte), implements test&set semantics and store 0xFF to the memory location, while returning the previous contents. Code snippet used for testing “lock acquire and release” using Ldstub is as shown below in Figure 1:

```c
void acquire_lock () {
    while (1) {
        //Successful acquire operation returns 0xff
        if (trylock (&lock_var)) {
            return;
        }
        trylock: ldstab [o0+3], %o0
        ret
    }
}
```

Figure 1: Synchronization test

Casxa, casa (compare&swap) and swap instructions are also tracked for mutex locks. Used mcs_acquire (swap) and mcs_release (casa) functions for mutual exclusion. Algorithm to identify mutex locks implemented using compare and swaps and swap instructions.

- Atomic operations writing a value different from the original value is considered a successful lock acquire operation.
- If the value written back is the same original value then this is considered as a failure. A contention is marked for this entry and the thread goes to sleep.
- Every store (even non-atomic) to the same address location is tracked. And the store from the thread acquiring a lock is marked as a release operation and is used to wake-up the sleeping thread.

Algorithms discussed above are not capable to capturing synchronization behavior of barrier, reader-writer locks. Identification of those complex synchronization schemes is difficult to observe by only observing instructions at the hardware level. Also, the mutual exclusion behavior in application cannot be mapped directly to the hardware level. e.g. pthread_mutex_locks(&lock) acquire three locks, to guard single critical section. This will be observed as a couple of critical sections in hardware level. In the original paper the mechanism of lock identification was proposed for Alpha architecture. The Alpha ISA provides explicit acquire, release, load lock (ldl_l) and store conditional (stl_c) instructions which makes it easier to identify the pair of acquire and release operation. Also, the implicit memory barriers were assumed around these synchronization instructions. Synchronization test is also written with membarrier instructions both before and after the implicit acquire and release operations.

The capability to model the latency for different categories (phases) is also measured: acquire latency, release latency, transfer latency. Acquire latency for a successful uncontended lock is simply the time to retire the instruction including cache miss. For contended lock, a time is calculated from the
beginning of first unsuccessful acquire until the end of successful acquire (between this time the thread gets halted).

III. LockBox Hardware Implementation

When a thread fails to acquire a lock, the lock instruction is stored in the thread’s lock box entry, and entry is marked NULL if it doesn’t contain any valid information. When an entry is created in lockbox a signal is sent to the fetch unit to stall fetching any future instructions. Lockbox entry is indexed using processor id or every processor holds a single entry in lockbox address table. When another thread releases the lock, hardware performs an associative comparison of the released lock address against the lock-box entries. On finding a blocked thread, the hardware allows the blocked thread to resume normal execution starting from the value stored in program counter and invalidates the blocked thread’s lock-box entry.

In the original implementation for Alpha ISA the “acquire” instruction is restartable as it never commits if it doesn’t succeed, a thread that is context-switched out of the processor while blocked for a lock always restart with the program counter pointing to the acquire or earlier. This is different for the implementation for SPARCv9 processor because the instruction failing to acquire critical section access retires normally storing either same old value to the memory in case of casxa, swap, ldstub. Flushing a blocked thread from the instruction queue is critical in preventing deadlock. Similar hardware mechanism was presented in patent US patent 6,493,741 [2] where execution of a halted instruction stream is resumed by an observation of an identified quiesce event or upon an expiration of an timer which was set upon execution of a quiesce instruction for which some other thread is waiting (if, watch flag is set).

Following changes were made in the simulator. Each retiring instruction is observed and based on the type of instruction (“atomic or normal store”), and physical address per processor lock table entry is allocated in this table (if it’s not there already). Entry in this table is also used for collecting per processor lock statistics (some logic here is inherited from the base line simulator) implemented and for tracking the timing information for different phases of execution: lock free section, lock contention section, critical section. Different phases of atomic transaction are tracked using this table. These phases are (1) write failed which can happen in case of casxa instruction if the comparison operation is not successful. (2) Contention, if the write of an atomic operation is successful and any non-zero value is written in memory, but the location is held by some other processor, this is feasible with swap instructions which can be used for acquiring lock operation and can write any data value, in case of contention swap instruction will retire successfully but a later conditional branch will take care of reissuing the swap instruction. To detect contention within a single thread a simple flag is used, and to detect contention among multiple threads an entry is created in global lock-box table. Global Lock Box table holds single entry per processor, assuming there can be only single atomic “acquire operation” waiting for a corresponding “release operation” in s single user thread context. (3.) Release operation, which is tracked at two places at the retirement time of an atomic operation or at the retirement time of a store operation. As the release operation generally is issued as a normal store writing zero value to the memory location. Upon every release global lock box table is CAM (content address match) or associatively searched for an entry with similar physical address. Entry at thread’s index location (because of previous acquire) is deleted at this time and for a match with different entry in the table a signal to resume its execution is sent. If there are multiple hits in the global lock box table then the thread entry with minimum fetch time is selected for wake-up.

Opal runs ahead of Simics functional simulation by fetching, decoding, dynamically scheduling, executing and speculatively accessing the memory model (ruby). At the time of retirement opal instructs the Simics processor to advance its processor states. The MAX_RETIRE flag is used to determine the retire count per cycle.

IV. Learnings and Issues Debugged

During the course of this project I developed good understanding of the OOO processor modeling. Opal is a timing first simulation model. The processor configuration selected for simulations and collecting statistics is as follows:

- FETCH_STAGES: 3,
- DECODE_STAGES: 4,
- RETIRE_STAGES: 3,
- MAX_FETCH: 4,
- MAX_DECOD: 4,
- MAX_DISPATCH: 4,
MAX_EXECUTE: 4,  
MAX RETIRE: 4  //experimented with several varied configurations.

The pipelining at each stage is implemented by creating a state machine and by incrementing the state each cycle. For example in execution unit based on the resource availability the instruction progresses from Wait_for_4_resources t Wait_for_1_resource followed by LSQ_wait_stage for bypassing data from the pending from the LSQ. two separate stages early_store_stage and early_atomic_stage which are implemented to take care of situations when the permissions to call retire are not available because the value to write is not ready but the operation got its memory value from LSQ or from a prefetch issued to the memory subsystem. Opal doesn’t model hardware multithreading and is heavily tied to SPARC. Opal models sequential consistency model.

[1] [Acquire Lock] [Thread Id is 1]
Ldstub: 0x21313 data:0x1
[0] [Acquire Lock] [Thread Id is 0]
Ldstub: 0x300022650a0 data:0x1
[1] [Acquire Lock] [Thread Id is 1]
Ldstub: 0xff2be04 data:0x1
[0] [Acquire Lock] [Thread Id is 0]
casxa: 0x30002265180 data:0x1
[0] [Lock Write using normal store Thread ID 0]
stx: 0x30002265180 0x1039530 STORE
[0] [Release Locks using normal store] [Thread: 0 ]
stx: 0x30002265180 data:0x1
[0] [Acquire Lock] [Thread Id is 0]
Ldstub: 0xff1c928c data:0x1
[0] [Write Failed] [Waiting Thread Id is 1]
Ldstub: 0xff1c130c data:0x0
[0] [Acquire Lock] [Thread Id is 0]
Ldstub: 0xff1c130c data:0x1
[0] [Release Lock] [Thread: 0 ]
swap: 0xff1c928c data:0x0
[0] [Release Lock] [Thread: 0 ]
swap: 0xff1c130c data:0x0
[0] [Acquire Lock] [Thread Id is 0]
Ldstub: 0x21374 data:0x1
[0] [Acquire Lock] [Thread Id is 0]
Ldstub: 0xff1c130c data:0x1
[0] [Release Lock] [Thread: 0 ]
swap: 0xff1c130c data:0x0
[1] [Acquire Lock] [Thread Id is 1]
casxa: 0x30003526028 data:0x1
[1] [Lock Write using normal store Thread ID 1]
stx: 0x30003526028 0x1039530 STORE
[1] [Release Locks using normal store] [Thread: 1 ]
stx: 0x30003526028 data:0x1
[0] [Acquire Lock] [Thread Id is 0]
casxa: 0x30001ead5d8 data:0x1
[0] [Acquire Lock] [Thread Id is 0]
casxa: 0x30001ead5d8 data:0x1
[0] [Acquire Lock] [Thread Id is 0]
Ldstub: 0x14ed9a0 data:0x1
[1] [Acquire Lock] [Thread Id is 1]
casxa: 0x30001ead5d8 data:0x1
[1] [Lock Write using normal store Thread ID 1]
stx: 0x30001ead5d8 0x1039530 STORE
[1] [Release Locks using normal store] [Thread: 1 ]

Figure 2: Trace from a [Lock Contention] Micro-benchmark

Let’s go through the trace file generated for detection of the potential acquire and release operations in the code stream. Below trace file is generated for 2 Processor configuration by running full system simulation model with Ruby and Opal. Ruby doesn’t model the data interface so the collection of write data values is interfaced from SIMICS at the time of instruction retirement. The benchmark was written for generating lock contention and using spin locks. The motivation behind showing this trace is measuring the correctness of implemented algorithm. From the trace it can be seen that Thread 0 first pairs of ldstub (load-store) issued as a single atomic operation for the cache lookup but at the time of retirement if the value of ldstub is returned as 0x1 that operation is considered as a beginning of acquire operation [which is correctly decoded here, in the stream], the next operation which is swap which writes 0x0 to that location is correctly decoded as a release for the previous lock. This small trace does not show the cases for address contention, which are also correctly decoded, e.g. one contention case is shown for 0x21313 which is acquired by thread 1 and thread 0 is contending for the same address (last instruction in the trace). The issues faced during the implementation are mainly related to the correct understanding of the SPARC assembly to know which assembly instruction corresponds to which high level instruction. The second issue was generating lock contention for a particular address or location of interest, solved this issue by inserting the delay loops of varied lengths after acquiring the lock and sometimes after/before releasing the lock so that the time taken in the critical section can be enlarged and the other thread can also reach at the same location. Small assembly loops using different SPARCV9 atomic instructions were written for implementing and testing spin locks.

Opal depends heavily on the SIMICS interface and with each release of SIMICS the Opal interface is not updated. So there were several issues resolved/pending related to getting real information about context switches, reading actual physical memory via SIMICS interface for data modeling. Due to time
constraint I couldn’t get memory tracing utility to work because of the modifications in the SIMICS Opal API’s used for tracing.

To mention some of the key learnings of the project, understood the OOO processor timing model, developed key insight for the behavior of spin locks, how the assembly code varies with small variations in the timings and order of instruction executions. The complete analysis of lock behavior in a real application is certainly a challenging task as that would depend on considering IO accesses to the acquired address location, context switches of threads, deadlock avoidance, and proper safety mechanism for time-out are necessary. One of the issues I debugged was related to the time-out, as the thread which acquired a lock hasn’t released it within the time set for difference in consecutive retirements. Also, careful thought has to be performed to ensure the consistency model violations are prevented. As Opal issues and executes instructions speculatively, sometimes multiple pending acquire operations at retire time from the same thread were seen. As the global lock box is designed to hold a single entry per processor, these scenarios were difficult to trace unless major modifications are made in the simulator so as to stop the later operation “at the execution stage”. I tried hacking the Load Store Queue for implementing this behavior but couldn’t finish the work completely. In addition to the complexities I just mentioned, while running the big benchmarks (SPLASH) the complex interactions between the high level software, locking libraries, operating system intervention were creating all sort of troubles. In future, I will be interested in working on modeling the synthetic benchmarks with fine grain parallelism to measure the performance analysis of the work on lock-box implementation.

V. REFERENCES