Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs?

Eddy Zheng Zhang
Yunliang Jiang
Xipeng Shen (presenter)

Computer Science Department
The College of William and Mary, VA, USA
Cache Sharing

• A common feature on modern CMP
Cache Sharing on CMP

- A double-edged sword
  - Reduces communication latency
  - But causes conflicts & contention
Cache Sharing on CMP

- A double-edged sword
  - Reduces communication latency
  - But causes conflicts & contention

Non-Uniformity
Many Efforts for Exploitation

• Example: shared-cache-aware scheduling
  • Assigning suitable programs/threads to the same chip

• Independent jobs
  • Job Co-Scheduling [Snively+:00, Snively+:02, El-Moursy+:06, Fedorova+:07, Jiang+:08, Zhou+:09]

• Parallel threads of server applications
  • Thread Clustering [Tam+:07]
Overview of this Work (1/3)

• A surprising finding
  • Insignificant effects from shared cache on a recent multithreaded benchmark suite (PARSEC)

• Drawn from a systematic measurement
  • thousands of runs
  • 7 dimensions on levels of programs, OS, & architecture
  • derived from timing results
  • confirmed by hardware performance counters
Overview of this Work (2/3)

- A detailed analysis
  - Reason
    - three mismatches between executables and CMP cache architecture
  - Cause
    - the current development and compilation are oblivious to cache sharing
Overview of this Work (3/3)

• An exploration of the implications
  • Exploiting cache sharing deserves not less but more attention.
  • But to exert the power, cache-sharing-aware transformations are critical
    • Cuts half of cache misses
    • Improves performance by 36%.
Outline

• Experiment design
• Measurement and findings
• Cache-sharing-aware transformation
• Related work, summary, and conclusion.
Benchmarks (1/3)

- PARSEC suite by Princeton Univ [Bienia+:08]

“focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors”
Benchmarks (2/3)

• Composed of
  • RMS applications
  • Systems applications
  • ……

• A wide spectrum of
  • working sets, locality, data sharing, synch., off-chip traffic, etc.
## Benchmarks (3/3)

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Parallelism</th>
<th>Working Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackscholes</td>
<td>Black–Scholes equation</td>
<td>data</td>
<td>2MB</td>
</tr>
<tr>
<td>Bodytrack</td>
<td>body tracking</td>
<td>data</td>
<td>8MB</td>
</tr>
<tr>
<td>Canneal</td>
<td>sim. Annealing</td>
<td>unstruct.</td>
<td>256MB</td>
</tr>
<tr>
<td>Facesim</td>
<td>face simulation</td>
<td>data</td>
<td>256MB</td>
</tr>
<tr>
<td>Fluidanimate</td>
<td>fluid dynamics</td>
<td>data</td>
<td>64MB</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>online clustering</td>
<td>data</td>
<td>16MB</td>
</tr>
<tr>
<td>Swaptions</td>
<td>portfolio pricing</td>
<td>data</td>
<td>0.5MB</td>
</tr>
<tr>
<td>X264</td>
<td>video encoding</td>
<td>pipeline</td>
<td>16MB</td>
</tr>
<tr>
<td>Dedup</td>
<td>stream compression</td>
<td>pipeline</td>
<td>256MB</td>
</tr>
<tr>
<td>Ferret</td>
<td>image search</td>
<td>pipeline</td>
<td>64MB</td>
</tr>
</tbody>
</table>
## Factors Covered in Measurements

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Variations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>benchmarks</td>
<td>10</td>
<td>from PARSEC</td>
</tr>
<tr>
<td>parallelism</td>
<td>3</td>
<td>data, pipeline, unstructured</td>
</tr>
<tr>
<td>inputs</td>
<td>4</td>
<td>simsmall, simmedium, simlarge, native</td>
</tr>
<tr>
<td># of threads</td>
<td>4</td>
<td>1,2,4,8</td>
</tr>
<tr>
<td>assignment</td>
<td>3</td>
<td>threads assignment to cores</td>
</tr>
<tr>
<td>binding</td>
<td>2</td>
<td>yes, no</td>
</tr>
<tr>
<td>subset of cores</td>
<td>7</td>
<td>The cores a program uses</td>
</tr>
<tr>
<td>platforms</td>
<td>2</td>
<td>Intel Xeon &amp; AMD Operon</td>
</tr>
</tbody>
</table>
Machines

Intel (Xeon 5310)

AMD (Opeteron 2352)
Measurement Schemes

• Running times
  • Built-in hooks in PARSEC
• Hardware performance counters
  • PAPI
  • cache miss, mem. bus, shared data accesses
Outline

• Experiment design
• Measurement and findings
• Cache-sharing-aware transformation
• Related work, summary, and conclusions
Observation I: Sharing vs. Non-sharing
Sharing vs. Non-sharing

VS.

VS.
Sharing vs. Non-sharing

VS.

VS.
Sharing vs. Non-sharing

Performance Evaluation (Intel)

- 2t simlarge
- 2t native
- 4t simlarge
- 4t native

Test Applications:
- blackscholes
- bodytrack
- canneal
- facesim
- fluidanimate
- streamcluster
- swaptions
- x264
Sharing vs. Non-sharing

Performance Evaluation (AMD)

- blackscholes
- bodytrack
- canneal
- facesim
- fluidanimate
- streamcluster
- swaptions
- x264
Sharing vs. Non-sharing

- L2-cache accesses & misses (Intel)
Reasons (1/2)

1) Small amount of inter-thread data sharing

![Bar chart showing sharing ratio of reads (%)(Intel)]

- blackscholes
- bodytrack
- canneal
- facesim
- fluidanimate
- streamcluster
- swaptions
- x264
## Reasons (2/2)

### 2) Large working sets

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Parallelism</th>
<th>Working Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackscholes</td>
<td>Black–Scholes equation</td>
<td>data</td>
<td>2MB</td>
</tr>
<tr>
<td>Bodytrack</td>
<td>body tracking</td>
<td>data</td>
<td>8MB</td>
</tr>
<tr>
<td>Canneal</td>
<td>sim. Annealing</td>
<td>unstruct.</td>
<td>256MB</td>
</tr>
<tr>
<td>Facesim</td>
<td>face simulation</td>
<td>data</td>
<td>256MB</td>
</tr>
<tr>
<td>Fluidanimate</td>
<td>fluid dynamics</td>
<td>data</td>
<td>64MB</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>online clustering</td>
<td>data</td>
<td>16MB</td>
</tr>
<tr>
<td>Swaptions</td>
<td>portfolio pricing</td>
<td>data</td>
<td>0.5MB</td>
</tr>
<tr>
<td>X264</td>
<td>video encoding</td>
<td>pipeline</td>
<td>16MB</td>
</tr>
<tr>
<td>Dedup</td>
<td>stream compression</td>
<td>pipeline</td>
<td>256MB</td>
</tr>
<tr>
<td>Ferret</td>
<td>image search</td>
<td>pipeline</td>
<td>64MB</td>
</tr>
</tbody>
</table>
Observation II: Different Sharing Cases

- Threads may differ
  - Different data to be processed or tasks to be conducted.
  - Non-uniform communication and data sharing.

- Different thread placement may give different performance in the sharing case.
Different Sharing Cases
Max. Perf. Diff (%)

- 2t simlarge
- 2t native
- 4t simlarge
- 4t native

Statistically insignificant---large fluctuations across runs of the same config.
Two Possible Reasons

- Similar interactions among threads
- Differences are smoothed by phase shifts
Temporal Traces of L2 misses

![Graph showing cache miss rates over instructions]

- Thread 1, 2 Share Cache
- Thread 1, 3 Share Cache
- Thread 1, 4 Share Cache

The College of William and Mary
Temporal Traces of L2 misses

The College of William and Mary
Two Possible Reasons

- Similar interactions among threads
- Differences are smoothed by phase shifts
Pipeline Programs

- Two such programs: ferret, and dedup
- Numerous concurrent stages
  - Interactions within and between stages
- Large differences between different thread-core assignments
- Mainly due to load balance rather than differences in cache sharing.
A Short Summary

- Insignificant influence on performance
  - Large working sets
  - Little data sharing
- Thread placement does not matter
  - Due to uniform relations among threads
- Hold across inputs, # threads, architecture, phases.
Outline

• Experiment design
• Measurement and findings
• Cache-sharing-aware transformation
• Related work, summary, and conclusions
Principle

- Increase data sharing among siblings
- Decrease data sharing otherwise

Non-uniform threads

Non-uniform cache sharing
Example: streamcluster

original code

thread 1

for i = 1 to N, step = 1

... ...

for j = T1 to T2

dist = foo(p[j], p[c[i]])

dist = foo(p[j], p[c[i]])

end

end

... ...

thread 2

for i = 1 to N, step = 1

... ...

for j = T2+1 to T3

dist = foo(p[j], p[c[i]])

dist = foo(p[j], p[c[i]])

end

end

... ...
Example: streamcluster

optimized code

thread 1

for i = 1 to N, step = 2
    ...
    ...
    for j = T1 to T3
        dist = foo(p[j], p[c[i]])
    end
    ...
end

thread 2

for i = 1 to N, step = 2
    ...
    ...
    for j = T1 to T3
        dist = foo(p[j], p[c[i+1]])
    end
    ...
end
Performance Improvement (streamcluster)

L2 Cache Miss
Mem Bus Trans
Other Programs

Normalized L2 Misses (on Intel)

4t Blackholes
8t Blackholes
4t Bodytrack
8t Bodytrack
Implication

• To exert the potential of shared cache, program-level transformations are critical.

• Limited existing explorations
  • Sarkar & Tullsen’08, Kumar & Tullsen’02, Nokolopoulos’03.

* A contrast to the large body of work in OS and architecture.
Related Work

First *systematic* examin. of the influence of cache sharing in modern CMP on the perf. of *contemporary multithreaded* apps.

- Co-runs of independent programs
  - Snavely+:00, Snavely+:02, El-Moursy+:06, Fedorova+:07, Jiang+:08, Zhou+:09, Tian+:09
- Co-runs of parallel threads of multithreaded programs
  - Liao+:05, Tuck+:03, Tam+:07
- Have been focused on certain aspects of CMP
  - Simulators-based for cache design
  - Old benchmarks (e.g. SPLASH-2)
  - Specific class of apps (e.g., server apps)
  - Old CMP with no shared cache
Summary

Measurement
Insignificant influence from cache sharing despite inputs, arch, # threads, thread placement, parallelism, phases, etc.

Analysis
Mismatch between SW & HW causing the observations.

Transformation
Large potential of cache-share-aware code optimizations.
Conclusion

Does cache sharing on CMP matter to contemporary multithreaded programs?

Yes. But the main effects show up only after cache-sharing-aware transformations.
Thanks!

Questions?