Compiler Support for Software Cache Coherence

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

The Ohio State University
University of Illinois at Urbana-Champaign

HiPC’16
Cache Coherence

- Cache Coherence is required on **Shared Memory multi-processors** that have **private caches** so that all processors see values of latest assignments to variables.
- Cache coherence in hardware (snooping bus, directory-based) is not scalable/introduces much complexity.

![Diagram]

The cache coherence problem. Initially processors 0 and 1 both read location $x$, initially containing the value 0, into their caches. When processor 0 writes the value 1 to location $x$, the stale value 0 for location $x$ is still in processor 1’s cache.

**Figure:** Need for **Cache Coherence on parallel systems** Source: Mark Heinrich
Software Cache Coherence

- Alternative - Software Cache Coherence (SCC): a compiler introduces coherence instructions - writebacks, and invalidates in a parallel program

- Benefits of SCC:
  - Scalable
  - Selective enforcement of coherence
  - Simpler hardware

- We develop compiler techniques for efficient orchestration of cache coherence in software

- We use the Polyhedral model to precisely identify coherence data for affine computations

- We develop an inspector-executor approach for iterative irregular computations
Execution of parallel programs on our software managed caches consists of **epochs** (intervals between global synchronization points).

**Self-invalidation:** In an epoch, a processor invalidates potentially stale words present in its cache (and which it may need to read).

**Writebacks:** A processor writes back to shared memory all the *dirty* words of its cache (and which may be needed by other processors): per-word dirty bits keep track of which words are *dirty*.

During an epoch, the write ranges of different threads should not overlap (a program should be data-race free); otherwise we may lose information by overwriting modified words.
The Coherence API

invalid_word(void *addr);
invalid_dword(void *addr);
invalid_qword(void *addr);
invalid_range(void *addr, int num_bytes);

writeback_word(void *addr);
writeback_dword(void *addr);
writeback_qword(void *addr);
writeback_range(void *addr, int num_bytes);
Regular code – Polyhedral algorithms
for \( t1 = 0; t1 \leq tsteps - 1; t1 ++ \) {
    \#pragma omp parallel for private(t3)
    for \( t2 = 0; t2 \leq n - 1; t2 ++ \) {
        for \( t3 = 1; t3 \leq n - 1; t3 ++ \) {
            S1: B[t2][t3] = B[t2][t3 + 1] + 1;
        }
    }
}

Iteration space:

\[
I^{S_1} = \{ S1[t_1, t_2, t_3] : (0 \leq t_1 \leq tsteps - 1) \land (0 \leq t_2 \leq n - 1) \land (1 \leq t_3 \leq n - 1) \}
\]

Array references:

\[
r^{S_1}_{write} = \{ S1[t_1, t_2, t_3] \mapsto B[t'_2, t'_3] : (t'_2 = t_2) \land (t'_3 = t_3) \}
\]
\[
r^{S_1}_{read} = \{ S1[t_1, t_2, t_3] \mapsto B[t'_2, t'_3] : (t'_2 = t_2) \land (t'_3 = t_3 + 1) \}
\]

Flow dependence:

\[
D_{flow} = \{ S1[t_1, t_2, t_3] \mapsto S1[t_1 + 1, t_2, t_3 - 1] : (0 \leq t_1 \leq tsteps - 2) \land (0 \leq t_2 \leq n - 1) \land (2 \leq t_3 \leq n - 1) \}
\]
Computation of Invalidation Set

```c
for (t1 = 0; t1 <= tsteps - 1; t1++) {
    #pragma omp parallel for private(t3)
    for (t2 = 0; t2 <= n - 1; t2++) {
        for (t3 = 1; t3 <= n - 1; t3++) {
            S1: B[t2][t3] = B[t2][t3+1] + 1;
        }
    }
}
```

Iterations mapped to a processor in an epoch - \(I_{current}\)

Iterators of the parallel loop, and its surrounding loops are parameterized:

\[I_{current}^{S_1} = \{S_1[t_1, t_2, t_3] : (t_1 = t_p) \land (t_2 = t_q) \land (1 \leq t_3 \leq n - 1)\}\]

Determination of data to be invalidated:

\[I_{source} = \mathcal{D}_{flow}^{-1}(I_{current}^{S_1}) \setminus I_{current}^{S_1}\]

\[D_{inflow} = r_{write}^{S_1}(I_{source}) = \{B[t_q, i_1] : 2 \leq i_1 \leq n\}\]
Computation of Writeback Set

\[
\text{for } (t1 = 0; t1 \leq \text{tsteps} - 1; t1 ++) \{
\text{pragma omp parallel for private}(t3)
\text{for } (t2 = 0; t2 \leq n - 1; t2 ++) \{
    \text{for } (t3 = 1; t3 \leq n - 1; t3 ++) \{
        \text{S1: } B[t2][t3] = B[t2][t3 + 1] + 1;
    \}
\}
\}
\]

Determination of data to be written-back:
\[
l_{\text{target}} = D_{\text{flow}}(l_{\text{current}}^{S_1}) \setminus l_{\text{current}}^{S_1};
\]
\[
l_{\text{producer}} = D_{\text{flow}}^{-1}(l_{\text{target}}) \cap l_{\text{current}}^{S_1}
\]
\[
D_{\text{outflow}} = r_{\text{write}}(l_{\text{producer}})
\]

Last writes: writes by iterations which are not sources of any output dependences
\[
l_{\text{live\_out}} = l_{\text{current}}^{S_1} \setminus \text{dom } D_{\text{output}}; D_{\text{live\_out\_data}} = r_{\text{write}}(l_{\text{live\_out}})
\]

Invalidate Set:
\[
D_{\text{writeback}}^{S_1} = (D_{\text{outflow}} \cup D_{\text{live\_out\_data}})
\]
\[
= \{ B[t_q, i_1] : (t_p \leq \text{tsteps} - 2 \land 2 \leq i_1 \leq n - 1) \lor
    (t_p = \text{tsteps} - 1 \land 1 \leq i_1 \leq n - 1) \} \]
for (t1 = 0; t1 <= tsteps - 1; t1++) {
    #pragma omp parallel for private(t3)
    for (t2 = 0; t2 <= n - 1; t2++) {
        invalidate_range(&B[t2][2], sizeof(double)*(n - 1));
        for (t3 = 1; t3 <= n - 1; t3++) {
            S1: B[t2][t3] = B[t2][t3+1] + 1;
        }
        if (t1 == tsteps - 1)
            writeback_range(&B[t2][1], sizeof(double)*(n - 1));
        if (t1 <= tsteps - 2)
            writeback_range(&B[t2][2], sizeof(double)*(n - 2));
    }
}

• Techniques described do not assume any particular mapping of iterations to processors.
• The coherence operations can be minimized with the knowledge of iteration-to-processor mapping (more details in the paper).
Irregular code
Many classes of programs have time loop and indirect data accesses.
For such code, we use inspector-executor approach.

```c
while (converged == false) {
#pragma omp parallel for
for (i=0;i<n;i++) {
    read A[B[i]]; /* data-dependent access*/
}

#pragma omp parallel for
for (i=0;i<n;i++) {
    write A[C[i]]; /* data-dependent access*/
} /* Setting of converged variable not shown*/
}
```

The inspection consists of two steps:

1. The writer thread ids are recorded
2. A data reference is marked conflicted if the reader and writer thread ids are not the same

In the execution phase, the conflicted references are written back and invalidated.
Other irregular code

- We introduce optimizations such as exclusion of read-only data
- In conjunction, conservative bulk coherence operations are used
  (more details in the paper)
### Table: Simulator parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor chip</td>
<td>8-core multicore chip</td>
</tr>
<tr>
<td>Issue width; ROB size</td>
<td>4-issue; 176 entries</td>
</tr>
<tr>
<td>Private L1 cache</td>
<td>32KB Write-back, 4-way, 2 cycle hit latency</td>
</tr>
<tr>
<td>Shared L2 cache</td>
<td>1MB Write-back, 8-way, multi-banked, 11 cycle round-trip time</td>
</tr>
<tr>
<td>Cache line size</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Cache coherence protocol</td>
<td>Snooping-based MESI protocol</td>
</tr>
<tr>
<td>Main Memory</td>
<td>300 cycle round-trip time</td>
</tr>
</tbody>
</table>
# Experimental Evaluation

## Table: Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gemm</td>
<td>Matrix-multiply: $C = \alpha A B + \beta C$</td>
</tr>
<tr>
<td>gemver</td>
<td>Vector Multiplication and Matrix Addition</td>
</tr>
<tr>
<td>jacobi-1d</td>
<td>1-D Jacobi stencil computation</td>
</tr>
<tr>
<td>jacobi-2d</td>
<td>2-D Jacobi stencil computation</td>
</tr>
<tr>
<td>LU</td>
<td>LU decomposition</td>
</tr>
<tr>
<td>trisolv</td>
<td>Triangular solver</td>
</tr>
<tr>
<td>CG</td>
<td>Conjugate Gradient method</td>
</tr>
<tr>
<td>backprop</td>
<td>Pattern recognition using unstructured grid</td>
</tr>
<tr>
<td>hotspot</td>
<td>Thermal simulation using structured grid</td>
</tr>
<tr>
<td>kmeans</td>
<td>Clustering algorithm used in data-mining</td>
</tr>
<tr>
<td>pathfinder</td>
<td>Dynamic Programming for grid traversal</td>
</tr>
<tr>
<td>srad</td>
<td>Image Processing using structured grid</td>
</tr>
</tbody>
</table>
Figure: L1 data cache read misses. The L1 read miss ratios for HCC are also shown.
**Figure:** Comparison of Execution times with HCC as the baseline
Figure: Traffic on the system bus. Average number of words per cycle for HCC is also shown.
Figure: Comparison of Energy Consumption with HCC as the baseline
Experimental Results and Conclusion

- For all benchmarks, performance of SCC-opt is similar to or better than that of HCC and is significantly higher than performance of SCC-basic.

- One of the bottlenecks for using SCC was its performance overhead: because of lack of precise compiler analysis, the techniques had to be conservative.

- The compiler analysis developed removes performance bottleneck for affine programs.

- SCC reduces energy expenditure in caches by 5%.

- The main source of energy savings is, elimination of snooping requests.

- Power reduction by simpler hardware: SCC removes any logic related to snooping and state machine for cache coherence from the cache controller.
THANK YOU