GLocks: Efficient Support for Highly-Contended Locks in Many-Core CMPs

Authors: José L. Abellán, Juan Fernández and Manuel E. Acacio

Presenter: Guoliang Liu
Outline

• Introduction
• Motivation
• Background on Global lines technology
• The GLocks Mechanism
  – Architecture
  – Synchronization protocol
  – Programming issues
• Evaluation
• Conclusion
Introduction

• CMP: Single-Chip Multi-Processors
  – Higher performance and lower-power than complex uniprocessors.
  – Power: All processors on one die share a single connection to rest of system
  – Omnipresent in almost all market segments.
    • Intel core i7, AMD phenom™ X4 Quad core, Sun Ultra SPARC T2 ...

• Following the well-known Moore’s law, future CMP layouts will integrate hundreds of cores——Many-core CMPS
Introduction

• Parallelism classifications:
  – Instruction level
  – Loop level
  – **Thread level** - Future trend
  – Process level - Future trend
Instruction Level Parallelism

• **ILP** is a measure of how many of the operations in a computer program can be performed simultaneously.
  – An example:
    • $e = a + b$
    • $f = c + d$
    • $g = e \times f$
Loop Level Parallelism

- Belongs to **ILP**
- Primary focus of dependence analysis
- Determine all dependences and find cycles
- Example:

```c
for (i=1; i<=100; i= i+1){
    a[i] = a[i] + b[i]; //s1
    b[i+1] = c[i] + d[i]; //s2
}
```

```c
a[1] = a[1] + b[1];
for (i=1; i<=99; i= i+1){
    b[i+1] = c[i] + d[i];
    a[i+1] = a[i+1] + b[i+1];
}
```

```c
b[101] = c[100] + d[100];
```
Thread Level Parallelism

• **TLP** (also known as *function parallelism* and *control parallelism*) is a form of parallelization of computer code across multiple processors in parallel computing environments.

• Programmers are good to find thread level parallelism.
Process Level Parallelism

- processes can coordinate work and share data using any and all of the inter-process communication (IPC) features
- In computing, a process is an instance of a computer program that is being sequentially executed by a computer system that has the ability to run several computer programs concurrently.
Introduction

• CMPs focus on exploiting thread-level parallelism (TLP).

• When we do this, we need consider the following questions
  – Synchronization
  – Lock acquisition and release cost
Introduction

- **Synchronization** mechanisms of TLP rely on shared variables to coordinate multithreaded access to shared data structures thus avoiding data dependency conflicts.

- **Lock acquisition and release operations timing** is deeply affected by the performance and scalability of the cache coherence protocol.

- Consequently, the longer the idle time spent on lock acquisition and release operations, the larger the parallel efficiency reduction.
Introduction

Question:
How to implement a fast lock acquisition and release mechanism?
Motivation

• Some well-known Lock implementations:
  – **Simple Lock**: inefficient under high contention.
  – Ticket Lock, Array-based Lock and **MCS lock**: scalable and fair implementations. Inefficient under low contention, high storage overhead and high latency.
  – Reactive Lock and smart Lock: Choose the best synchronization algorithm depending on lock contention
  – Queue-on-lock: similar implementation with MSC
Mutual Exclusion: We want to prevent two or more threads from being active concurrently for some period, because their actions may interfere incorrectly.

How to realize? One way is the Lock mechanism.
Simple Lock

• Simple Lock also called spin locks, is a Boolean variable that indicates whether or not one of the processes is in its critical section:
  – lock == 1 – some process is in its CS (the lock is “locked”)
  – lock == 0 – no process in CS (the lock is “unlocked”)
Implementing a spin Locks

```c
bool lock = false;
process P[i = 0 to n-1] {
    while (true) {
        Lock(&lock); /* CSentry */
        critical section;
        Unlock(&lock); /* CSexit */
        non-critical section;
    }
}
procedure Lock(var bool location) {
    <await (!location) location = true;>
}
procedure Unlock(var bool location) {
    location = false;
}
```
How to realize lock and unlock in simple lock mechanism

• Test-and-set lock using Test&Set instruction (t&s):

Lock:    t&s register, location  // try to lock the location
         bnz Lock               // if not 0, try again
         ret                   // return to caller

Unlock:  st location, #0|     // write 0 to location
         ret                 // return to caller

• Drawbacks: Causes high memory contention while waiting for the lock:
  – T&s is treated as a write operation – invalidates cached copies if any
  – Unsuccessful t&s generate memory accesses (bus traffic)
  – Also wasting CPU time because of busy waiting
How to realize lock and unlock in simple lock mechanism

• Test&set lock with exponential backoff:

```assembly
lock:    ld    reg2, #1    // initial delay
retry:  t&s   reg1, location  // try to lock
        beqz   done        // if 0, branch to done
        sll    reg2, reg2, 1  // increase delay by 2
        pause  reg2        // delay by value in reg2
        j      retry       // jump to retry
done:    ret         // return control to caller
```

• Drawbacks: Causes high memory contention while waiting for the lock:
  – T&s is treated as a write operation – invalidates cached copies if any
  – Unsuccessful t&s generate memory accesses (bus traffic)
  – Also wasting CPU time because of busy waiting
How to realize lock and unlock in simple lock mechanism

• An enhancement of test-and-set.
• Test-Test&Set Lock

```assembly
lock: ld reg1, location // load of lock
bnz lock // if not 0, spin
t&s reg1, location // try to lock
bnz lock // if not 0, retry
ret // return control to caller
```

• Alleviate the contention, but still with high cost under high contention.
MCS Lock

- MCS Lock – Based on Linked List
- Acquire
  1. Fetch & Store Last processor node (Get predecessor & set tail)
  2. Set arriving processor node to locked
  3. Set last processor node’s next node to arriving processor node
  4. Spin till Locked=false

![Diagram of MCS Lock process]
MCS Lock

- MCS Lock – Based on Linked List
- Release
  - Check if next processor node is set (check if we completed acquisition)
  - If set, make next processor node unlocked
MCS Lock

• Advantages:
• Scalable and fair implementations
• Efficient under high contention.
Background on Global lines technology

- Global-Line technology (GLines):
  - Every G-line is basically a shared wire that broadcasts 1-bit messages (signals from now on) across one dimension of the chip in a single clock cycle.
  - G-lines are to broadcast the control signals of EVC in order to communicate the availability of free buffers and virtual channels much more accurately (original EVC uses thresholds to conservatively communicate available resources).

The GLocks Mechanism

- The GLocks mechanism relies on a G-line-based Network.

- Components:
  - G-lines
  - Controllers(R, Sx and Cx)
    - Local controllers(Cx)
    - Lock managers(R and Sx)
      - Primary(R)
      - Secondary(Sx)
The GLocks Mechanism

• Logical view of the G-lines-based network

Figure 2. GLocks architecture for a 9-core CMP with a 2D-mesh network. Figure 3. Logical view of the G-line-based network for a 9-core CMP with a 2D-mesh network.
The GLocks Mechanism

• **Synchronization Protocol:**
  – G-line-based network enables the exchange of 1-bit messages (signals) between the *local controllers* and the *lock managers*
  – Tree types of signals to perform the synchronization
    • REQ
    • REL
    • TOKEN
The GLocks Mechanism

- Strategy to grant the lock among those processor cores: Round-robin strategy.

Figure 3. Logical view of the G-line-based network for a 9-core CMP with a 2D-mesh network.
The GLocks Mechanism

- Synchronization Protocol:
The GLocks Mechanism

• **Synchronization Protocol:**

(b) Lock is granted to Core0 (cycle 4).
The GLocks Mechanism

• **Synchronization Protocol:**

(c) Core0 releases the lock (cycle $m$) and $SI$ designates Core1 to be the next lock holder (cycle $m+1$).
The GLocks Mechanism

• *Synchronization Protocol:*

(d) Core2 releases the lock (cycle $p$) and S2 designates Core3 to be the next lock holder (cycle $p+3$).
The GLocks Mechanism

• Programmability Issues:

```c
GL_Lock() {
    asm {
        # Arrival at the CS: set lock_req
        mov 1, lock_req
        
        # Busy-wait until lock_req is reset
        loop:
            bnz lock_req, loop
    }
}

GL_Unlock() {
    asm {
        # Release lock: set lock_rel
        mov 1, lock_rel
    }
}
```

Figure 5. Encapsulating the GLocks functionality into the lock/unlock library-level methods.
The GLocks Mechanism

• Implementation Cost for GLocks:

<table>
<thead>
<tr>
<th>Component</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>G-lines</td>
<td>$C - 1$</td>
</tr>
<tr>
<td>Primary Lock Managers</td>
<td>1</td>
</tr>
<tr>
<td>Secondary Lock Managers</td>
<td>$\sqrt{C}$</td>
</tr>
<tr>
<td>Local controllers</td>
<td>$C - 1$</td>
</tr>
<tr>
<td>fsx Flags</td>
<td>$\sqrt{C}$</td>
</tr>
<tr>
<td>fx Flags</td>
<td>$C$</td>
</tr>
<tr>
<td>Lock Acquire (worst case)</td>
<td>4 cycles</td>
</tr>
<tr>
<td>Lock Acquire (best case)</td>
<td>2 cycles</td>
</tr>
<tr>
<td>Lock Release</td>
<td>1 cycle</td>
</tr>
</tbody>
</table>
Evaluation

- Testbed

<table>
<thead>
<tr>
<th></th>
<th>CMP BASELINE CONFIGURATION.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of cores</strong></td>
<td>32</td>
</tr>
<tr>
<td><strong>Core</strong></td>
<td>3GHz, in-order 2-way model</td>
</tr>
<tr>
<td><strong>Cache line size</strong></td>
<td>64 Bytes</td>
</tr>
<tr>
<td><strong>L1 I/D-Cache</strong></td>
<td>32KB, 4-way, 2 cycles</td>
</tr>
<tr>
<td><strong>L2 Cache (per core)</strong></td>
<td>256KB, 4-way, 12+4 cycles</td>
</tr>
<tr>
<td><strong>Memory access time</strong></td>
<td>400 cycles</td>
</tr>
<tr>
<td><strong>Network configuration</strong></td>
<td>2D-mesh</td>
</tr>
<tr>
<td><strong>Network bandwidth</strong></td>
<td>75 GB/s</td>
</tr>
<tr>
<td><strong>Link width</strong></td>
<td>75 bytes</td>
</tr>
</tbody>
</table>
Evaluation

- Benchmark:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Size</th>
<th>Locks</th>
<th>H-C Locks</th>
<th>Access Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCTR</td>
<td>1,000 iterations</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>MCTR</td>
<td>1,000 iterations</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>DBLL</td>
<td>1,000 iterations</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>PRCO</td>
<td>1,000 iterations</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>ACTR</td>
<td>1,000 iterations</td>
<td>2</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>RAYTR</td>
<td>teapot</td>
<td>34</td>
<td>2</td>
<td>SCTR</td>
</tr>
<tr>
<td>OCEAN</td>
<td>258x258 ocean</td>
<td>3</td>
<td>1</td>
<td>SCTR</td>
</tr>
<tr>
<td>QSORT</td>
<td>16384 elements</td>
<td>1</td>
<td>1</td>
<td>PRCO</td>
</tr>
</tbody>
</table>
Evaluation

• contended locks are implemented using GLocks mechanisms.
• Simple Lock algorithm are enhanced with test-and-test&Set optimization.
• Test on simulated 32-core CMP
Evaluation

- LCR(lock’s contention rate) of a particular lock:
  
  \[ LCR_{grAC_i} = \frac{Cycles(Lock, grAC_i)}{\sum_{g=1}^{32} Cycles(Lock, grAC_g)} \]  

- grAC: the number of concurrent requesters, range(1~32)
Evaluation

• Execution time:

An average reduction of 42% and 14% in execution time
Evaluation

• Network traffic:

![Network Traffic Graph]

Figure 9. Normalized Network Traffic. 
an average reduction of 76% and 23% in network traffic
Evaluation

- Energy-delay$^2$ metric for full CMP:
- Energy-delay-product:
  - Short as $EDP = E \times D$;
  - Then $EP^2P = E \times D^2$;

![Graph showing overall normalized CMP energy-delay² product.](image)

*Figure 10. Normalized energy-delay² product (ED²P) metric for the full CMP.*

An average reduction of 78% and 28% in the energy-delay² product (ED²P) metric.
Conclusion

• Lock contention is recognized as a key constraint to performance and scalability on many-core CMPs when trying to exploit thread-level parallelism.
  – High contention: GLocks
  – Low contention: Simple Locks
References