Addressing Shared Resource Contention in Multicore Processors via Scheduling

ASPLOS’10 by Sergey Zhuravlev, et al.
Simon Fraser University

Presenter: Huanchen ZHANG, Zhuo CHEN
Problem Statement

• Multicore processors were prevalent (2010)
  – Even truer today
  – Opportunity for thread level parallelism

• Scheduling among multiple cores is hard
  – Simply keep cores busy is not good enough
  – Apps may compete for shared resource (e.g. cache)

What is the best scheduling approach to deal with resource contention?
Scheduling Does Matter

Figure 1. The performance degradation relative to running solo for two different schedules of SPEC CPU2006 applications on an Intel Xeon X3565 quad-core processor (two cores share an LLC).
Why Worse Than Solo?

• Thought experiment:
  – Two apps: A: low miss rate, B: high miss rate
  – Who will suffer more when sharing cache with another application C?
  – Cache attention: C brings its own data to cache

Answer 1:
A, because B already has very high miss rate anyway.
Assumption is cache attention is the main cause of performance degradation.

Answer 2:
B, because the miss penalty is larger
Assumption is cache attention is NOT the main cause of performance degradation.
Outline: Cache-aware Scheduling

• Classification scheme
  – Classification scheme is the information you use to make a decision
  – How can we study classification scheme alone?

• Classification scheme + Scheduling policy
  – Scheduling policy is how you use the information
Outline: Cache-aware Scheduling

• Classification scheme
  – Classification scheme is the information you use to make a decision
  – **How** can we study classification scheme alone?

• Classification scheme + Scheduling policy
  – Scheduling policy is how you use the information
Study Classification Scheme Alone

- Perfect scheduling policy
Study Classification Scheme Alone

• Perfect scheduling policy

**Worst Schedule:**
Average Degradation = 22%

- MCF 90.4% MILC
- GAMESS -2.4% NAMD

**Best Schedule:**
Average Degradation = 2.3%

- MCF 4.67% GAMESS
- MILC 4.56% NAMD
Evaluating Classification Scheme

- Optimal Schedule (OS)
  - Optimal classification scheme + Perfect scheduling policy
- Estimated Best Schedule (EBS)
  - Classification scheme under evaluation + Perfect scheduling policy
- Degradation due to classification scheme

\[
\text{Relative Degradation} = \frac{\text{Degradation of EBS} - \text{Degradation of OS}}{\text{Degradation of OS}}
\]
Collecting Cache Performance data

- Stack Distance Profile

<table>
<thead>
<tr>
<th>LRU Stack</th>
<th>MRU</th>
<th>LRU</th>
<th>Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Counter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># of sets</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

associativity
Classification Schemes - SDC

• Key Idea
  – Model how two application threads compete for the LRU stack positions
Classification Schemes – Animal Classes

• 4 classes of application threads (classified based on stack distance profiles)
  – **Turtle**: low use of the shared cache
  – **Sheep**: low miss rate, insensitive to \# of cache ways
  – **Rabbit**: low miss rate, sensitive to \# of cache ways
  – **Devil**: high miss rate, tends to thrash the cache

Relative Performance Degradation Table

<table>
<thead>
<tr>
<th></th>
<th>Turtle</th>
<th>Sheep</th>
<th>Rabbit</th>
<th>Devil</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turtle</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sheep</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rabbit</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Devil</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Classification Schemes – Miss Rate

• Simply use “miss rate” as heuristics
  – Identify high miss rate application threads and separate them into different caches
  – Why?
    • exclusive cache lowers miss rate
    • exclusive prefetching HW and lowly-contended front-side bus reduces miss penalty
Classification Schemes - Pain

• Cache Sensitivity
  – How much an application will suffer due to cache contention
    \[ S = \left( \frac{1}{1+n} \right) \sum_{i=0}^{n} i \times h(i) \]

• Cache Intensity
  – How aggressively an application thread uses cache
    \[ Z = \# cache accesses per one million instructions \]

• Pain of Co-Schedule

  \[ \text{Pain}(A_B) = S(A) \times Z(B) \]

  \[ \text{Pain}(A, B) = \text{Pain}(A_B) + \text{Pain}(B_A) \]
Comparing Classification Schemes

- Workload: 10 benchmarks from SPEC2006 Suite

**Figure 3.** Degradation relative to optimal experienced by each classification scheme on systems with different numbers of cores.
Performance Degradation Factors

- **DRAM controller**
- **Front-side bus (FSB)**
- **core**
- **L2 cache**
- **Prefetching HW**
DRAM Contention

\[
\text{DRAM contention} = \frac{\text{DiffSocketPrefetchOFF} - \text{SoloPrefetchOFF}}{\text{SoloPrefetchOFF}}
\]
DRAM contention = \frac{\text{DiffSocketPrefetchOFF}}{\text{SoloPrefetchOFF}} - \text{SoloPrefetchOFF}
FSB Contention

FSB contention = \[ \frac{\text{DiffCachePrefetchOFF} - \text{DiffSocketPrefetchOFF}}{\text{SoloPrefetchOFF}} \]
FSB Contention

FSB contention = \( \frac{\text{DiffCachePrefetchOFF}}{\text{SoloPrefetchOFF}} - \frac{\text{DiffSocketPrefetchOFF}}{\text{SoloPrefetchOFF}} \)
FSB Contention

\[
\text{FSB contention} = \frac{\text{DiffCachePrefetchOFF} - \text{DiffSocketPrefetchOFF}}{\text{SoloPrefetchOFF}}
\]
Cache Contention

Cache contention = \( \frac{\text{SameCachePrefetchOFF} - \text{DiffCachePrefetchOFF}}{\text{SoloPrefetchOFF}} \)
Cache Contention

![Diagram of cache architecture]

Cache contention = \[ \frac{\text{SameCachePrefetchOFF} - \text{DiffCachePrefetchOFF}}{\text{SoloPrefetchOFF}} \]
Cache contention = \( \frac{\text{Same Cache Prefetch OFF} - \text{Diff Cache Prefetch OFF}}{\text{Solo Prefetch OFF}} \)
Total Degradation

Total Degradation = \frac{\text{SameCachePrefetchON}}{\text{SoloPrefetchON}}
Total Degradation = \frac{\text{SameCachePrefetchON} - \text{SoloPrefetchON}}{\text{SoloPrefetchON}}
Prefeching Contention

Prefeching Contention =
   Total Degradation (PF ON)
   - Cache Contention (PF OFF)
   - FSB Contention (PF OFF)
   - DRAM Contention (PF OFF)
Prefetching Contention

Prefetching Contention =
  Total Degradation (PF ON)
   - Cache Contention (PF OFF)
   - FSB Contention (PF OFF)
   - DRAM Contention (PF OFF)
Contributions of Degradation Factors

Figure 4. Percent contribution that each of the factors have on the total degradation.
Outline: Cache-aware Scheduling

• Classification scheme
  – Classification scheme is the information you use to make a decision
  – How can we study classification scheme alone?

• Classification scheme + Scheduling policy
  – Scheduling policy is how you use the information
Scheduling Algorithms

• Pick one classification scheme
  – Pain is the best (offline), but overhead is big
  – Picked miss rate

• Distributed Intensity (DI)
  – Sort based on solo miss rate
  – Goal: miss rates are distributed evenly

• Distributed intensity Online (DIO)
  – Get miss rate dynamically
Average Performance

• Intel Xeon X5365; Eight workloads
• Compare to DEFAULT (Linux scheduler)
Not much Better?

- Consider a case where
  - Four cores; two shared cache
  - Two intensive applications (high miss rate), two non-intensive applications (low miss rate)
- DI/DIO makes sure the two intensive ones don’t run together
- But the worst case only happens with 1/3 probability...
Worst-case Performance

(a) The relative performance improvement of the worst case DI and DIO over the worst case DEFAULT for Intel 8 threads.

Performance deviation

(b) Deviation of the same application in the same workload with DI, DIO and Default (low bars are good) for Intel 8 threads.
Conclusions

• Cache contention is NOT the dominant cause (?)

• Evaluated different classification schemes
  – Pain is the best; miss rate is the most practical

• Miss rate performs well in real scheduling

• Contention-aware scheduling is good for
  – Improving average performance (not so much)
  – QoS & performance isolation
Other papers...

• Fine grained scheduling
  – Software scheduling overhead is too high, hardware?

• Contention for shared resource (critical section)
  – Optimizing for locks

• Scheduling at Clusters
  – A node has multiple VMs, a VM has multiple threads