The summary for the presentation on “Conditional Memory Ordering” presented by Mr Renwei Yu is given below.

The paper focuses on Memory reordering operations in multiprocessors when the cores are out of order type. In such scenarios, memory barrier instructions are added by the programmer or compiler to enable a consistent view of the memory by all processors, as required by the program order. However in certain cases, these operations are extraneous, and can be avoided to achieve better throughput. Thus the paper proposes Conditional Memory Ordering to avoid unnecessary memory ordering operations. Common cases of such unwanted locking are when a variable is accessed by only one thread running on a processor core or, when the locality of accesses of a variable is confined to a single thread. Now we summarize the positive and negative points about Conditional Memory Ordering.

Pros:

1) Their experimentation is conducted on both high cost and low cost memory ordering implementations and it produces a speedup of up to 11% in high and 3% in low MOI systems.
2) Simulations have been carried out over an extensive set of SPEC benchmarks and multi-threaded Java Benchmarks.
3) 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.
4) 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.
5) If implemented, CMO must include support for an instruction (synch_conditional) 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.
6) It points out that previous algorithms inserting memory fences indiscriminately during
compilation are based conservative program analysis, and can have a considerable performance impact.

7) Compared to the previous processor-centric architectures, the mechanisms provided in this paper allow one processor to control the ordering of operations performed by another processor, which not processor-centric.

8) It summaries the characters of synchronization and memory ordering operations in lock intensive Java workloads and demonstrate that a lot of memory ordering operations occur superfluously.

Cons:

1. Implementation of CMO need to go through the program and find those unnecessary memory orderings, it may cause performance degradation to the system.
2. The performance improvements in the software prototype are hindered by the high cost of remote memory ordering.
3. The mechanism provided in this paper address only specific classes of synchronization stalls. It may not work well for those programs with few lock intensive operations.
4. 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).
5. 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_remote) 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.
6. 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.
7. 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.

8. 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.

Thus, the main conclusion that can be drawn here is that CMO technique is very effective in enhancing throughput in specific cases, but it's generally not true. It will not really scale for massively multithreaded application scenarios due to inter-processor communication bottlenecks. Also since the evaluation was done only using Java specs, it needs further study on other workloads to prove efficacy and be considered a candidate for implementation as it involves additional hardware complexity and is not platform independent.