Architecting and Programming a Hardware-Incoherent Multiprocessor Cache Hierarchy

Wooil Kim, Sanket Tavarageri, P. Sadayappan, Josep Torrellas

University of Illinois at Urbana-Champaign
Ohio State University

IPDPS 2016. May 2016
Motivation

• Continued progress in transistor integration $\rightarrow$ 1,000 cores/chip
• Need to improve energy efficiency
• Example: Intel Runnemedes [Carter HPCA 2013]
Intel Runnemede

- Simplifies architecture:
  - Narrow-issue cores
  - Cores and memories hierarchically organized in clusters
  - Single address space
  - On-chip cache hierarchy without hardware cache coherence

- Hardware-incoherent caches:
  - Easier to implement
  - How to program them?
Goal and Contributions

Goal: Programming environment for a hardware-incoherent cache hierarchy

Contributions:

- **Hardware extensions** to manage hardware-incoherent caches
  - Flavors of Writeback (WB) and Self-invalidate (INV) instructions
  - Two small buffers next to the L1 cache
  - Hardware table in the cache controllers
- Two user-friendly programming models
  - Rely on annotating synchronization operations and relatively simple compiler analysis
- Average performance only 5% lower than hardware-coherent caches
How to Ensure Data Coherence?

Hardware-incoherent caches do not rely on snooping or directory

1. write A
2. writeback A
3. self-invalidate A
4. read A

P0

wr x
WB x
sync

P1

sync
INV x
rd x
WB and INV Instructions

- Memory instructions that give commands to the cache controller
- **WB(Variable):** writes back variable to the shared cache
  - Cache lines have fine-grain dirty bits
  - WB operates on whole line but only writes back the modified bytes
    - Different cores don’t overwrite each other in case of false sharing

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Data</th>
<th>D</th>
<th>Data</th>
<th>D</th>
<th>Data</th>
</tr>
</thead>
</table>

- **INV(Variable):** self-invalidates variable from the local cache
  - Uses a per-line valid bit
  - Writes back dirty bytes in the line, then invalidates the line
    - Prevents losing any dirty data

- **WB ALL, INV ALL** // write back / invalidate the whole cache
Programming Models

1. Shared-memory model inside each block and MPI across blocks
2. Shared-memory across all cores
Model 1: Shared Inside Block + MPI across

P0
wr A
wr B
...
wr C
...
WB
sync
...

P1
...

sync
INV
...
rd A
rd B
...
rd C

What to writeback

When to writeback

What to invalidate

When to invalidate
Orchestrating Communication

- Use synchronization as hints for communication
  - WB(vars) before every synchronization point; INV(vars) after
  - If communication variables cannot be computed, use WB/INV ALL
## Annotations for Different Communication Patterns

<table>
<thead>
<tr>
<th>Barriers</th>
<th>Critical sections</th>
<th>Flags</th>
<th>Dynamic happens-before epoch ordering (e.g. task queue)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epoch(s)</td>
<td>Epoch INV LockAcq WB LockRel</td>
<td>Epoch</td>
<td>Epoch WB LockAcq LockRel</td>
</tr>
<tr>
<td>WB Barrier</td>
<td>Epoch</td>
<td>Epoch</td>
<td>Epoch</td>
</tr>
<tr>
<td>INV Epoch(s)</td>
<td>Epoch WB LockRel</td>
<td>Flag wait INV Epoch</td>
<td>LockAcq LockRel</td>
</tr>
</tbody>
</table>

Need to detect data race communication and enforce it with WB/INV
### Application Analysis

<table>
<thead>
<tr>
<th>Application</th>
<th>Barrier</th>
<th>Critical Section/flag</th>
<th>Dyn Happens-Before</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LU</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHOLESKY</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>BARNES</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>RAYTRACE</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>VOLREND</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>OCEAN</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>WATER</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

- **Instrumentation procedure**
  - Analyze the communication patterns
  - If had sophisticated compiler, could do more efficient WB/INV
Hardware Support for Small Critical Sections

- **Modified Entry Buffer (MEB)**:
  - For small code sections such as critical sections
  - Accumulates the written line entry numbers $\rightarrow$ WB only those at end

```
lock
wr A
wr B
wr B
...
// do not WB whole cache
unlock
```

![Diagram showing cache and MEB with entries](image_url)
Hardware Support for Small Critical Sections (II)

• Invalidated Entry Buffer (IEB):
  – For small code sections such as critical sections
  – Accumulate invalidated line addresses → avoid invalidating twice

```c
lock
// don’t inval whole cache
rd A
rd B
rd B
...
unlock
```

![Cache and IEB diagram](image)
Programming Models

1. Shared-memory model inside each block and MPI across blocks
2. Shared-memory across all cores
Model 2: Shared-Memory Across All Cores

- Inefficient solution: always WB/INV through L3 cache

- Propose: Level-adaptive WB and INV
  - WB/INV automatically communicate through the closest shared level of the cache

- Closest shared cache level depends on thread mapping, which is unknown at compile time
Idea: Exploit Producer-Consumer Information

- Software identifies producer-consumer thread pairs
  - (e.g., thread $i$ produces data that will be consumed by thread $j$)
- Thread $\rightarrow$ core mapping unknown at compile time

\[
\begin{align*}
\text{Thread } i \quad &\quad \text{Thread } j \\
X = &\quad \text{INV\_PROD}(x,i) \\
\ldots &\quad \ldots = X \\
\text{WB\_CONS}(x,j) &\quad \text{Epoch}
\end{align*}
\]

- Software instruments the code with level-adaptive WB/INV
  - Producer: \text{WB\_CONS} (addr, ConsID)
  - Consumer: \text{INV\_PROD} (addr, ProdID)
Hardware Support for Level Adaptive WB/INV

- L2 cache controller has a hardware table (ThreadMap)
  - Contains IDs of the threads that have been mapped in the block
- When executing WB_CONS (addr, ConsID):
  - Hardware checks if ConsID is running on the block
  - If so: WB pushes data to L2 only; else, to both L2 and L3
- Same when executing INV_PROD (addr, ProdID)
Compiler Support to Extract P-C Pairs

- **Approach:** Use ROSE compiler to
  - Find P-C relation across OpenMP *for* constructs
  - Inter-procedural CFG
  - Dataflow analysis between potential P-C pairs
- **Assumption:** Static scheduling of threads to processors

```c
#pragma omp parallel for
for (i=0; i<N; i++) {
    A[i] = ...;
    B[i] = ...;
}

#pragma omp parallel for
for (i=0; i<N; i++) {
    ... = A[i] + ...;
    ... = B[i+1] + ...;
}
```

- **Region Assignments:**
  - \( A[i] \): Region(A, 0, N, \# of threads)
    \[ = A: [ (N/th)*myid, (N/th)*(myid+1) ] \]
  - \( B[i] \): Region(A, 0, N, \# of threads)
    \[ = B: [ (N/th)*myid, (N/th)*(myid+1) ] \]
  - \( B[i+1] \): Region(A, 1, N, \# of threads)
    \[ = B: [ (N/th)*myid +1, (N/th)*(myid+1) +1 ] \]
Evaluation

- SESC simulator
- 4-issue out-of-order cores with 32KB L1 caches
- MESI Coherence protocol

- Intra-block experiments:
  - 16 cores sharing a 2MB banked L2 cache
  - Each core: 16-entry MEB, 4-entry IEB
  - SPLASH2 applications

<table>
<thead>
<tr>
<th>HCC</th>
<th>Directory hardware cache coherence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>Basic WB / INV</td>
</tr>
<tr>
<td>B+M</td>
<td>Base + MEB</td>
</tr>
<tr>
<td>B+I</td>
<td>Base + IEB</td>
</tr>
<tr>
<td>B+M+I</td>
<td>Base + MEB + IEB</td>
</tr>
</tbody>
</table>
Execution Time

With MEB/IEB: average performance is only 2% lower than HCC

Not shown: network traffic also comparable
Evaluation

• Inter-block experiments:
  – 4 blocks of 8 cores each
  – Each block has a 1MB L2 cache
  – Blocks share a 16MB banked L3
  – NAS applications analyzed with the ROSE compiler

<table>
<thead>
<tr>
<th></th>
<th>WB/INV all cached data to L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>WB/INV selective data to L3 (compiler analysis)</td>
</tr>
<tr>
<td>Addr+L</td>
<td>Level Adaptive: WB_CONS/INV_PROD</td>
</tr>
</tbody>
</table>
• When Level-Adaptive WB/INV is applicable, performance improves
  – EP, IS have reductions $\rightarrow$ no ordering, hence no P-C

• Not shown: performance is on average 5% lower than HCC
Conclusions

• Programming a hardware-incoherent cache hierarchy is challenging

• Proposed HW extensions to manage it:
  • Flavors of WB and INV, including level-adaptive
  • Small MEB and IEB buffers next to the L1 cache
  • ThreadMap table in the cache controllers

• Proposed two user-friendly programming models

• Average performance only 5% lower than hardware-coherent caches

• Future work: Enhance the performance with advanced compiler support
Architecting and Programming a Hardware-Incoherent Multiprocessor Cache Hierarchy

Wooil Kim, Sanket Tavarageri, P. Sadayappan, Josep Torrellas

University of Illinois at Urbana-Champaign
Ohio State University

IPDPS 2016. May 2016
Instruction Reordering by HW or Compiler

• Instruction ordering requirement
  – WR → WB → Synchronization → INV → RD

• Other orderings are desirable (e.g. to reduce traffic)
• Cache lines can be evicted at any time
BACK-UP SLIDES
WB and INV Instructions

- Operate at line granularity to minimize cache modifications
- User unaware of the data placement
- Granularity of dirty bits may vary (from byte to entire line)
- Different flavors:
  - WB_byte // variable is a byte
  - WB_halfword
  - WB_ALL // write back the whole cache
  - WB_L3 // push all the way to L3
Issued WB/INV

Reduction in issued WB/INV in some applications