**Paper**: Conditional Memory Ordering by Christoph von Praun et al.

**Presenter**: Renwei Yu

The paper proposes a new multiprocessor synchronization algorithm called *Conditional Memory Ordering (CMO)*, with the objective of curtailing memory ordering overheads thereby delivering improved performance. Given below is a critique of the paper that covers aspects that favor or go against this purpose:

**Points that Favor:**

- **Throughput Improvement**: The methodology taps performance improvement by identifying the dynamic lock/unlock operations for which memory ordering is unnecessary and speculatively omitting the associated memory ordering instructions (release synchronization). Furthermore, the hardware implementation of inter-processor communication makes the improvement effective relative to the Speedup results obtained through evaluation using S-CMO software prototype.

- **Handling of Thread Migration and Context Switches**: Context switch and thread migration cannot take place when program execution is atomic. However, a switch can take place when the program being executed is in its critical section (enclosed by the lock operations). Even in such a case, CMO sees that a 'stale' release number does not breach memory synchronization. A given example in the paper talks about such a switch between the synch-release and unlock statements.

- **Backward Compatibility**: If implemented, CMO must include support for an instruction (*synch有条件*) to provide a mechanism with which one processor may initiate memory ordering at another processor. Since, this is the only architecturally visible aspect of CMO, it can be implied as backward compatible with respect to existing software.

**Points that go against:**

- **Platform dependence**: The solution talks about memory ordering only in a shared memory architecture. However, it is not a common case solution that can be implemented on any shared memory model. The solution assumes the existence of explicit instruction support for memory ordering like in the case of PowerPC 4 and PowerPC 5 (*isynch* and *lwsynch*).

- **Overheads**: The improved memory ordering using CMO, reduces CPU time wasted in handling redundancy. However, it increases overheads in the form of hardware changes and memory latency (particularly the cost of *synch远程*) that involves maintaining mirrored copies of n-sized *release count vectors* across n-cores. The overall speedup is thus dependent upon how frequent these redundancies appear. Eg. WAS systems show a lag in performance which can be attributed to this.

- **Scalability**: CMO, in general, is effective for configurations that have a large number of threads than processors. Adding to this, the inter-processor communication required for initiating remote memory ordering proves to be a hindrance to scalability of a system implementing CMO.

- **Reduced Scope of Evaluation**: The authors have selected Java based benchmarks for the evaluation of CMO. This is good due to the frequent use of lock operations in Java. However, a better picture could have been conclusively arrived at by widening the scope of test environments. Also, the work in the paper restricts itself to optimizing locks at the JVM level and does not explore implementing CMO in kernel lock routines.
• **Very Little details on Adaptive S-CMO**: The authors refer to the use of an Adaptive S-CMO protocol and show favorable results as part of evaluating CMO using software prototypes. However, they reveal very less details on its implementation, especially the logic involved in switching between S-CMO and the standard synchronization protocol.

**References**:
1. [http://en.wikipedia.org/wiki/Memory_ordering](http://en.wikipedia.org/wiki/Memory_ordering)

***