Compiler Support for Software Cache Coherence

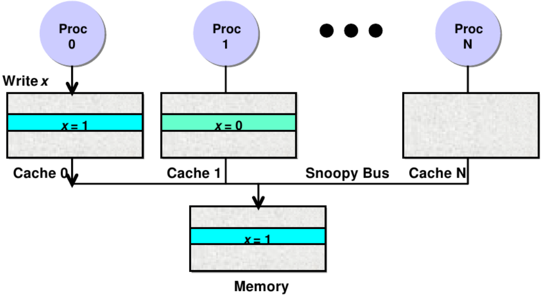
Sanket Tavarageri, Wooil Kim, Josep Torrellas, **P. Sadayappan**

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.



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

```
invalidate_word(void *addr);
invalidate_dword(void *addr);
invalidate_qword(void *addr);
invalidate_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<=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;
    }
  }
}
```

Iteration space:

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

Array references:

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

Flow dependence:

$$\mathcal{D}_{flow} = \{S1[t_1,t_2,t_3] \mapsto S1[t_1+1,t_2,t_3-1] : (0 \le t_1 \le tsteps-2) \wedge (0 \le t_2 \le n-1) \wedge (2 \le t_3 \le n-1)\}$$

```
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:

$$I^{S_1}_{current} = \{S1[t_1, t_2, t_3] : (t_1 = t_p) \wedge (t_2 = t_q) \wedge (1 \leq t_3 \leq n-1)\}$$

Determination of data to be invalidated:

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

$$D_{inflow} = r^{S_1}_{write}(I_{source}) = \{B[t_q, i_1] : 2 \leq i_1 \leq n\}$$

```
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;
    }
  }
}
```

Determination of data to be written-back:

$I_{target} = \mathcal{D}_{flow}(I^{S_1}_{current}) \setminus I^{S_1}_{current}; I_{producer} = \mathcal{D}^{-1}_{flow}(I_{target}) \cap I^{S1}_{current}$

$D_{outflow} = r^{S_1}_{write}(I_{producer})$

Last writes: writes by iterations which are not sources of any output dependences

$I_{live\_out} = I^{S_1}_{current} \setminus dom\ \mathcal{D}_{output}; D_{live\_out\_data} = r^{S_1}_{write}(I_{live\_out})$

Invalidate Set:

$$
\begin{aligned}
D^{S_1}_{writeback} =& (D_{outflow} \cup D_{live\_out\_data}) \\
=& \{B[t_q, i_1] : (t_p \leq tsteps - 2 \wedge 2 \leq i_1 \leq n-1) \vee \\
& (t_p = tsteps - 1 \wedge 1 \leq i_1 \leq n-1)\}
\end{aligned}
$$

Coherence Instructions Inserted Code

```
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.

```
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)

Experimental Evaluation

Table: Simulator parameters

Processor chip	8-core multicore chip
Issue width; ROB size	4-issue; 176 entries
Private L1 cache	32KB Write-back, 4-way, 2 cycle hit latency
Shared L2 cache	1MB Write-back, 8-way, multi-banked 11 cycle round-trip time
Cache line size	32 bytes
Cache coherence protocol	Snooping-based MESI protocol
Main Memory	300 cycle round-trip time

Table: Benchmarks

Benchmark	Description
gemm	Matrix-multiply : $C = \alpha.A.B + \beta.C$
gemver	Vector Multiplication and Matrix Addition
jacobi-1d	1-D Jacobi stencil computation
jacobi-2d	2-D Jacobi stencil computation
LU	LU decomposition
trisolv	Triangular solver
CG	Conjugate Gradient method
backprop	Pattern recognition using unstructured grid
hotspot	Thermal simulation using structured grid
kmeans	Clustering algorithm used in data-mining
pathfinder	Dynamic Programming for grid traversal
srad	Image Processing using structured grid

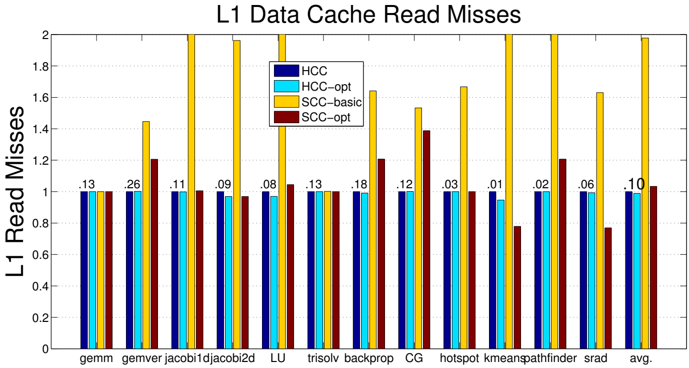


Figure: L1 data cache read misses. The L1 read miss ratios for HCC are also shown.

Experimental Evaluation

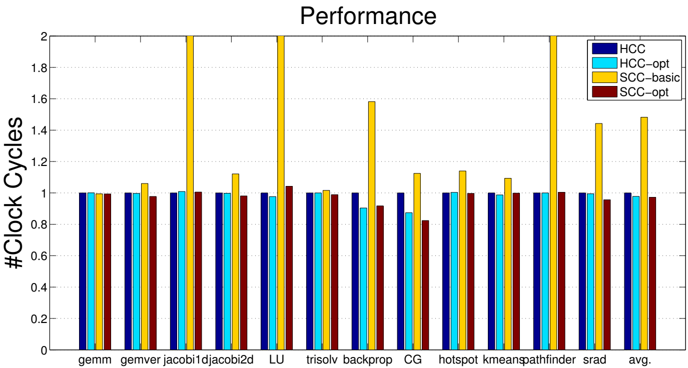


Figure: Comparison of Execution times with HCC as the baseline

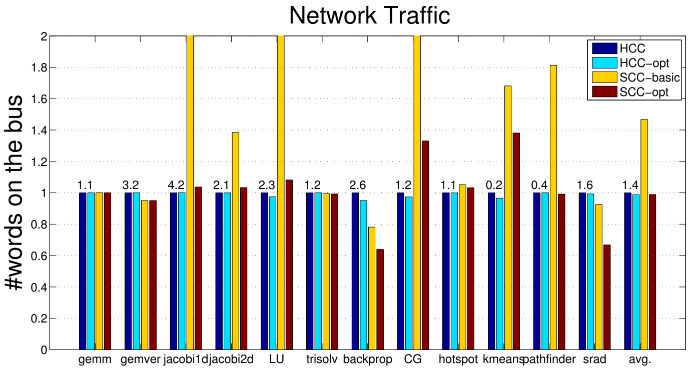


Figure: Traffic on the system bus. Average number of words per cycle for HCC is also shown.

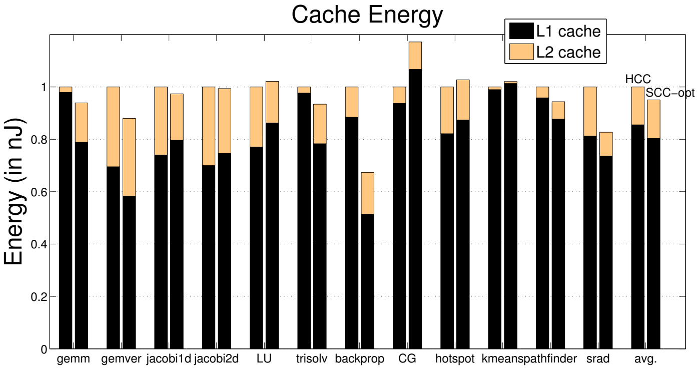


Figure: Comparison of Energy Consumption with HCC as the baseline

- 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