Deep-dive Analysis of the Data Analytics Workload in CloudSuite

Ahmad Yasin - Intel Corporation & University of Haifa
Avi Mendelson - Technion, Yosi Ben-Asher - University of Haifa
Preface

• ISCA 2013 Analysis Methodologies Tutorial
  - [link]

• A workload: CloudSuite [1]
  - Scale-out apps: Data Serving, **Data Analytics**, Media Streaming, Web etc
  - Different Characteristics:
    - Higher i-cache misses
    - Lower ILP and MLP
    - Bigger working sets
    - Low Memory BW and sharing
  - No root-cause

• A tool: Top Down Analysis [2]
  - A structured, accurate and fast method for critical bottleneck identification in out-of-order cores


Ahmad Yasin -- Deep-dive Analysis of the Data Analytics Workload in CloudSuite (IISWC 2014)
Motivation

- Exponential data growth
- Massively-parallel hardware systems
- Orchestration software layers
  - Hadoop, Spark
- New scale-out applications
  - Store and process big data
  - Different Characteristics
- No understanding of the root causes
- Data Analytics (key for big data → value)

Small improvement at a compute engine
→ large impact on datacenter
Scope

• Data Analytics (aka BDA)
  - Of-the-shelf setup from CloudSuite 2.0
  - Utilizes popular packages: Hadoop, Mahout
  - In-memory DB → CPU Bound
  - Balanced compute and memory demands
  - Blessed by other works: HiBench, CMU [4]

• Single-workload single-machine
  - Intentional to permit deep understanding
  - Proof-by-optimization approach
  - Future work: multi-node setup

Agenda

✓ Introduction
  • The workload
  • Threefold Analysis
  • Findings
  • Vs other workloads
  • Conclusions
# Experimental Setup

- **Hadoop**
  - 16 mappers
  - 2 reducers
  - 2GB heap/job
- **3 JVMs**
- **CPU**
  - Keep unused cores intact
  - Turbo enabled
- **Each result is average of 3 runs**

<table>
<thead>
<tr>
<th>Hardware CPU</th>
<th>uarch</th>
<th>Intel Xeon E5-2697 v2, Ivy Bridge μarch, 30MB LLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>2.7 GHz (Turbo→3.5)</td>
<td></td>
</tr>
<tr>
<td># sockets/ cores/ threads</td>
<td>2 / 12 / 1 or 2 (threads)</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>1600 MHz. Max BW 60 GB/s</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software OS</th>
<th>Centos 6.5, Kernel 2.6.32</th>
</tr>
</thead>
<tbody>
<tr>
<td>JVM</td>
<td>Oracle HotSpot JDK 6u29</td>
</tr>
<tr>
<td></td>
<td>OpenJDK IcedTea6 1.13.0pre</td>
</tr>
<tr>
<td></td>
<td>IBM J9 2.4</td>
</tr>
<tr>
<td>Hadoop</td>
<td>Version 0.20.2</td>
</tr>
<tr>
<td>Mahout</td>
<td>2GB Java heap per job</td>
</tr>
<tr>
<td></td>
<td>Version 0.6, Naive Bayes algo.</td>
</tr>
</tbody>
</table>
The Data Analytics Workload

- Map-Reduce model
- Classifies Wikipedia pages into categories using Mahout Bayesian classification
- 32 data partitions distributed by Hadoop, map is dominant
- Negligible system impact
  - OS, I/O, Hadoop system
Ahmad Yasin -- Deep-dive Analysis of the Data Analytics Workload in CloudSuite (IISWC 2014)
Agenda

✓ Introduction
✓ The workload
✓ Threefold Analysis
  • Findings
    - System
    - Application
    - Architecture
  • Vs other workloads
  • Conclusions
System

- Characterization Metrics
  - System: CPU Utilization, Effective CPUs, Mem BW
  - Core: IPC, UPI (Uops Per Instruction)
  - Memory: MLP, Off-core Bound

- JVM – generated code efficiency is critical
  - System: CPU over utilization due to GC scheduling
  - Core: inefficient instruction selection

<table>
<thead>
<tr>
<th>JVM type</th>
<th>Speed up</th>
<th>CPU Utilization</th>
<th>Effective CPUs</th>
<th>IPC</th>
<th>Off-core Boun</th>
<th>Miss ratio</th>
<th>UPI</th>
<th>MS Switc hes</th>
</tr>
</thead>
<tbody>
<tr>
<td>HotSpot (Baseline)</td>
<td>1.43x</td>
<td>77%</td>
<td>12.4</td>
<td>1.17</td>
<td>27%</td>
<td>12%</td>
<td>1.03</td>
<td>5%</td>
</tr>
<tr>
<td>IBM J9</td>
<td>1.38x</td>
<td>91%</td>
<td>14.6</td>
<td>1.28</td>
<td>24%</td>
<td>9%</td>
<td>1.02</td>
<td>6%</td>
</tr>
<tr>
<td>OpenJDK</td>
<td>1.00x</td>
<td>128%</td>
<td>20.5</td>
<td>0.77</td>
<td>12%</td>
<td>9%</td>
<td>1.40</td>
<td>25%</td>
</tr>
</tbody>
</table>
Call-stacks

Application Level

Examine where application’s most time is spent

Ahmad Yasin -- Deep-dive Analysis of the Data Analytics Workload in CloudSuite (IISWC 2014)
Application

• Issue:
  - WordCount is performed for each category!
  - Severely harms big caches

• Optimization:
  - Hoist WordCount loop
  - 50% speedup
    - Enabled LLC data reuse
    - LLC misses reduced by 2x, Miss Ratio: 12% → 7%

• Hadoop Applications are widely inefficient
  - CMU[4] surveyed apps of 3 Hadoop scientific setups, and reported large inefficiencies. Mahout was actually the “most optimized”

Listing 1: Main classification loop pseudo-code

```java
(1) Label classifyDocument(document) {
    label = default_label;
    (3) foreach (category : categories) {
        hash = new HashMap<String, int>;
        (5) foreach (word : document) {
            hash.update(word, 1);
        }
        (7) result = 0;
        (9) foreach (pair-of [word, frequency] : hash) {
            result += frequency * featureWeight(category, word);
        }
        (11) if (result is a maximum) {
            label = category;
        }
        (15) return label;
    }
    where featureWeight is:
(17) double featureWeight(label, word) {
    (18) return - log((getW("weight", label, word) + getW("params","oi")) / (getW("labelWeight", label) + getW("sumWeight", "vocabCount")));
```

Ahmad Yasin -- Deep-dive Analysis of the Data Analytics Workload in CloudSuite (IISWC 2014)
Top Down Performance Analysis

IPC of ~1.2

10% are MS uops

Bottleneck #1: Memory $\rightarrow$ L3 Bound

Bottleneck #2: Core Bound $\rightarrow$ Divider

Bottleneck #3: Bad Speculation $\rightarrow$ Mispredictions

Bottleneck #3: Mispredictions $\rightarrow$ Branch Resteers


Ahmad Yasin -- Deep-dive Analysis of the Data Analytics Workload in CloudSuite (IISWC 2014)
### Per hotspot drill down

#### Level #1

<table>
<thead>
<tr>
<th>Function / Call Stack</th>
<th>% time</th>
<th>IPC</th>
<th>Retiring</th>
<th>BadSpeculation</th>
<th>Backend Bound</th>
<th>Frontend Bound</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>BayesAlgorithm::apply</td>
<td>24.8%</td>
<td>1.09</td>
<td>18%</td>
<td>5.1%</td>
<td>60%</td>
<td>17%</td>
<td>[Compiled Java code]</td>
</tr>
<tr>
<td>HashMap::indexOfKey</td>
<td>23.0%</td>
<td>1.15</td>
<td>34%</td>
<td>5.2%</td>
<td>46%</td>
<td>17%</td>
<td>[Compiled Java code]</td>
</tr>
<tr>
<td>shMap::indexOfInsert</td>
<td>16.2%</td>
<td>1.27</td>
<td>42%</td>
<td>15.5%</td>
<td>21%</td>
<td>26%</td>
<td>[Compiled Java code]</td>
</tr>
<tr>
<td>HashMap::get</td>
<td>3.1%</td>
<td>1.11</td>
<td>30%</td>
<td>15.4%</td>
<td>30%</td>
<td>27%</td>
<td>[Compiled Java code]</td>
</tr>
<tr>
<td>..Datastore::getWeight</td>
<td>3.1%</td>
<td>1.18</td>
<td>41%</td>
<td>0.8%</td>
<td>45%</td>
<td>14%</td>
<td>[Compiled Java code]</td>
</tr>
</tbody>
</table>

#### Level #2

<table>
<thead>
<tr>
<th>BASE</th>
<th>Microcode Sequencer</th>
<th>BadSpeculation</th>
<th>Memory Bound</th>
<th>Core Bound</th>
<th>Frontend Latency</th>
<th>Frontend Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>18%</td>
<td>4%</td>
<td>5%</td>
<td>51%</td>
<td>34%</td>
<td>13%</td>
<td>4%</td>
</tr>
<tr>
<td>35%</td>
<td>3%</td>
<td>5%</td>
<td>32%</td>
<td>44%</td>
<td>7%</td>
<td>10%</td>
</tr>
<tr>
<td>49%</td>
<td>4%</td>
<td>16%</td>
<td>14%</td>
<td>43%</td>
<td>12%</td>
<td>14%</td>
</tr>
</tbody>
</table>

#### Level #3.1

<table>
<thead>
<tr>
<th>L1 Bound</th>
<th>L2 Bound</th>
<th>L3 Bound</th>
<th>MEM Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>11%</td>
<td>16%</td>
<td>13%</td>
<td>11%</td>
</tr>
<tr>
<td>23%</td>
<td>17%</td>
<td>15%</td>
<td>16%</td>
</tr>
<tr>
<td>20%</td>
<td>23%</td>
<td>21%</td>
<td>21%</td>
</tr>
<tr>
<td>18%</td>
<td>19%</td>
<td>25%</td>
<td>37%</td>
</tr>
</tbody>
</table>

#### Level #3.2

<table>
<thead>
<tr>
<th>FP X87</th>
<th>FP Scalar</th>
<th>FP Vector</th>
<th>Branch Mispredicts</th>
<th>Machine Clears</th>
<th>ICache Misses</th>
<th>Branch Resteers</th>
<th>DSB Switches</th>
<th>Bandwidth DSB</th>
<th>Bandwidth MITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7%</td>
<td>1%</td>
<td>0%</td>
<td>5%</td>
<td>0%</td>
<td>0%</td>
<td>2%</td>
<td>1%</td>
<td>16%</td>
<td>8%</td>
</tr>
<tr>
<td>5%</td>
<td>0%</td>
<td>0%</td>
<td>5%</td>
<td>0%</td>
<td>0%</td>
<td>3%</td>
<td>1%</td>
<td>36%</td>
<td>3%</td>
</tr>
<tr>
<td>3%</td>
<td>0%</td>
<td>0%</td>
<td>15%</td>
<td>0%</td>
<td>0%</td>
<td>4%</td>
<td>0%</td>
<td>41%</td>
<td>0%</td>
</tr>
</tbody>
</table>
Example#1: Cache misses in Hashing

- **Java:**
  - final int length=table.length;

- **Bytecode:**
  - *getfield table
  - *arraylength

- **ASM:** 4.9% of time spent in two loads:
  - mov r13d, dword ptr [r12+r9*8+0x2c]
  - mov ecx, dword ptr [r12+r13*8+0xc]

- **Fix → speedup**
  - SW workaround to cache a copy of table’s length
  - 5% app-level speedup

---

**Listing 2: OpenObjectIntHashMap original source code**

```java
def index0fKey(T key) {
    int length = table.length;
    int hash = key.hashCode() & 0x7FFFFFFF;
    int i = hash % length;
    int decrement = hash % (length - 2);
    if (decrement == 0)
        decrement = 1;
    while ((state[i] != FREE) && ... ) {
        i -= decrement;
        if (i < 0)
            i += length;
    }
    if (state[i] == FREE)
        return -1;
    return i;
}
```

---

<table>
<thead>
<tr>
<th>System</th>
<th>Memory</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimizatio\nsetting</td>
<td>Speed</td>
<td>IPC</td>
</tr>
<tr>
<td>4 mappers</td>
<td>0.32x</td>
<td>1.20</td>
</tr>
<tr>
<td>Baseline</td>
<td>1.00x</td>
<td>1.17</td>
</tr>
<tr>
<td>Opt. 5.2</td>
<td>1.05x</td>
<td>1.18</td>
</tr>
<tr>
<td>Opt. 5.3</td>
<td>1.07x</td>
<td>1.18</td>
</tr>
<tr>
<td>Opt. 5.4</td>
<td>1.14x</td>
<td>1.08</td>
</tr>
</tbody>
</table>
Example #2: Computation in Hashing

- Integer Divide is inefficient
  - Implemented as 9-uop flow
  - Contention with FP divides (transcendentals)
  - Contention w/ sibling thread
- In BDA
  - TopDown tags 2nd hotspot as Backend → Core Bound → Divider
    - Divider Busy 17%, Frontend Bound 20%, UPI 1.03
- Fix → speedup
  - SW workaround to avoid IDIV should there be no collision
  - 2% app-level additional speedup

```
int indexOfKey(T key) {
    int length = table.length;
    int hash = key.keyCode() & 0x7FFFFFFF;
    int i = hash % length;
    int decrement = hash % (length - 2);
    if (decrement == 0)
        decrement = 1;
    while (((state[i] != FREE) & ...}) {
        i -= decrement;
        if (i < 0) i += length;
    }
    if (state[i] == FREE) return -1;
    return i;
}
```

<table>
<thead>
<tr>
<th>Optimizations/setting</th>
<th>System Speedup</th>
<th>IPC</th>
<th>Instructions reduction</th>
<th>Memory Mem BW</th>
<th>Offcore Bound</th>
<th>UPI</th>
<th>Divider Active</th>
<th>Frontend Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>1.00x</td>
<td>1.17</td>
<td>n/a</td>
<td>2.91</td>
<td>27%</td>
<td>1.03</td>
<td>17%</td>
<td>17%</td>
</tr>
<tr>
<td>Opt. 5.2</td>
<td>1.05x</td>
<td>1.18</td>
<td>0%</td>
<td>3.16</td>
<td>26%</td>
<td>1.03</td>
<td>17%</td>
<td>20%</td>
</tr>
<tr>
<td>Opt. 5.3</td>
<td>1.07x</td>
<td>1.18</td>
<td>-3%</td>
<td>3.22</td>
<td>27%</td>
<td>1.00</td>
<td>13%</td>
<td>16%</td>
</tr>
<tr>
<td>Opt. 5.4</td>
<td>1.14x</td>
<td>1.08</td>
<td>-21%</td>
<td>3.25</td>
<td>29%</td>
<td>1.01</td>
<td>14%</td>
<td>14%</td>
</tr>
</tbody>
</table>
SMT

• SMT technology
  - 2x threads with improved utilization (or contended) of core resources

• 33% speedup vs 8C
  - 14% power reduction vs 16C
  - Core IPC improved by ~40%

• Implications
  - Optimizations potential (optimizes vs baseline) is higher by 5% thanks to SMT
  - Most datacenters keep SMT enabled
# Findings Summary

<table>
<thead>
<tr>
<th>Level</th>
<th>Parameter</th>
<th>Observation and/or Optimizations’ potential</th>
</tr>
</thead>
</table>
| System| JVM selection                 | Hotspot/OpenJDK = 1.43x  
|       |                               | IBM-J9/OpenJDK = 1.38x                                                          |
|       | SMT                            | MT vs CMP: 35% speedup, poor power reduction                                      |
|       | Turbo                          | Benefits reduce- and straggler map-jobs                                          |
|       |                                |                                                                                  |
| Application and Language | Algorithm                      | Wide inefficiencies. Demoed 50% speedup with 2x reduction in ext. memory demand  |
|       | Programming style              | Too abstracted code limits exploiting upcoming JVM and CPU parallelization features|
|       | Polymorphic Objects            | 25% uop reduction, 6% sample speedup                                             |
| μ/Architecture and Runtime | JVM code generation            | Overuses memory dereferences. 6% sample speedup                                  |
|       | CPU inefficiencies             | Fetch bandwidth and contended (SMT) EUs. e.g. Integer Divides. 2% sample speedup  |
|       | Control flow predication       | Data-dependent branch mispredictions. ~16% uops waste power on miss-speculated paths |
Agenda

- Introduction
- The workload
- Threefold Analysis
- Findings
  - Vs other workloads
  - Conclusions
Comparison to Traditional Workloads

- Decent IPC. Low Memory BW demand.
- Metrics: Off-core Bound [TopDown] vs Miss-Ratio [traditional], e.g. 462.libquantum
- Modest Off-core Bound hints on non-memory bottlenecks exist.
Microarchitectural Comparison

- Most left bar: black is % of cycles with retiring uops; white no retirement

Ahmad Yasin -- Deep-dive Analysis of the Data Analytics Workload in CloudSuite (IISWC 2014)
Summary

• Data Analytics (scale-out) has new characteristics
  - Deep software layers, heavy abstractions, Wide inefficiencies
  - Plenty of software optimizations opportunities

• Presented a customized 3-fold analysis method for System, Application and Architectural Levels

• Revealed BDA performance is limited on managing rather than accessing the data
  - Root-caused inefficiencies at the three levels
  - Most time is spent in few hotspots, unlike traditional Enterprise
  - Got 65% speedup through sample fixes

Try out this method on your favorite workload
Thank You

Questions?

(or to get best practices doc to perform analysis on your favorite workload)

Ahmad.Yasin@intel.com
Acknowledgements

• Koby Gottlieb
• Stas Bratanov
• Sandhya Viswanathan
• Tal Katz