Multicore Cache Coherence Control by a Parallelizing Compiler

Hironori Kasahara, Keiji Kimura, Boma A. Adhi, Yuhei Hosokawa, Yohei Kishimoto, Masayoshi Mase

Department of Computer Science and Engineering
Waseda University

Waseda University @ COMPSAC2017, CAP, Turin, July 5th, 2017
What is Cache Coherency?

**Cache coherency:**

Data should be consistent from all CPU Core

Waseda University @ COMPSAC2017, CAPI, Turin, July 5th, 2017
What is Cache Coherency?
What is Cache Coherency?

x = 2;

PE0 Cache

PE1 Cache

x is now invalid

Waseda University @ COMPSAC2017, CAP, Turin, July 5th, 2017
Why Compiler Controlled Cache Coherency?

Current many-cores CPU (Xeon Phi, TilePro 64, etc) uses hardware cache coherency mechanism.

• Hardware cache coherency for many-cores (hundreds to thousand cores):
  • Hardware will be complex & occupy large area in silicon
  • Huge power consumption
  • Design takes long time & larger cost

• Software based coherency:
  • Small hardware, low power & scalable
  • No efficient software coherence control method has been proposed.
Proposed Software Coherence Method on OSCAR Parallelizing Compiler

• Coarse grain task parallelization with earliest condition analysis (control and data dependency analysis).

• OSCAR compiler automatically controls coherence using following simple program restructuring methods:
  • To cope with stale data problems:
    • Data synchronization by compilers
  • To cope with false sharing problem:
    • Data Alignment
    • Array Padding
    • Non-cacheable Buffer

MTG generated by earliest executable condition analysis

Waseda University @ COMPSAC2017, CAP, Turin, July 5th, 2017
The Renesas RP2 Embedded Multicore

2 clusters of 4 cores hardware cache coherent control SMP, No coherence support for more than 5 cores
Non Coherent Cache Architecture

Local Cache

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

V: valid bit
D: dirty bit

_local cache control instructions from the owner CPU_
- self-invalidate
- Writeback
- flush

PE0

CPU

Local Cache

PE1

CPU

Local Cache

...PEn

CPU

Local Cache

Interconnection Network

Shared Memory

Waseda University @ COMPSAC2017, CAP, Turin, July 5th, 2017
Problem in Non-coherent Architecture (1): Stale Data

Stale Data:

An obsolete shared data exists on a processor cache after a new value is updated by another processor core.

Global Variable Declaration
int a = 0;
int b = 0;
int c = 10;

Correct value with coherent control
20 0 20 -

A=20 by PE0 is not published and PE1 calculates c using stale data

Value is not coherent

Waseda University @ COMPSAC2017, CAP, Turin, July 5th, 2017
Solving Stale Data Problem

The compiler automatically inserts a writeback instruction and a synchronization barrier on PE0 code. Then, it inserts a self-invalidate instruction on PE1 after the synchronization barrier.

- **Modified** leads to **Valid**.
  - **Valid or Stale**
    - SB1(); /* def A */
      - Writeback A;
      - sync_flg = 1;
      - Writeback sync_flg;
    - SB2();
  - do {
      - Self-invalidate sync_flg;
      - while (sync_flg != 1);
    }
  - Self-invalidate A
  - SB3(); /* def or use A */

Waseda University @ COMPSAC2017, CAP, Turin, July 5th, 2017
Problem in Non-coherent Architecture (2): False Sharing

**False sharing:**

A condition which multiple cores share the same memory block or cache line. Then, each processor updates the different elements on the same block/cache line.

Depending on the timing of the line replacement, the stored data will be inconsistent.

With cache coherency:

```
10 20 - -
```

```
? ? - -
```

```
10 0 - -
```

```
0 20 - -
```

```
0 0 - -
```

```
a and b are different elements on the same cache line
```

Waseda University @ COMPSAC2017, CAP, Turin, July 5th, 2017
Solving False Sharing (1): Data Alignment

Data alignment solves the false sharing problem.

Different data accessed by different PE should not share a single cache line.

This approach works for scalar variables and small-sized one-dimensional array.

Variable Declaration

```c
int  a __attribute__((aligned(16))) = 0;
int  b __attribute__((aligned(16))) = 0;
/* a, b are assigned to the first element of each cache line */
```

PE0 Cache

```
10 - - -
```

PE0 Cache

```
20 - - -
```

Mem

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
</tbody>
</table>
```

Updates by PE0 and PE1 are separately performed by write back.
Two-dimensional Array Problem

- Splitting an array cleanly along the cache line is not always possible.
  - Lowest dimension is not integer multiply of cache line

```c
int a[6][6];

for (i = 0; i < 3; i++) {
    for (j = 0; j < 6; j++) {
        a[i][j] = i * j;
    }
}

for (i = 3; i < 6; i++) {
    for (j = 0; j < 6; j++) {
        a[i][j] = i * j;
    }
}
```
Solving False Sharing (2): Array Padding

The compiler inserts a padding (dummy data) to the end of the array to match the cache line size.

```c
int a[6][8] __attribute__((aligned(16))); /* a is aligned to the first element of the cache line */
```

```c
for (i = 0; i < 3; i++) {
    for (j = 0; j < 6; j++) {
        a[i][j] = i * j;
    }
}
```

```c
for (i = 3; i < 6; i++) {
    for (j = 0; j < 6; j++) {
        a[i][j] = i * j;
    }
}
```
Solving False Sharing (3): Non-cacheable Buffer

Data transfer using non-cacheable buffer:
The idea is to put a small area in the main memory that should not be copied to the cache along the border between area modified by different processor core.

```c
for (i = 4; i < 5; i++) {
    for (j = 0; j < 6; j++) {
        a[i][j] = i * j;
        b[i-1][j] = i * j;
    }
}
```

```c
for (i = 4; i < 5; i++) {
    for (j = 0; j < 6; j++) {
        a[i][j] = i * j;
        nc_buf[i-4][j] = i * j;
    }
}
```

```c
for (i = 5; i < 6; i++) {
    for (j = 0; j < 6; j++) {
        a[i][j] = i * j;
        b[i-1][j] = i * j;
    }
}
```

```c
int a[6][6] __attribute__((aligned(16)));
/* a is assigned to the first element of the cache line */
int nc_buf[1][6] __attribute__((section("UNCACHE")));
/* nc_buf is prepared in non-cacheable area */
```
Performance Evaluation

• 10 Benchmark Application in C
  • In Parallelizable C format (Similar to Misra C in embedded field)
  • Fed into source to source automatic parallelization OSCAR Compiler with non-cache coherence control
  • Parallelized code then fed into Renesas C Compiler

• Evaluated in RP2 Processor
  • 8 core
  • Dual 4 core modules
  • Hardware cache coherency for up to 4 cores in the same module
Performance of Hardware Coherence Control & Software Coherence Control on 8-core RP2
## SPEC Result

<table>
<thead>
<tr>
<th>Application</th>
<th>The Number of Processor Core</th>
<th>SPEC2000</th>
<th>SPEC2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>equake</td>
<td>1</td>
<td>1.00</td>
<td>4.37</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.07</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.52</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2.63</td>
<td></td>
</tr>
<tr>
<td>art</td>
<td>1</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.67</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2.65</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.95</td>
<td></td>
</tr>
<tr>
<td>lbm</td>
<td>1</td>
<td>1.00</td>
<td>2.90</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.06</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.76</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3.65</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.28</td>
<td></td>
</tr>
<tr>
<td>hammer</td>
<td>1</td>
<td>1.00</td>
<td>2.99</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.79</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3.19</td>
<td></td>
</tr>
</tbody>
</table>

**Speedup**

In the diagram, the blue bars represent the SMP (Hardware Coherence) and the red bars represent the application performance. The x-axis shows the number of processor cores, while the y-axis represents the speedup. The data is sourced from SPEC2000 and SPEC2006.
NAS Parallel Benchmark and MediaBench II

<table>
<thead>
<tr>
<th>Application/the number of processor core</th>
<th>SMP (Hardware Coherence)</th>
<th>MediaBench II</th>
</tr>
</thead>
<tbody>
<tr>
<td>cg</td>
<td>1.07 1.32 1.84 3.71</td>
<td>4.92 3.02</td>
</tr>
<tr>
<td>mg</td>
<td>1.00 1.32 2.01 5.66</td>
<td>4.92 3.02</td>
</tr>
<tr>
<td>bt</td>
<td>1.00 1.05 1.95 2.86</td>
<td>2.19 2.19</td>
</tr>
<tr>
<td>lu</td>
<td>1.00 1.05 1.79 2.87</td>
<td>2.19 2.19</td>
</tr>
<tr>
<td>sp</td>
<td>1.00 1.07 1.55 2.70</td>
<td>1.70 1.67</td>
</tr>
<tr>
<td>MPEG2 Encoder</td>
<td>1.00 1.02 1.89 3.32</td>
<td>1.89 1.89</td>
</tr>
</tbody>
</table>

Waseda University @ COMPSAC2017, CAP, Turin, July 5th, 2017
Conclusions

• OSCAR compiler automatically controls coherence using following simple program restructuring methods:
  • To cope with stale data problems:
    • Data synchronization by compilers
  • To cope with false sharing problem:
    • Data Alignment
    • Array Padding
    • Non-cacheable Buffer
• Evaluated using 10 benchmark applications Renesas RP2 8 core multicore processor.
  • SPEC2000
  • SPEC2006
  • NAS Parallel Benchmark
  • MediaBench II
Conclusion (cont’d)

• Provides **good speedup automatically**, few examples with 4 cores:
  • 2.63 times SPEC2000 equake (2.52 with hardware)
  • 3.28 times SPEC2009 lbm (2.9 with hardware)
  • 3.71 times on NPB cg (3.34 with hardware)
  • 3.02 times on MediaBench MPEG2 Encoder (3.17 with hardware)

• **Enable usage of 8 cores** automatically on RP2 with good speedup:
  • 4.37 times SPEC2000 equake
  • 4.76 times SPEC2009 lbm
  • 5.66 times on NPB cg
  • 4.92 times on MediaBench MPEG2 Encoder
Thank you