Characterizing and Optimizing the Performance of Multithreaded Programs Under Interference

Yong Zhao, Jia Rao
The University of Texas at Arlington

Qing Yi
University of Colorado Colorado Springs

PACT 2016@Haifa, Israel
Cloud Computing

Cloud computing, powered by warehouse-scale data centers, provides users with abundant parallelism and potentially unlimited scalability.

- Characteristics
  - Virtualization
  - Scalability and elasticity
  - Economy of scale
  - Multi-tenancy and workload consolidation
# Ideal for Running Parallel Programs

## General Purpose

<table>
<thead>
<tr>
<th>Model</th>
<th>vCPU</th>
<th>Mem (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2.nano</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>t2.micro</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>t2.small</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>t2.medium</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>t2.large</td>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

(Use Cases: small databases, data processing tasks, cluster computing etc.)

## Compute Optimized

<table>
<thead>
<tr>
<th>Model</th>
<th>vCPU</th>
<th>Mem (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2.nano</td>
<td>2</td>
<td>3.75</td>
</tr>
<tr>
<td>t2.micro</td>
<td>4</td>
<td>7.5</td>
</tr>
<tr>
<td>t2.small</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>t2.medium</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>t2.large</td>
<td>36</td>
<td>60</td>
</tr>
</tbody>
</table>

(Use Cases: HPC applications, batch processing, distributed analytics etc.)

## Memory Optimized

<table>
<thead>
<tr>
<th>Model</th>
<th>vCPU</th>
<th>Mem (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1.32xlarge</td>
<td>128</td>
<td>1952</td>
</tr>
</tbody>
</table>

(Use Cases: HPC applications, Apache Spark, Presto etc.)

— From [https://aws.amazon.com/ec2/instance-types/]
**Parallel Performance in the Cloud**

**Suboptimal** and **unpredictable** performance due to multi-tenant interference

- Shared resources (e.g., LLC, Mem)
- CPU multiplexing
- Memory footprint, access pattern ...
- Fair sharing algorithm ...
- Slowdown on individual threads
- Sync methods, load balancing
- Overall slowdown

---

**Normalized slowdown**

- Alone
- w/ streamcluster
- w/ fluidanimate

<table>
<thead>
<tr>
<th>Program</th>
<th>Alone</th>
<th>w/ streamcluster</th>
<th>w/ fluidanimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>streamcluster</td>
<td>1.0</td>
<td>4.3x</td>
<td></td>
</tr>
<tr>
<td>canneal</td>
<td>1.0</td>
<td>2.8x</td>
<td></td>
</tr>
<tr>
<td>facesim</td>
<td>1.0</td>
<td>1.3x</td>
<td></td>
</tr>
<tr>
<td>fluidanimate</td>
<td>1.0</td>
<td>1.3x</td>
<td></td>
</tr>
<tr>
<td>swaptions</td>
<td>1.0</td>
<td>1.3x</td>
<td></td>
</tr>
<tr>
<td>dedup</td>
<td>1.0</td>
<td>1.3x</td>
<td></td>
</tr>
<tr>
<td>raytrace</td>
<td>1.0</td>
<td>1.3x</td>
<td></td>
</tr>
<tr>
<td>lu</td>
<td>1.0</td>
<td>1.3x</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>1.0</td>
<td>1.3x</td>
<td></td>
</tr>
</tbody>
</table>

---

**Interference**

- Parallel program

---

**Sync methods, load balancing**

---

**Overall slowdown**

---

**Memory footprint, access pattern ...**

---

**Shared resources (e.g., LLC, Mem)**

---

**CPU multiplexing**

---

**Fair sharing algorithm ...**

---

**Slowdown on individual threads**

---

**Sync methods, load balancing**

---

**Overall slowdown**

---
Related Work

• Reducing sync delays
  - BWS [USENIX ATC’ 14]
  - Demand-based coordinated scheduling [ASPLOS’ 13]
  - Balancing scheduling [EuroSys’ 13]
  - Adaptive scheduling [HPDC’ 11]
  - Lock-aware scheduling [VM’ 04]
  - Relaxed co-scheduling [VMWare]

• Modeling shared resource contention
  - numerous excellent work: Qureshi et al., [MICRO’06], Suh et al., [HPCA’02],
    Tam et al., [ASPLOS’09], Chandra et al., [HPCA’05], Guo et al., [SIGMETRICS’06],
    Iyer et al., [SIGMETRICS’07], Moscibroda et al., [USENIX Security’07], Mutlu et
    al., [MICRO’07, ISCA’08], Blagodurov et al., [TOCS’10], Xiang et al., [ASPLOS’13]
    …
Our Focus

Why parallel performance is so difficult to reason about under interference?

Parallel programs are complex
- Parallel models
- Synchronization methods
- User-level work assignment
- Inter-thread data sharing

Interference is dynamic
- Contention on the memory hierarchy
- Contention on CPU cycles
- Interplays between the two

We focus on studying how parallel programs respond to interference
Synthetic Interferences

- The effects of interference
  - Slowdown individual parallel threads
  - Cause asynchrony among threads

- Synthetic interferences to abstract these effects
  - Slowdown threads -> reducing CPU allocations
  - Asynchrony -> stoping threads at different times

- Synthetic Interferences: **zero** memory footprint, **controllable**

  - **Persistent**: simple while(1) CPU hog
  - **Periodic**: demands CPU at regular intervals otherwise stays idle (10ms busy - 10ms idle)
  - **Intermittent**: demands CPU at irregular intervals (busy periods randomly selected from 1ms to 40ms, idle periods match the busy periods at each interval)
Profiling Parallel Performance

• Decomposing parallel runtime
  
  • **compute** - computation + memory access time
  
  • **sync** - blocking or spinning time
  
  • **steal** - time that the multi-tenant system is serving other users

  \[
  \text{Parallel runtime} = \text{compute} + \text{sync} + \text{steal}
  \]

• Recording the performance events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YIELD</td>
<td>Voluntary yield to other vCPUs due to idling</td>
</tr>
<tr>
<td>PREEMPT</td>
<td>Involuntary preemption by the hypervisor</td>
</tr>
<tr>
<td>IDLE</td>
<td>The time pCPU in the idle state</td>
</tr>
<tr>
<td>HARDWARE</td>
<td>Hardware performance counters (MPKI/L2 Cache Miss/</td>
</tr>
<tr>
<td>Statistics</td>
<td>L3 Reference etc.)</td>
</tr>
</tbody>
</table>
Methodology

Differential analysis

- Compare the parallel runtime breakdown under interference with that in an interference-free environment to identify the causes of performance degradation.
Varying CPU Allocation

**Foreground:** four-thread parallel programs

**Background:** single-thread persistent interference

\[
\frac{\text{steal}}{\text{compute}} = 1
\]

Fair CPU allocation between workloads

\[
\frac{\text{steal}}{\text{compute}} < 1
\]

More resilient to interference

- Common in popular hypervisors: Xen and KVM
- Due to vCPU preemption and prioritization
- Affect parallel programs with blocking synchronizations

- Steal time should be considered to understand parallel performance in shared systems
- Steal time can be managed by controlling preemption
Compute Time Changes under Interference

Foreground: four-thread parallel programs
Background: single-thread persistent interference

Parallel runtime breakdown

Compute = computation + data access time

(compute time dropped by as much as 50%)
Reduced Data Access Time

(a) OFFCORE_STALL
(b) MPKI
(c) L3 reference
(d) L2 cache misses

Lessons learned

✓ Memory access time increases due to inter-program contentions on shared resources.
✓ Memory cost can drop due to alleviated intra-program contentions, e.g., less coherence misses
Varying Memory Cost Under Interference

Changes to Offcore Stalls (%)

-50  -25  0  25  50  75  100

Increase in memory cost: loss of locality

Reduction in memory cost: mitigation of intra-program contentions

<table>
<thead>
<tr>
<th></th>
<th>No inter.</th>
<th>w/ inter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU%</td>
<td>400%</td>
<td>363% (-9.2%)</td>
</tr>
<tr>
<td>Runtime</td>
<td>1004s</td>
<td>914s (+9.0%)</td>
</tr>
</tbody>
</table>

Case study: better sp performance with less resources

Lessons learned

- Memory access cost changes under interference in most parallel applications
- Avoid inter-program contention and exploit the mitigation of intra-program contention in workload consolidation
- Compute time must be closely monitored to understand parallel performance
Complex Interactions with the Scheduler (1/2)

Foreground: four-thread parallel programs with blocking sync
Background: four-thread interferences

Different applications exhibit different degree of degradations under the same type of interference

Lessons learned:

✓ Reducing the number of preemptions would help improve performance under interference.
✓ System idle time is a good indicator of scheduling efficiency.
Complex Interactions with the Scheduler (2/2)

**Foreground:** four-thread parallel programs with spinning sync

**Background:** four-thread interferences

Lessons learned:

- ✓ Fine-grained scheduling helps stop spinning vCPUs in a timely manner so that the overall sync time is reduced
- ✓ Out-of-sync execution due to intermittent interference helps reduce memory access time
Online Performance Prediction

- **Objective:** predict the overall slowdown before completing the program.

- **Method:** sample parallel execution under contention and compare it to a reference profile.

- **Design:** compare the amount of *useful work* done in two samples assuming an ideal memory system with zero latency and perfect load balancing.

\[
t_{\text{ideal}} = t_{\text{total}} - t_{\text{steal}} - t_{\text{sync}} - t_{\text{mem}} \\
\text{slowdown} = t'_{\text{ideal}} / t_{\text{ideal}}
\]

- \( t_{\text{total}} \) is the sampling period and \( t_{\text{steal}} \) can be directly measured
- \( t_{\text{mem}} \) can be approximated by OFFCORE_STALL on Intel processors
- \( t_{\text{sync}} \) due to spinning is accounted using our BPI-based spin detection [PPoPP’14]
- \( t_{\text{sync}} \) due to blocking is the sum of blocked time and context switch costs
Prediction Accuracy

Prediction is based on the sum of metrics on all threads

- Closer to zero is better!
- 4.5% mean absolute percentage error (MAPE)

Not accurate for programs with frequent phase changes
Performance Optimizations

**Insight-1**: uncontrolled preemption is a major source of unpredictability and inefficiency

**Delayed Preemption (DP)**: minimize pre-mature/involuntary preemptions

**Insight-2**: memory-bound programs benefit from out-of-sync execution

**Differential Scheduling (DS)**: intentionally create out-of-sync execution with differential time slices
Delayed Preemption

- **Objective:** Maximize execution efficiency by minimizing idle time

- **Method:** temporarily delays a wakeup vCPU in the hope that the current running vCPU would voluntarily yield CPU

- **Design:** the selection of preemption delay
  - static delay: 2ms, 8ms.
  - adaptive delay: adjusting the preemption delays according to system idle time
Delayed Preemption - Results

(a) Co-run with streamcluster

(b) Co-run with fluidanimate

Adaptive-DP outperforms Xen by 12% in overall performance and runtime variation is minimized.
Differential Scheduling

- **Objective:** Mitigate intra-program contention on the memory hierarchy and reduce wasteful spin time

- **Method:** create out-of-sync execution on different CPUs by assigning them different time slices

- **Design:** randomly select time slices from [10ms, 30ms] but ensure that the mean time slices on different CPUs are the same
Differential Scheduling - Results

(a) Persistent
(b) Periodic
(c) Corun with SP

Lower is better!

On average, DS outperforms Xen by 32% and significantly reduces runtime variations.
Conclusions

- **Objective:** uncover the causes of performance degradation and unpredictability of multi-threaded parallel programs in consolidated systems

- **Method:** synthetic interference + parallel runtime breakdown + differential analysis

- **Findings:**
  - Existing CPU scheduling algorithms fail to predictably and fairly allocate CPU time to programs with various demand patterns
  - The behaviors of parallel programs as a whole change under interference, e.g., placing varying pressure on the memory hierarchy or having changing overall CPU demands

- **Results:**
  - Identify the invariant in parallel runtime to perform **online** prediction
  - Propose two scheduling optimizations: **Delayed Preemption** and **Differential Scheduling**
Acknowledgement

Questions?