

# Architecture Supported Synchronization-Based Cache Coherence Protocol For Many-Core Processors

He Huang, Nan Yuan, Wei Lin, Guoping Long, Fenglong Song, Lei Yu, Yuping Liu, Lei Liu, Yongbin Zhou, Xiaochun Ye, Junchao Zhang, Dongrui Fan, Zhimin Tang  
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

huangh@ict.ac.cn

## ABSTRACT

The efficient support of cache coherence is extremely important to design and implement many-core processors. In this paper, we propose a synchronization-based coherence protocol to efficiently support cache coherence for shared memory many-core architectures. The unique feature of our scheme is that it doesn't use directory at all. Inspired by scope consistency memory model, our protocol maintains coherence at synchronization point. Within critical section, process cores record write sets (which lines have been written in critical sections) with bloom-filter functions. When the core releases the lock, the write set is transferred to a synchronization manager. When another core acquires the same lock, it gets the write set from the synchronization manager and invalidates stale data in its local cache.

We are now evaluating our scheme on Godson-T many-core processor using SPLASH-2 applications. We expect our synchronization-based coherence protocol can achieve similar performance in cost-effective way compared to a directory-based protocol that requires large amount of hardware resources and huge design verification effort.

## 1. Introduction

Traditional cache coherence approaches, such as snoopy protocol and directory-based protocol, either cannot be used or is not suitable for many-core processors. A directory-based protocol has well-known problems: directory-based protocol maintains cache coherence at cache line granularity. This fine-grain granularity incurs substantial overhead in directory storage and on-chip-networks traffics. For example, every cache line has a bit vector to record which cores share the same cache line in their local cache. As the number of on-chip core increasing, the bit vector will take up a lot of on-chip storage, even comparable to cache data storage. In fact, bit vector is not only over-consuming storage but also inefficient. It is common for many-core processor that several processes run on it currently. Only intra-process coherence is demanded. Inter-process coherence has no use except for OS or runtime data. Directory-based protocol has no awareness about this issue, since its bit vector covers all on-chip cores. In terms of on-chip-network traffics, every store to shared data must invalidate other replications; this becomes more costly as core count increases. Furthermore, directory-based protocols are notoriously hard to implement and verify for their huge state space and many corner cases. Therefore, we believe many-core architectures require alternative approaches for cache coherence.

In this paper, we propose a complexity-effective, performance-comparable scheme to support cache coherence on many-core

architectures. Instead of directory-based protocol, we propose architecture supported, synchronization-based cache coherence protocol that maintains cache coherence at synchronization point and does not use directory at all.

Our contributions in this work are summarized as follows:

We propose an alternative scheme to maintain cache coherence for many-core architectures. Essentially, synchronization-based protocol provides cache coherence at feasible design and debug cost and supports lock-based programming model on shared memory systems.

This paper is organized as follows: Section 2 presents the design details; Section 3 discusses the implementation issues; Section 4 gives evaluation plan.

## 2. Architecture Supported Synchronization-Based Cache Coherence Protocol

This section describes the paper's key ideas for maintaining cache coherence on many-core without directory structures. The ideas are to exploit coherence demands implicitly expressed in shared memory parallel program itself and use effective hardware to enforce coherence at synchronization points.

### 2.1 Details of Synchronization-Based Protocol

For shared memory parallel programs, programmers use synchronization primitives, such as lock and barrier, to ensure exclusive access to shared data or to coordinate program phases (barrier can be seen as ending a critical section first and entering another one right after). Beside mutual exclusion semantics, synchronization primitives also express the coherence demands for shared data objects they protect. From this view, we maintain cache coherence at synchronization point instead of at cache line granularity in directory-based protocol, and replace directory with our synchronization manager in many-core architectures.

Directly porting synchronization-based protocol used in software DSM to tiled many-core architecture is difficult if not impossible, because generating and comparing diff at OS page granularity is not practical for its huge hardware overhead and tremendous possibility of false sharing. Our scheme attacks these problems by using bloom filters to represent read set and write set at cache line granularity. Bloom filters are well-known techniques in processor microarchitecture as it can represent almost unbounded set using limited resources and it can manipulate the set using only simple bitwise logic. Here, we describe our protocol as follows.

Each L2 controller manages a portion of total memory partitioned by interleaving some middle bits of memory address. Each

memory block has a fixed home L2; it can be cached in both home and non-home L1 data cache in one of following three states: Invalid (INV), Valid (V), and Dirty (D). Different L1 data caches can have same cache block in different states simultaneously (since we do not maintain cache coherence at cache line granularity). L1 data cache uses write back policy. Each synchronization variable (lock or barrier) has its home synchronization manager located using the same interleaving method as ordinary memory block.

When executing in critical section, process core records cache line modified by store instruction into *W-set* (dirty cache line addresses), which is represented using bloom filters. The *diff* which is the difference between original page and modified page in software DSM has no use in our scheme.

Right before a release, the releasing tile writes back dirty lines which are still cached in L1 data cache to their home L2 using information extracted from *W-set*. Then, the releasing tile sends release message to home synchronization manager; *W-set* is piggybacked on the release message. When the synchronization manager gets the release message, it saves attached *W-set*, and makes the released lock available to other threads.

On an acquire, the requesting core sends an acquire message to lock's home synchronization manager. If the lock is available, synchronization manager grants the acquiring core by sending it an acquire-ack message with an attached *W-set* that was previously saved for this lock. After the acquiring core receives the acquire-ack message, it extracts cache lines from the piggybacked *W-set* are already stale and then invalidates these lines from local L1 data cache.

We view barrier synchronization as doing a release first and acquiring a synchronization variable right after (these release and acquisition are relative to consistency and coherence schematics; they have different synchronization meanings to lock acquire/release). We assume threads are initialized with *W-set* recording function enabled. When a thread reaches a barrier, it sends a release message to barrier's home synchronization manager with *W-set* attached on. The releasing thread delays until all other threads participating in the same barrier also reach this barrier. The *W-set* is recorded in synchronization manager on per core basis. When the last thread reaches this barrier, the synchronization manager merges all *W-sets* together, attaches merged *W-set* to acquire-ack message and feeds it back to all stalling cores one by one. Similar to lock acquire, when waiting core receives acquire-ack message, it uses attached *W-set* to invalidate stale cache lines in local L1 data cache.

## 2.2 Synchronization Manager Design

The Synchronization Manager takes charge of lock and barrier operations. In our scheme, locks and barriers are supported by hardware structures, because we need explicit synchronization information to support cache coherence. Pure software synchronization mechanisms using compare-and-swap like atomic instructions are not enough for our scheme.

The Synchronization Manager consists of hardware part and software part. Hardware part records synchronization variables state in small, fast on-chip storages. Software part provides several routines to manage *W-set* and *R-set* attached on synchronization variables. Furthermore, software part provides

extra synchronization variables state buffers to back up hardware part to provide almost unbounded number of nested critical sections.

Each Synchronization variables state buffer (SVSB) consists of following fields: (1) state field indicates current status of synchronization variable; (2) required and current field combined to support barrier synchronization: required field specifies how many threads must reach this barrier before these threads can leave this barrier; current field indicates how many threads have reached this barrier until now; (3) waiting vector is a bit vector where each bit indicates if corresponding core is pending on this synchronization variable; (4) participating vector is a bit vector where each bit indicates if corresponding core is participating in this synchronization operation; it is used for lock synchronization only. We will give its usage later.

Thread Synchronization State Buffers (TSSB) are software structures residing in virtual memory space. TSSB is used to record *W-set* and *R-set* for each synchronization variable and thread pair. It has following fields: (1) incarnation number used to generate more precise coherence information, we illustrate its usage in later section; (2) *W-set* and *R-set*, using bloom filter technique, record which cache lines have been written or read in thread's critical section.

There are several software routines associated with synchronization manager to manage *W-set*. Later section gives some usages about these routines.

## 2.3 W-set Management

Before sending an acquire-ack message, synchronization manager generates *W-set* to indicate which cache lines have been modified. If synchronization manager merely records *W-set* from every release message, and combines all of them into a new *W-set* piggybacked on acquire-ack message to acquiring cores, finally the *W-set* will be full, and local L1 data cache will be flushed every time acquiring core receives the acquire-ack message. We deal with these issues as follows.

For barrier synchronization, the returned *W-set* for thread *i* is merged from all participating thread's *W-set* in its TSSB except thread *i*.

$$W - set_{reti} = \bigcup_{j \neq i} W - set_j$$

For lock synchronization, we use incarnation number in TSSB to prevent it from unnecessarily invalidating local cache lines copy. At this time, the returned *W-set* is:

$$W - set_{reti} = \bigcup_{TSSB[j].incar > TSSB[i].incar} W - set_j$$

## 2.4 Nested Synchronization

Nested Synchronization is necessary for software composition. Since we use hardware structure to assist synchronization, we must provide an unbounded resource illusion although we have only limited hardware budget.

We virtualize SVSB by using a large, almost unlimited software SVSB in virtual memory space to back up bounded hardware

SVSB. Similar to TLB mechanism, if a request for a synchronization variable is not located in hardware SVSB, an internal exception is fired, and software routine of synchronization manager refills requested one from software SVSB and swaps out a hardware entry if necessary.

## 2.5 Implement Scope Consistency

To support scope consistency, we must guarantee that before a thread  $t$  is allowed to enter in a critical section, any store previously performed in the same critical section must be visible to thread  $t$ . We satisfy this requirement by: (1) before release operation is allowed to execute, any store operation before release operation and after corresponding acquire operation in program order must be performed in the memory hierarchy next to core's local storage. Here, local storage is core's L1 data cache, and its next level memory hierarchy is shared L2 cache. In our scheme, we must write back dirty cache line to L2 cache (we can write dirty cache line back to L2 cache by using information extracted from W-set). To be noted, at this moment, these in-critical-section stores are not performed with respect to other cores, since those cores may have stale copy in their L1 data cache; (2) after receiving an acquire-ack message, acquiring core uses information extracted from W-set which is attached to acquire-ack message to invalidate its local stale data from L1 data cache. At this time, those stores in the same critical section executed before previous release is performed with respect to the acquiring core.

## 3. Implementation Issues

### 3.1 Software Assisted Signature Management

As described above, synchronization manager uses software routines to manage W-set. To facilitate these routines, we adopt SoftSig mechanisms. SoftSig exposes signatures as an architectural infrastructure for software programmers. By this means, we can use these signature specific instructions to maintain W-set efficiently. Otherwise, we can only do signature operations at most one word length per instruction.

### 3.2 Programmability Issues

Scope consistency has weaker programmability than release consistency. This introduces some design and implementation burdens for programmers. For example, shared memory data accessed out of critical section will not be guaranteed consistency by ScC systems. To amend this issue, we provide programmers two instructions to guarantee consistency. The `open_scope(scopeid)` instruction opens a consistency scope specified by `scopeid` whereas `close_scope(scopeid)` instruction closes a consistency scope.

### 3.3 Fine-grain Disambiguation

We do disambiguation at cache line granularity until now. If two threads store to different parts of one cache line, system must identify which part of cache line is written by each thread. We can add bit mask for this purpose, where each bit indicate whether

the corresponding byte is modified. Using this approach, we can merge data when cache line is written back to L2 cache.

## 3.4 Thread Suspension and Migration

To support thread suspension and migration, we must: (1) save current W-set (and R-set) in thread context; (2) write back cache lines indicated by current W-set from local L1 data cache to L2. Step (2) is required since if this thread is scheduled to another core later, the modified data remain in this core's L1 data cache, and they will not be written back to L2 cache when the thread leaves critical section sometime later.

As described in previous section, SVSB uses bit vector to record which cores are waiting on this synchronization variable. Since each bit indicates whether a core, not a thread, is waiting on this synchronization variable, so if a pending-on-synchronization thread is suspended and then is scheduled to another core, it will not receive acquire-ack message, since this message will be sent to the core indicated by waiting bit vector. To deal with this issue, we should use link list instead of bit vector to represent waiting threads. For each node in link list, it records thread's id, which can be used by software routine to locate its current resident core. For the sake of simplicity, we still use bit vector approach in our evaluation.

## 4. Evaluation Plan

We are evaluating our design now. The evaluation work is not finished yet by the submission of this paper. Therefore, we give our evaluation methods only; the quantitative results will be present in other publications.

### 4.1 Experimental Setup

We evaluate our scheme using sim-godson cycle-accurate execution-driven simulator which implements MIPS ISA. The simulator models in-order processor core, memory hierarchy and on-chip networks in sufficient detail that can be used to design real logic in verilog RTL.

To support shared memory parallel program, such as SPLASH-2 benchmark, we add a thread library to support thread management and synchronization.

### 4.2 Performance

We will compare the overall performance between directory-based protocol and our synchronization-based protocol using SPLASH-2 programs. We expect that the performance of our scheme will be comparable to directory-based protocol.

## Acknowledgements

We thank Rajeev Balasubramonian for providing assistance with improving my writing.

This work is supported in part by the National Natural Science Foundation of China under Grant No.60736012; the National Grand Fundamental Research 973 Program of China under Grant No.2005CB321600.