Understanding Application Sensitivities: The Key to Shared Resource Modeling

David Black-Schaffer
Assistant Professor, Department of Information Technology
Uppsala University

Special thanks to the Uppsala Architecture Research Team:
David Eklöv, Prof. Erik Hagersten, Nikos Nikoleris, Andreas Sandberg, Andreas Sembrant
Multicore Memory Systems

Intel Nehalem Memory Hierarchy (3GHz)

Latency to private L1: 4 cycles

Latency to DRAM: 200 cycles

Bandwidth to Private L1: 15 bytes/cycle per core

Bandwidth to DRAM: 4-8 bytes/cycle total
1-2 bytes/cycle per core

Impact of Resource Sharing

Throughput

30-40% slowdown due to sharing

bzip2  |  Ibm  |  libquantum  |  gamess
---|---|---|---
Alone: 1.2 | Mixed Workload: 0.8 | Alone: 1.5 | Mixed Workload: 1.8

David Black-Schaffer  
Uppsala University / Department of Information Technology  
12/10/2012
Measuring Shared Resource Sensitivity

1. Cache Pirate
   - Measuring sensitivity to shared cache allocation
   - General technique for measuring sensitivity in real HW/SW

2. Bandwidth Bandit
   - Measuring sensitivity to shared bandwidth allocation

3. Modeling Cache Usage
   - Predicting shared cache allocation and performance impact
   - Use Cache Pirate data to include HW/SW complexities
1. Cache Pirating (David Eklöv)

- Measure **cache sensitivity** by stealing cache
  - **Steal cache** with a “Pirate” application
  - **Measure performance** of the Target
  - **Monitor the Pirate** to verify cache stolen

- **Shared Cache**

  Captures performance data for all cache sizes in one run.

- **Graph:**
  - **Cache Size:** 0MB, 8MB
  - **Time:**
    - **Target**
    - **Pirate**

- **If the Pirate misses** in the cache then we aren’t stealing what we want.

- **5% Overhead**
  - Accurate: Includes all HW/SW effects
Application Cache Sensitivities

482.sphinx3

- Miss Ratio
- Fetch Ratio

0% 2% 4%

F/M Ratio

BW

Cache size

1M 3M 5M 7M

Effect of hardware prefetching.

Performance maintained by increasing bandwidth usage.

470.lbm

1M 3M 5M 7M
Predicting Multicore Scaling (Cache)

**Experiment**
- Run 1-4 independent instances of the same program on a 4-core Nehalem
- **Performance affected by shared cache**
  - ¼ of the shared cache → 20% slower

Profile performance as a function of shared cache
⇒ Predict multicore scalability
Bandwidth Limits

- Bandwidth as a function of shared cache size

4 cores @ 3GB/s = 12GB/s
System max = 10.4GB/s

...until you run out of shared bandwidth.

No performance loss with less cache space...

50% reduction in cache → 57% increase in bandwidth
2. Bandwidth Bandit (David Eklöv)

- Measure bandwidth sensitivity by stealing bandwidth
- More complex than Cache Pirating
  - Memory controller (access patterns, row buffers, re-ordering, page allocation)
  - Latency and throughput sensitivities

Insight: No correlation between bandwidth usage and sensitivity to bandwidth contention!
Application Bandwidth Sensitivities

- **Latency sensitive:** Slowdown before BW is saturated.
- **Bandwidth sensitive:** Slowdown when BW is saturated.

- Significant variation in application sensitivities
- Leads to different impact of resource sharing
Predicting Multicore Scaling (BW)

• Run 4 instances of the same application
  (Page coloring to guarantee no cache interference. Each gets exactly ¼ of the cache; BW effects only)

Profile performance as a function of shared bandwidth
⇒ Predict multicore scalability
USING SENSITIVITY MEASUREMENTS TO MODEL SHARING
Modeling Cache Usage (Andreas Sandberg)

• Use Pirate data to model caches
• Caches have inflows and outflows
  – At steady state these are equal
  – Relative flow rates are proportional to the content of the cache

App 1
App 2
Rates from Pirate Data

Evictions

LRU cache has “sticky” data that is accessed frequently enough to keep it in the cache.
Pirate Data → Cache Contents

- **Fetch Rate**
  - Z₀⁻
  - Z₀
  - Z₁⁻
  - Z₁
  - C

- **Hit Rate**
  - Z₀⁻
  - Z₀
  - Z₁⁻
  - Z₁
  - C

- **Sticky Data**
  - Size
  - Z₀
  - Z₁
  - C

**Notes:**
- Gives us the data set sizes
- Gives us the intensity of reuse in each data set
- Have sticky data sizes and reuse intensities

**Equation 4:**
\[ s \]
Fighting for Space in the Cache

- We know the sizes and access intensities of each application’s data sets.
- Applications fight for cache space based on intensity of access (LRU).
- The data with the greatest access intensities stays in the cache.
- If the data set won’t fit, then it’s reuse effectively goes away (data is evicted before it is used again).
Predicting Shared Cache Usage

**Background & Motivation**

Modeling Cache Sharing

**Evaluation**

Cache Size Prediction

LRU, Pairs, Simulator

<table>
<thead>
<tr>
<th>Modeled Size [MB]</th>
<th>Simulated Size [MB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>5% Error --- 10% Error</td>
<td></td>
</tr>
</tbody>
</table>

Pairs

- **0.9% Average Error**

Groups of 4

- **1.3% Average Error**

**Simulator LRU**
Individual profiles of performance as a function of shared cache ➔ Predict workload scalability
APPLICATIONS TO TASK-BASED RUNTIMES
Understanding: Waste Cores

Task A Performance

Task B Performance

Energy?

Percent of Shared Cache

Energy?

Percent of Shared Cache

CPU 0
A

CPU 1
B

CPU 2
B

CPU 3
B

CPU 0
A

CPU 1
A

CPU 2
B

CPU 3
B

CPU 0
A

CPU 1
A

CPU 2
A

CPU 3
B

Energy?
Adapting: Run Bad Code

**Good Task Performance**

Energy?

**Bad Task Performance**

Energy?

### Percent of Shared Cache

- **Bandwidth:**
  - 0%
  - 25%
  - 50%
  - 75%
  - 100%

### Percent of Bandwidth

- **Energy?**
  - CPU 0: Good
  - CPU 1: Good
  - CPU 2: Good
  - CPU 3: Good

### Energy?

- CPU 0: Good
- CPU 1: Bad
- CPU 2: Bad
- CPU 3: Total time is shorter because we have three tasks doing the work.
Putting it Together

• **Profile tasks**
  – Need to run individually
    (Probably can’t use other cores at the same time)
  – Can cache results for future runs

• **Predict performance**
  – Decide which tasks to run together

• **Adapt tasks**
  – Compiler/runtime interaction

• **Problems:**
  – Tasks sized for private caches → no shared resource use
  – Homogeneous tasks → little opportunity for scheduling
  – Combinatorial explosion → too many scheduling choices
Acknowledgements

• **PhD Students:**
  – David Eklöv
    • StatStack, StatCC, Cache Pirate
  – David Eklöv and Nikos Nikoleris
    • Bandwidth Bandit
  – Andreas Sandberg
    • Cache Pollution, Cache Sharing Models
  – Andreas Sembrant
    • Phase Detection, Phase Memory Modeling

• **Colleagues:**
  – Erik Hagersten
  – Stefanos Kaxiras
QUESTIONS?
PHASES
with other applications' memory accesses. For example, when running astar/lakes and bwaves from SPEC CPU2006, we measure the slowdown of an application pair due to cache contention, since an application's performance depends on how its memory accesses are interleaved with other applications. We show that our method can predict application performance for the same application pair. The variability occurs due to different overlappings of application phases that occur when they are offset in time. As runs. This variability is important for generating an accurate view of the performance interactions and bandwidth demands of co-running applications by quickly evaluating hundreds of overlappings. The impact of cache sharing on application performance has been shown to be critical resources for performance measurements. Shared caches in contemporary multicores have repeatedly been shown to be critical resources for performance measurements.

The main contributions of this paper are: An extension to a statistical cache-sharing model developed by Sembrant et al. of each application running in isolation. This means that only a small number of profiling runs need to be done for performance measurements. This is a both time- and resource-efficient way to measure performance in a system.

Population [%]

Average

Slowdown [%]

0 5 10 15 20

0 2 7.7 17

2% slowdown 17% of the time

16% slowdown 15% of the time

Program behavior is not constant.
Application Phases (Andreas Sembrant)

- Applications have time-varying behavior
- Need phase information for accurate insight
- With phase information we can do smarter profiling

Motivation

Background

Related Work

ScarPhase

Phase Guided Profiling

Average Bandwidth

Average CPI

Cycles per Instructions

Branch Miss Predictions
Phase Detection: ScarPhase

ScarPhase: Sample-based Classification and Analysis for Runtime Phases

- Online (while the program is profiled)
- **2% overhead** via hardware performance counters (Intel PEBS)
Efficient Data Collection with Phases

Application Execution

Profile Phases

Pirate/Bandit

Profile

Average Bandwidth

Time in Billions of Instructions
Better *and* Faster with Phases

- Faster and more accurate
- Easier to use: adapts to complexity of application
Phases Key for Insight

StatStack (cache modeling) with phases 20% overhead
Phases in Parallel Applications

More threads $\rightarrow$ shorter phases $\rightarrow$ harder to optimize DVFS, cache size, etc.
The Big Picture

- **Individual task profiles enable:**
  - **Performance prediction** for co-execution (development)
  - **Efficient scheduling** for resource contention (runtime)
Future Work

• Phases + Pirate + Bandit
  – Lower-overhead profiling & more detailed information

• Sharing Model + Bandit
  – Remove the “unlimited” bandwidth assumption

• Sharing Model + Phases
  – Understand execution alignment (variability)

• Understanding the cause of slowdowns
  – Cache? Bandwidth? Synchronization?

• Plus power...
SENSITIVITY IN MORE DETAIL
More Detail: Cache Sharing

- Three SPEC benchmark applications
- Different behaviors due to different properties
- Each will respond differently to cache sharing

<table>
<thead>
<tr>
<th>Bandwidth (GB/s)</th>
<th>Performance (lower better)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1M</td>
</tr>
<tr>
<td>Perf:</td>
<td></td>
</tr>
<tr>
<td>BW:</td>
<td>0%</td>
</tr>
</tbody>
</table>

- 0% slower
- 50x bandwidth increase
- 50% slower
- 10x bandwidth increase
- 0% slower
- 2x bandwidth increase

- Full working set fits in the cache
- Most of the working set fits in the cache
- Little of the working set fits in the cache

- Insensitive to latency
- Sensitivity to latency
- Insensitive to latency

Fig 8.8 Performance (CPI, bandwidth requirements (BW), and fetch/miss ratios (F9M) for several benchmarks. This data was collected with hardware.
More Detail: The Impact of Prefetching

- Different degrees of prefetching depending on application access pattern and hardware
- Prefetching reduces application sensitivity to latency

Difference between fetches and misses is prefetch rate.

- **No Prefetching**
  - Miss Ratio
  - Fetch Ratio

- **Minimal Prefetching**
  - CPI
  - Bandwidth

- **Significant** prefetching, and it benefits from it
  - CPI
  - Bandwidth
More Detail: Cache Pollution

• We can measure how **greedy** an application is and how **sensitive** it is
• By changing the code to use non-caching instructions we can make an application less greedy without hurting performance
More Detail: Bandwidth Sharing

- Sensitivity is a function of the application
  - **Latency sensitivity** (memory level parallelism)
  - **Bandwidth requirement** (data rate)
- And the hardware
  - Ability to handle out-of-order requests (queue sizes)
  - Access pattern costs (streaming vs. random in DRAM banks)
- **BW consumption is not a good indicator of BW sensitivity**

![Diagram showing slowdown and bandwidth consumption for different applications](image)

**Slowdown at 90% of saturation bandwidth**