IEEE International Symposium on
Workload Characterization (IISWC 2014 )
October 26-28, Raleigh, NC

# 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
  - https://sites.google.com/site/analysismethods/isca2013/program-1
- 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
    - –

[1] M. Ferdman, et al. "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," ASPLOS 2012.

[2] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," ISPASS 2014

# Motivation

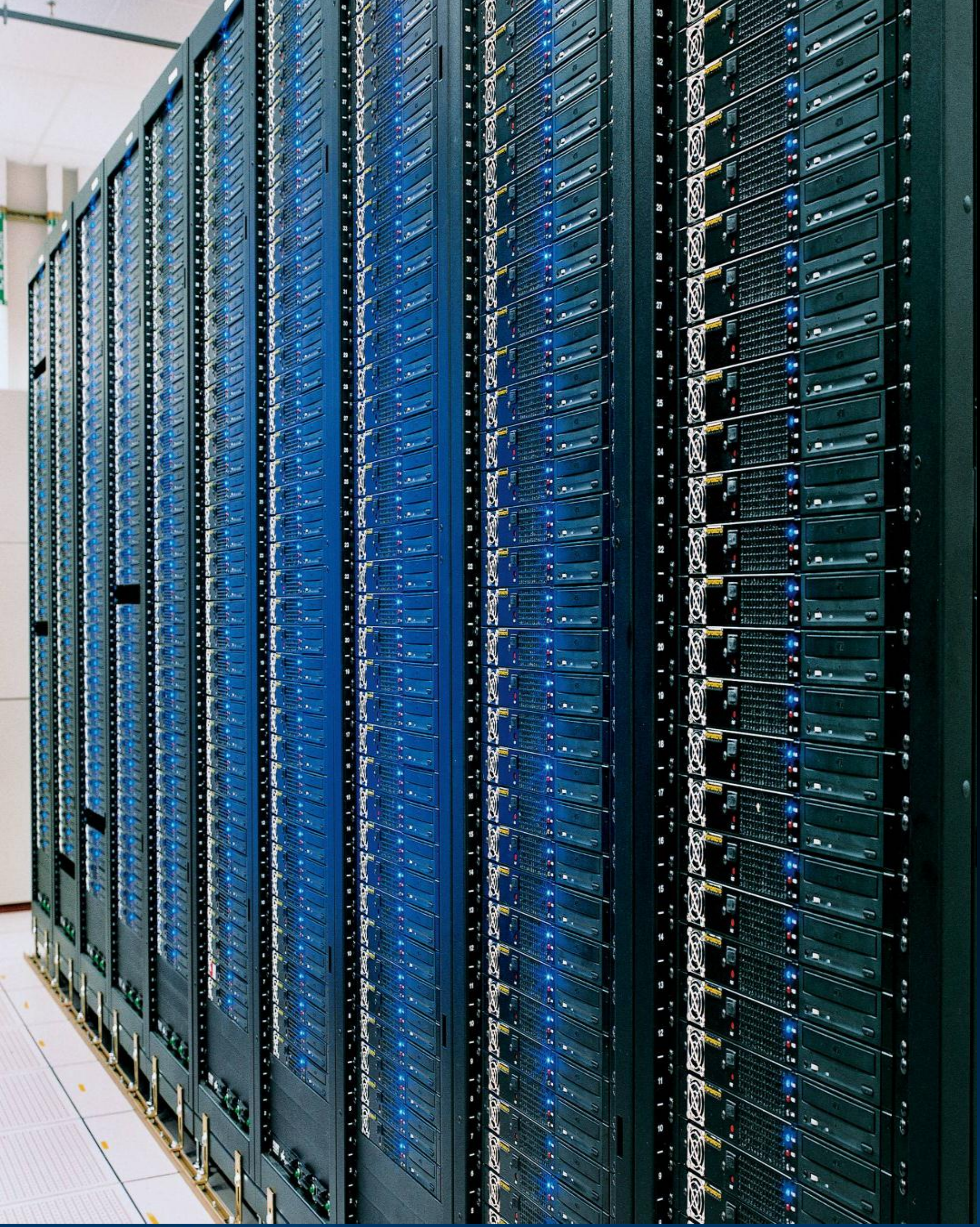
- Exponential data growth
- Massively-parallel hardware systems
- Orchestration software layers
  - Hadoop, Spark
- New scale-out applications
  - Store and process big data
  - Different Characteristics
- No understanding of the root causes
- Data Analytics (key for big data → value)

**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*

[4] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence: an analysis of Hadoop usage in scientific workloads,", VLDB 2013.



# Agenda

- ✓ Introduction
- The workload
- Threefold Analysis
- Findings
- Vs other workloads
- Conclusions

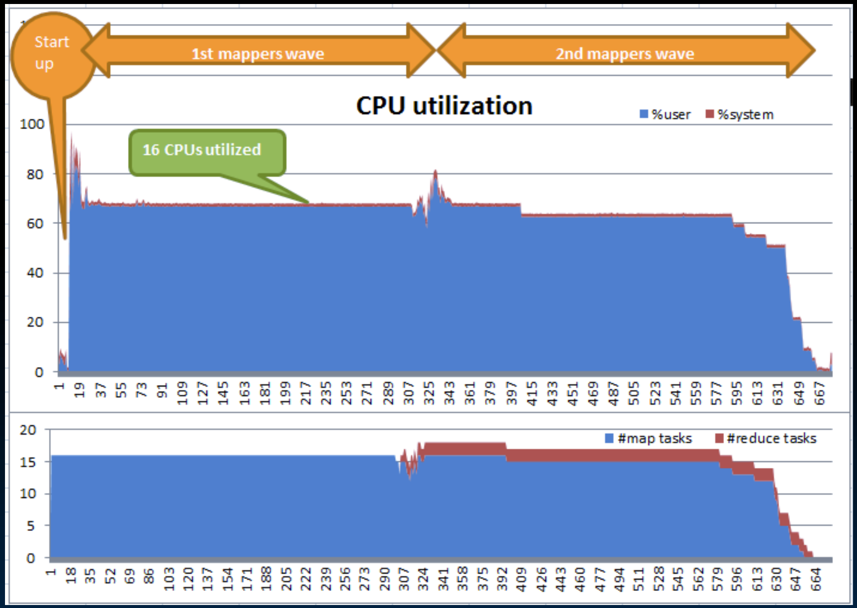
# Experimental Setup

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

| | | | |
|---|---|---|---|
| Hardware | CPU | uarch | Intel Xeon E5-2697 v2, Ivy Bridge μarch, 30MB LLC |
| | | Frequency | 2.7 GHz (Turbo→3.5) |
| | | # sockets/ cores/ threads | 2 / 12 / 1 or 2 (threads) |
| | Memory | | 1600 MHz. Max BW 60 GB/s |
| Software | OS | | Centos 6.5, Kernel 2.6.32 |
| | JVM | Oracle | HotSpot JDK 6u29 |
| | | OpenJDK | IcedTea6 1.13.0pre |
| | | IBM | J9 2.4 |
| | Hadoop | | Version 0.20.2 2GB Java heap per job |
| | Mahout | | Version 0.6, Naive Bayes algo. |

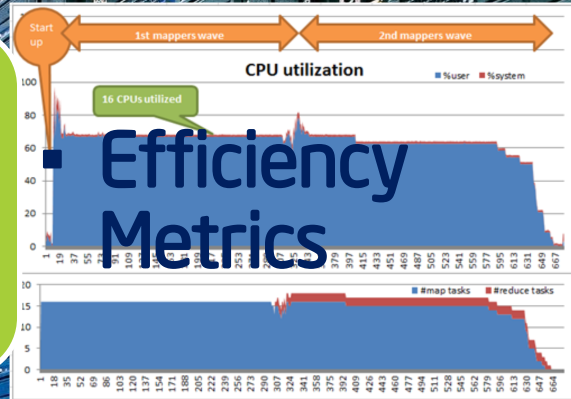
# The Data Analytics Workload

- 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



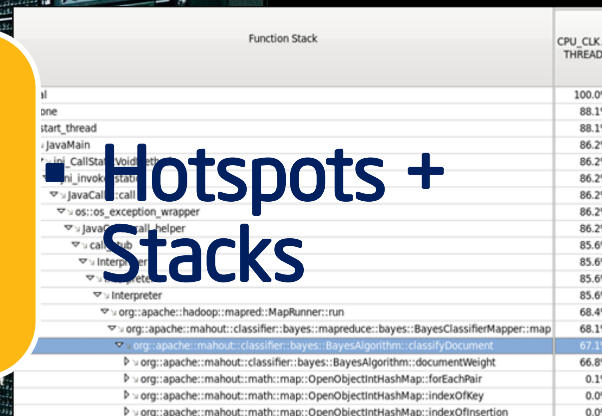
# Customized threefold analysis

System



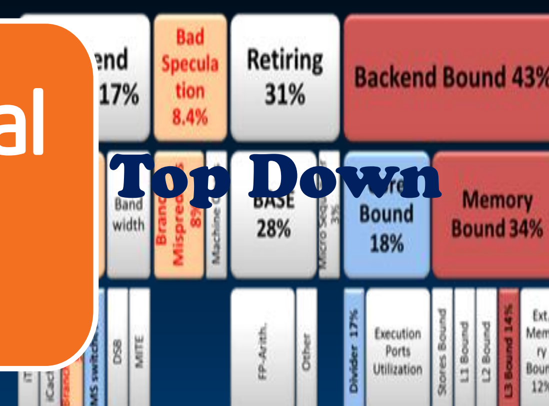

- Efficiency Metrics

Application + Runtime

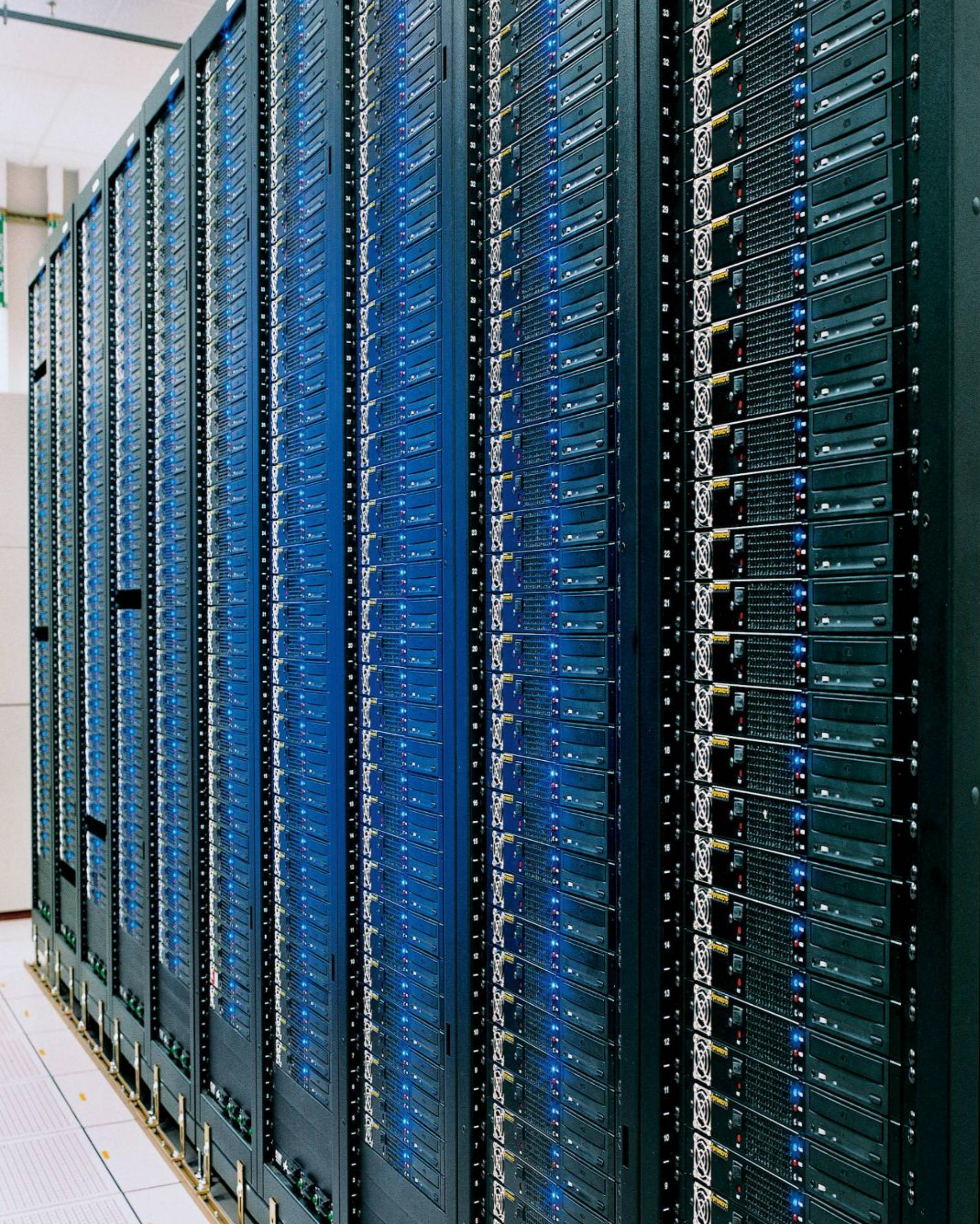



- Hotspots + Stacks

Architectural + µArch.




Top Down



# Agenda

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

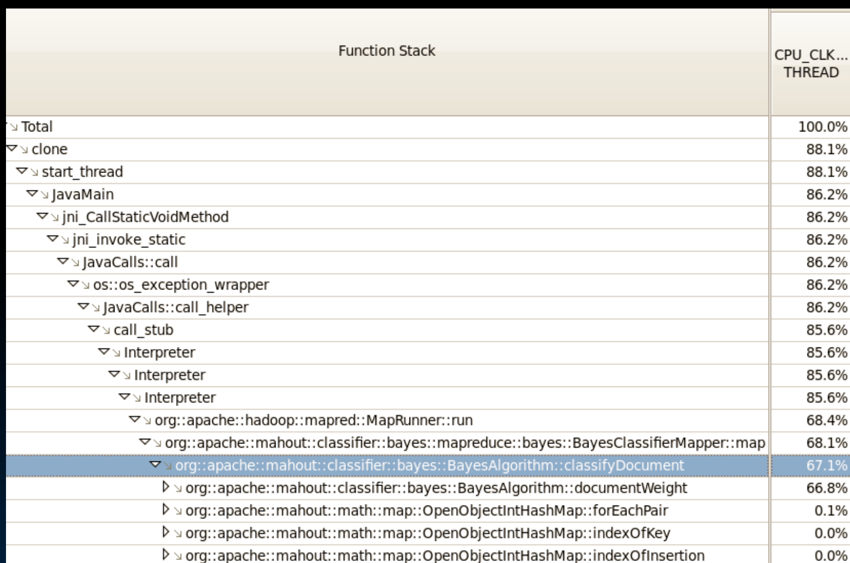
# System

- 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

| | | System | | | Memory | | Core | |
|---|---|---|---|---|---|---|---|---|
| JVM type | Speedup | CPU Utilization | Effective CPUs | IPC | Off-core Boun | Miss ratio | UPI | MS Switches |
| HotSpot (Baseline) | 1.43x | 77% | 12.4 | 1.17 | 27% | 12% | 1.03 | 5% |
| IBM J9 | 1.38x | 91% | 14.6 | 1.28 | 24% | 9% | 1.02 | 6% |
| OpenJDK | 1.00x | 128% | 20.5 | 0.77 | 12% | 9% | 1.40 | 25% |

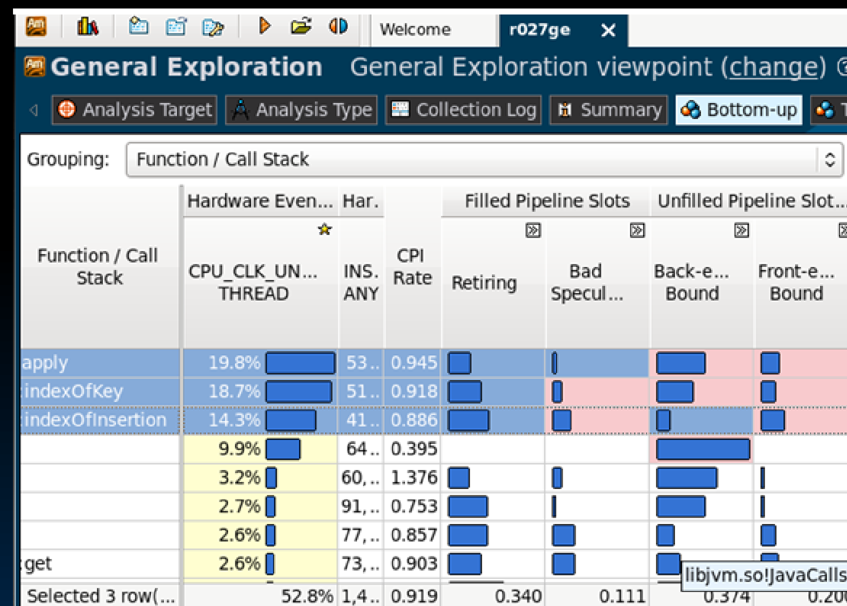
# Call-stacks

## Application Level



| Function Stack | CPU_CLK... THREAD |
|---|---|
| Total | 100.0% |
| clone | 88.1% |
| start_thread | 88.1% |
| JavaMain | 86.2% |
| jni_CallStaticVoidMethod | 86.2% |
| jni_invoke_static | 86.2% |
| JavaCalls::call | 86.2% |
| os::os_exception_wrapper | 86.2% |
| JavaCalls::call_helper | 86.2% |
| call_stub | 85.6% |
| Interpreter | 85.6% |
| Interpreter | 85.6% |
| Interpreter | 85.6% |
| org::apache::hadoop::mapred::MapRunner::run | 68.4% |
| org::apache::mahout::classifier::bayes::mapreduce::bayes::BayesClassifierMapper::map | 68.1% |
| org::apache::mahout::classifier::bayes::BayesAlgorithm::classifyDocument | 67.1% |
| org::apache::mahout::classifier::bayes::BayesAlgorithm::documentWeight | 66.8% |
| org::apache::mahout::math::map::OpenObjectIntHashMap::forEachPair | 0.1% |
| org::apache::mahout::math::map::OpenObjectIntHashMap::indexOfKey | 0.0% |
| org::apache::mahout::math::map::OpenObjectIntHashMap::indexOfInsertion | 0.0% |

## u/Architectural Level




**Examine where application's most time is spent**

# 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"

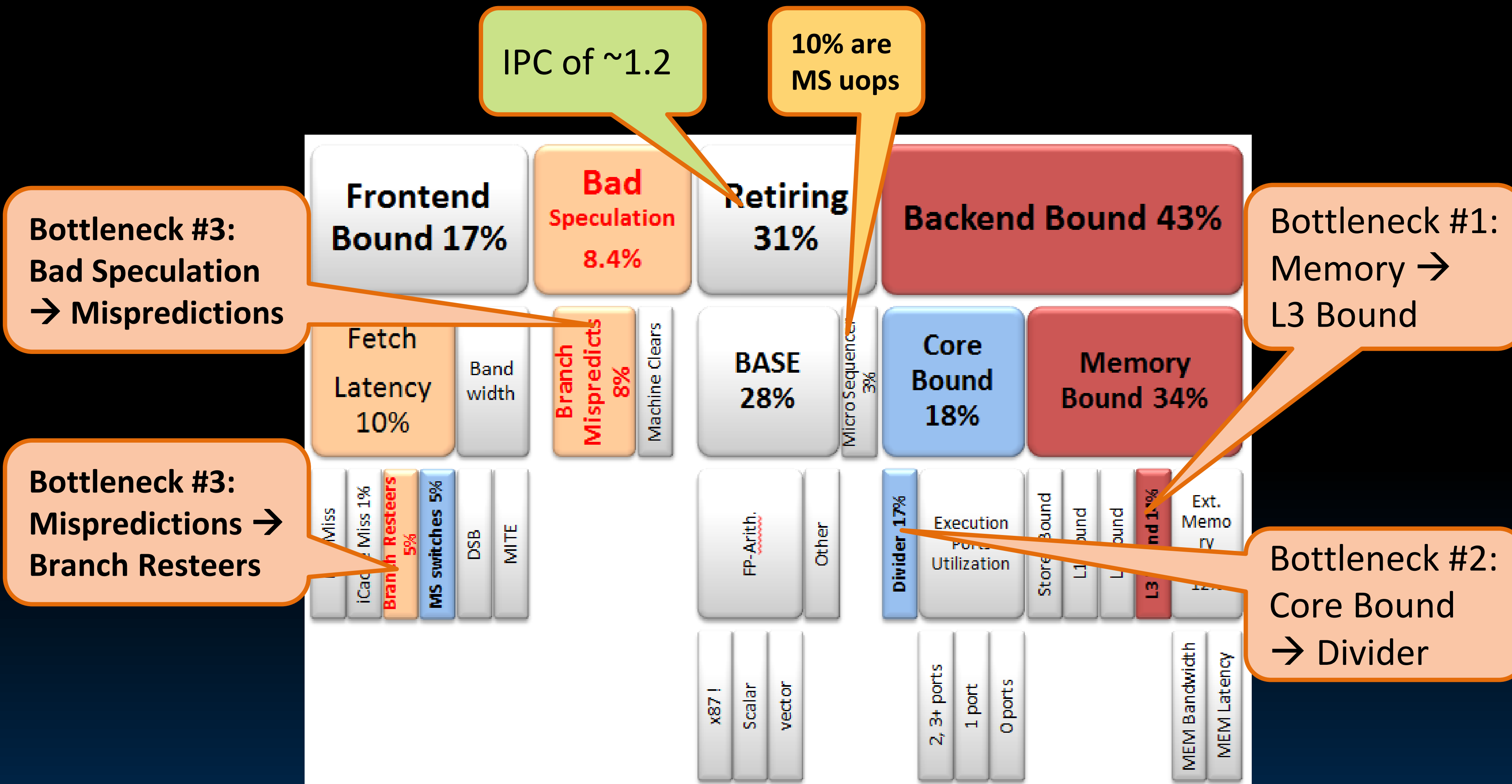
*Listing 1: Main classification loop pseudo-code*

```
Label classifyDocument(document) {
  label = default_label;
  foreach (category : categories) {
    hash = new HashMap<String><int>;
    foreach (word : document)
      hash.update(word, 1);

    result = 0;
    foreach (pair-of [word, frequency] : hash)
      result += frequency *
        featureWeight(category, word);

    if (result is a maximum)
      label = category;
  }
  return label;
}
 where featureWeight is:
double featureWeight(label, word) {
   return - log[(getW("weight", label, word) +
 getW("params","αi")) /(getW("labelWeight",label) +
 getW("sumWeight","vocabCount"))];
}
```

# Top Down Performance Analysis




[2] A. Yasin, “A Top-Down Method for Performance Analysis and Counters Architecture”, ISPASS 2014

# Per hotspot drill down

**Level#1**

| Function / Call Stack | % time | IPC | Retiring | BadSpeculation | Backend Bound | Frontend Bound | Module |
|---|---|---|---|---|---|---|---|
| BayesAlgorithm$1::apply | 24.8% | 1.09 | 18% | 5.1% | 60% | 17% | [Compiled Java code] |
| HashMap::indexOfKey | 23.0% | 1.15 | 34% | 5.2% | 46% | 17% | [Compiled Java code] |
| shMap::indexOfInsertion | 16.2% | 1.27 | 42% | 15.5% | 21% | 26% | [Compiled Java code] |
| HashMap::get | 3.1% | 1.11 | 30% | 15.4% | 30% | 27% | [Compiled Java code] |
| ..Datastore::getWeight | 3.1% | 1.18 | 41% | 0.8% | 46% | 14% | [Compiled Java code] |

**Level#2**

| BASE | Microcode Sequencer | BadSpeculation | Memory Bound | Core Bound | Frontend Latency | Frontend Bandwidth |
|---|---|---|---|---|---|---|
| 18% | 4% | 5% | 51% | 34% | 13% | 4% |
| 35% | 3% | 5% | 32% | 44% | 7% | 10% |
| 49% | 4% | 16% | 14% | 43% | 12% | 14% |

**Level#3.1**

| L1 Bound | L2 Bound | L3 Bound | MEM Bound | Divider Active | Cycles of 1 Port Utiliz. | Cycles of 2 Ports Utiliz. | Cycles of 3+ Ports Utiliz. |
|---|---|---|---|---|---|---|---|
| 11% | 16% | 13% | 11% | 23% | 17% | 15% | 16% |
| 10% | 20% | 1% | 0% | 20% | 23% | 21% | 21% |
| 8% | 5% | 0% | 0% | 18% | 19% | 25% | 37% |

**Level#3.2**

| FP X87 | FP Scalar | FP Vector | Branch Mispredicts | Machine Clears | ICache Misses | Branch Resteers | DSB Switches | Bandwidth DSB | Bandwidth MITE |
|---|---|---|---|---|---|---|---|---|---|
| 7% | 1% | 0% | 5% | 0% | 0% | 2% | 1% | 16% | 8% |
| 5% | 0% | 0% | 5% | 0% | 0% | 3% | 1% | 36% | 3% |
| 3% | 0% | 0% | 15% | 0% | 0% | 4% | 0% | 41% | 0% |

# 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`*

```
int indexOfKey(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;
}
```

| | System | | | Memory | | Core | | |
|---|---|---|---|---|---|---|---|---|
| Optimization/*setting* | Speed up | IPC | Insts. reduction | Mem BW | Offcore Bound | UPI | Divider Active | Frontend Bound |
| *4 mappers* | *0.32x* | *1.20* | *n/a* | *1.17* | *26%* | *1.03* | *16%* | *17%* |
| Baseline | 1.00x | 1.17 | n/a | 2.91 | 27% | 1.03 | 17% | 17% |
| Opt. 5.2 | 1.05x | 1.18 | 0% | 3.16 | 26% | 1.03 | 17% | 20% |
| Opt. 5.3 | 1.07x | 1.18 | -3% | 3.22 | 27% | 1.00 | 13% | 16% |
| Opt. 5.4 | 1.14x | 1.08 | -21% | 3.25 | 29% | 1.01 | 14% | 14% |

# Example#2: Computation in Hashing

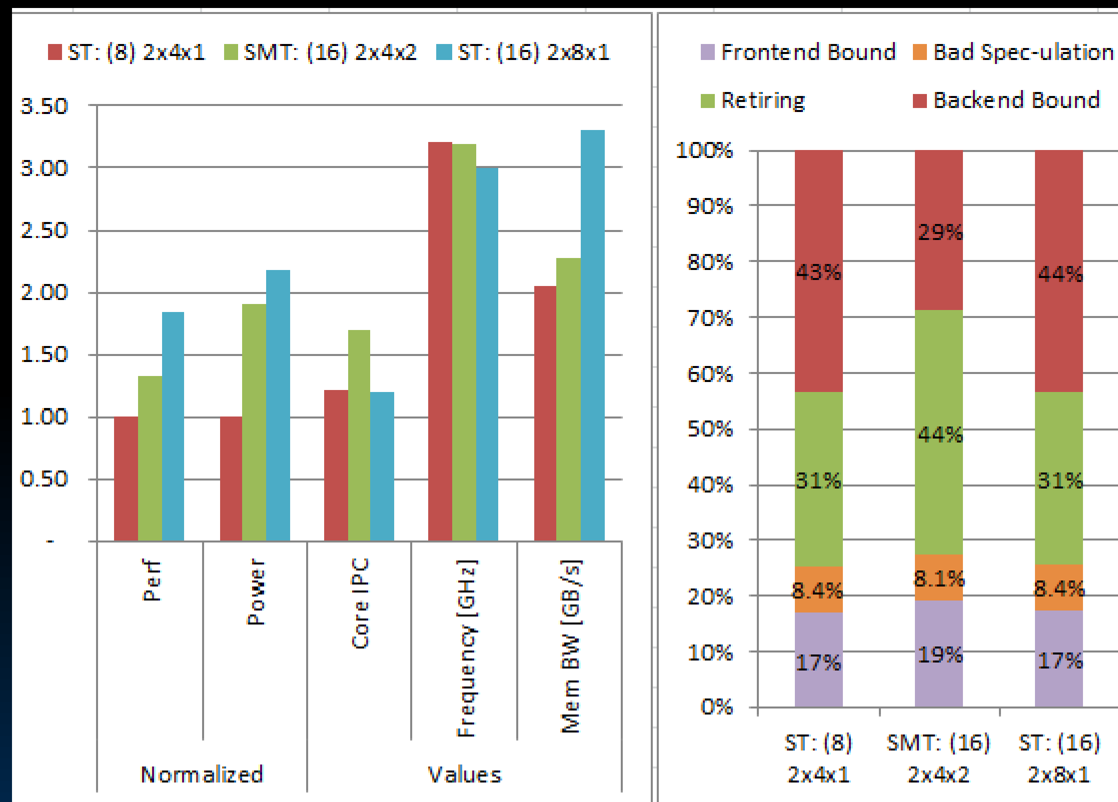
- Integer Divide is inefficient
  - Implemented as 9-uop flow
  - Contention with FP divides (transcendentals)
  - Contention w/ sibling thread
- In BDA
  - TopDown tags 2nd hotspot as Backend→Core Bound→Divider
  - Divider Busy 17%, Frontend Bound 20%, UPI 1.03
- Fix → speedup
  - SW workaround to avoid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

```
int indexOfKey(T key) {
  int length = table.length;
  int hash = key.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;
}
```

| | System | | | Memory | | Core | | |
|---|---|---|---|---|---|---|---|---|
| Optimization/*setting* | Speed up | IPC | Insts. reduction | Mem BW | Offcore Bound | UPI | Divider Active | Frontend Bound |
| *4 mappers* | *0.32x* | *1.20* | *n/a* | *1.17* | *26%* | *1.03* | *16%* | *17%* |
| Baseline | 1.00x | 1.17 | n/a | 2.91 | 27% | 1.03 | 17% | 17% |
| Opt. 5.2 | 1.05x | 1.18 | 0% | 3.16 | 26% | 1.03 | 17% | 20% |
| Opt. 5.3 | 1.07x | 1.18 | -3% | 3.22 | 27% | 1.00 | 13% | 16% |
| Opt. 5.4 | 1.14x | 1.08 | -21% | 3.25 | 29% | 1.01 | 14% | 14% |

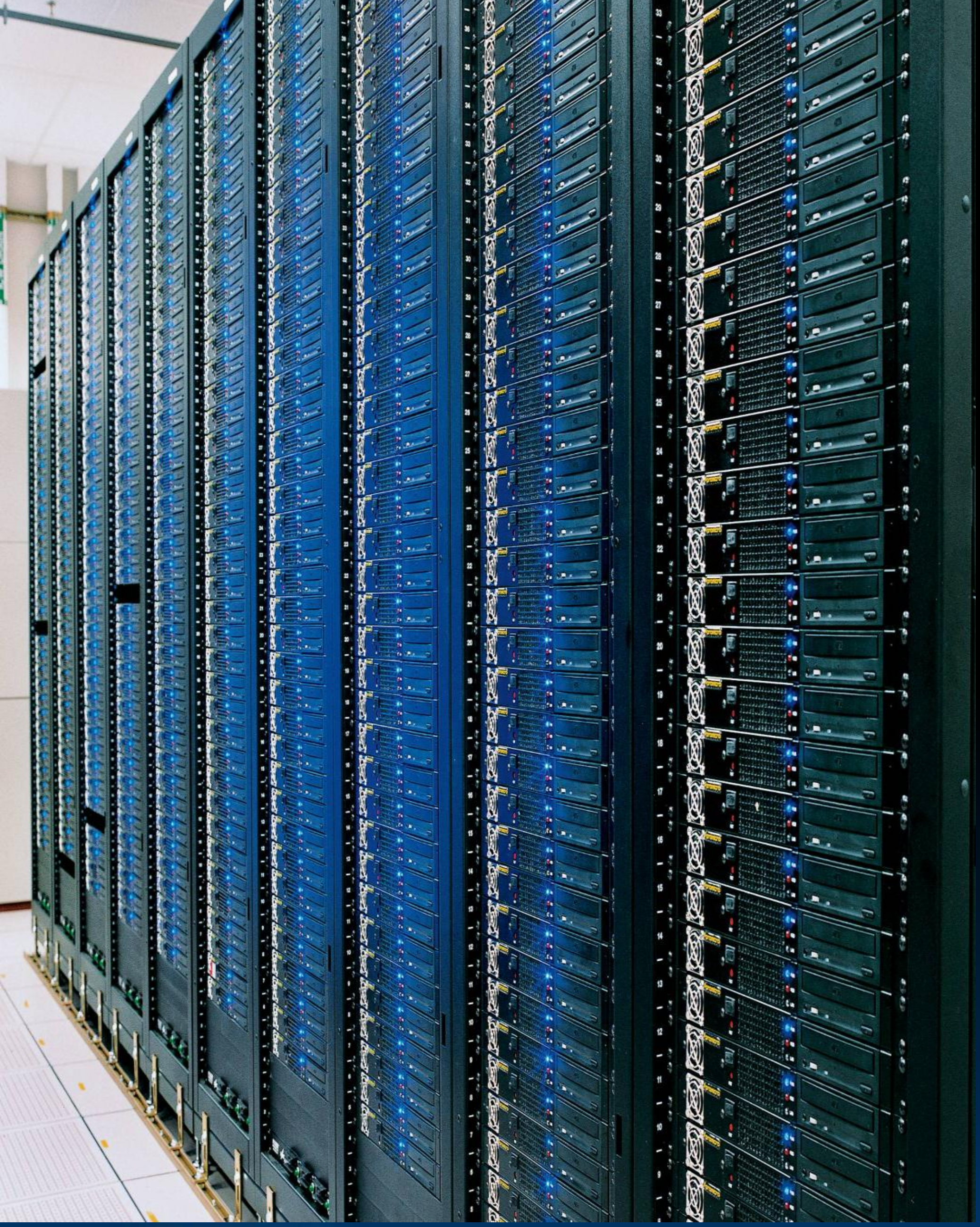
# SMT

- 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

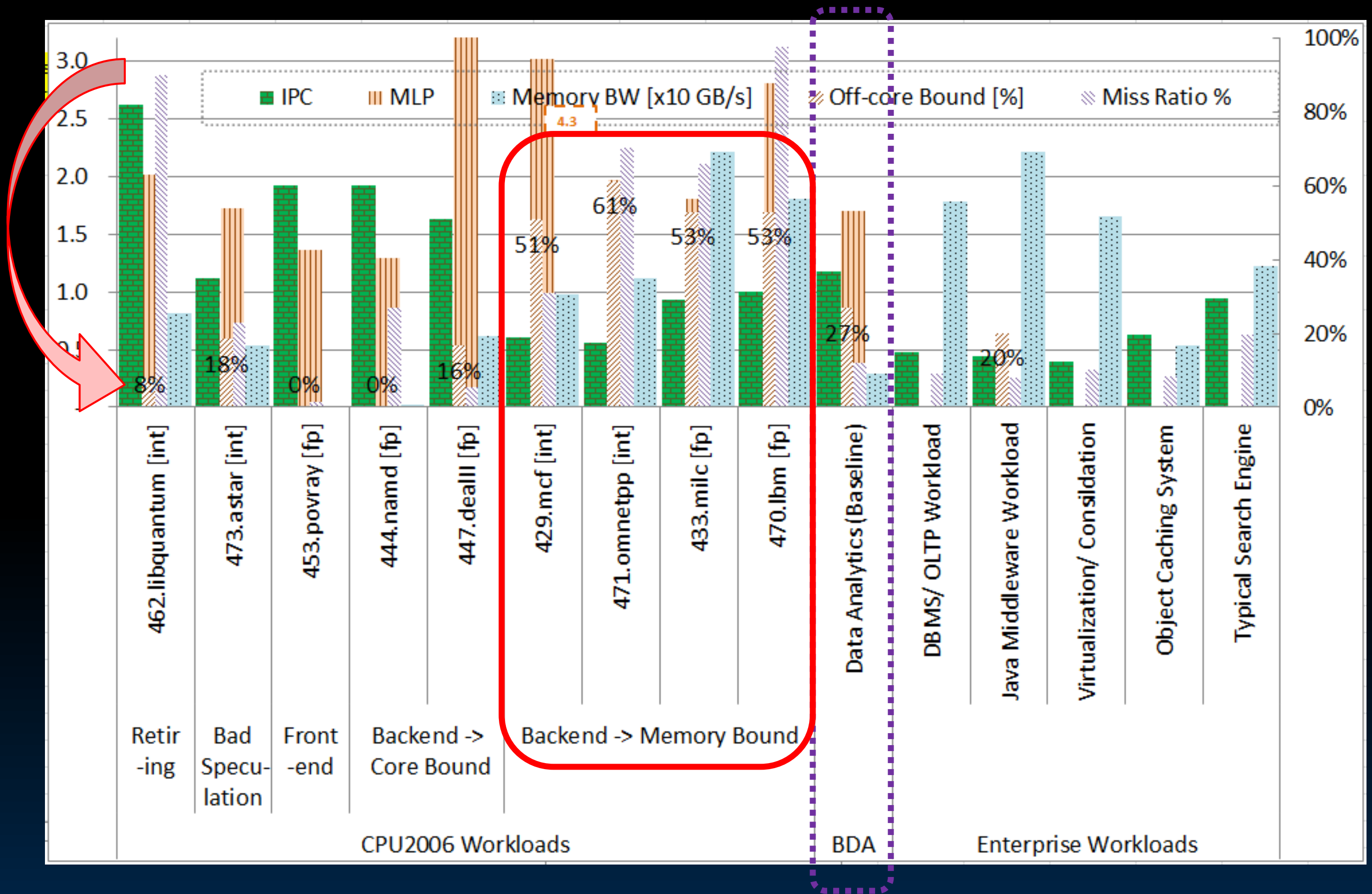
| Level | Parameter | Observation and/or Optimizations' potential |
|---|---|---|
| System | JVM selection | Hotspot/OpenJDK = 1.43x<br>IBM-J9/OpenJDK = 1.38x |
| | SMT | MT vs CMP: 35% speedup, poor power reduction |
| | Turbo | Benefits reduce- and straggler map-jobs |
| Application and Language | 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

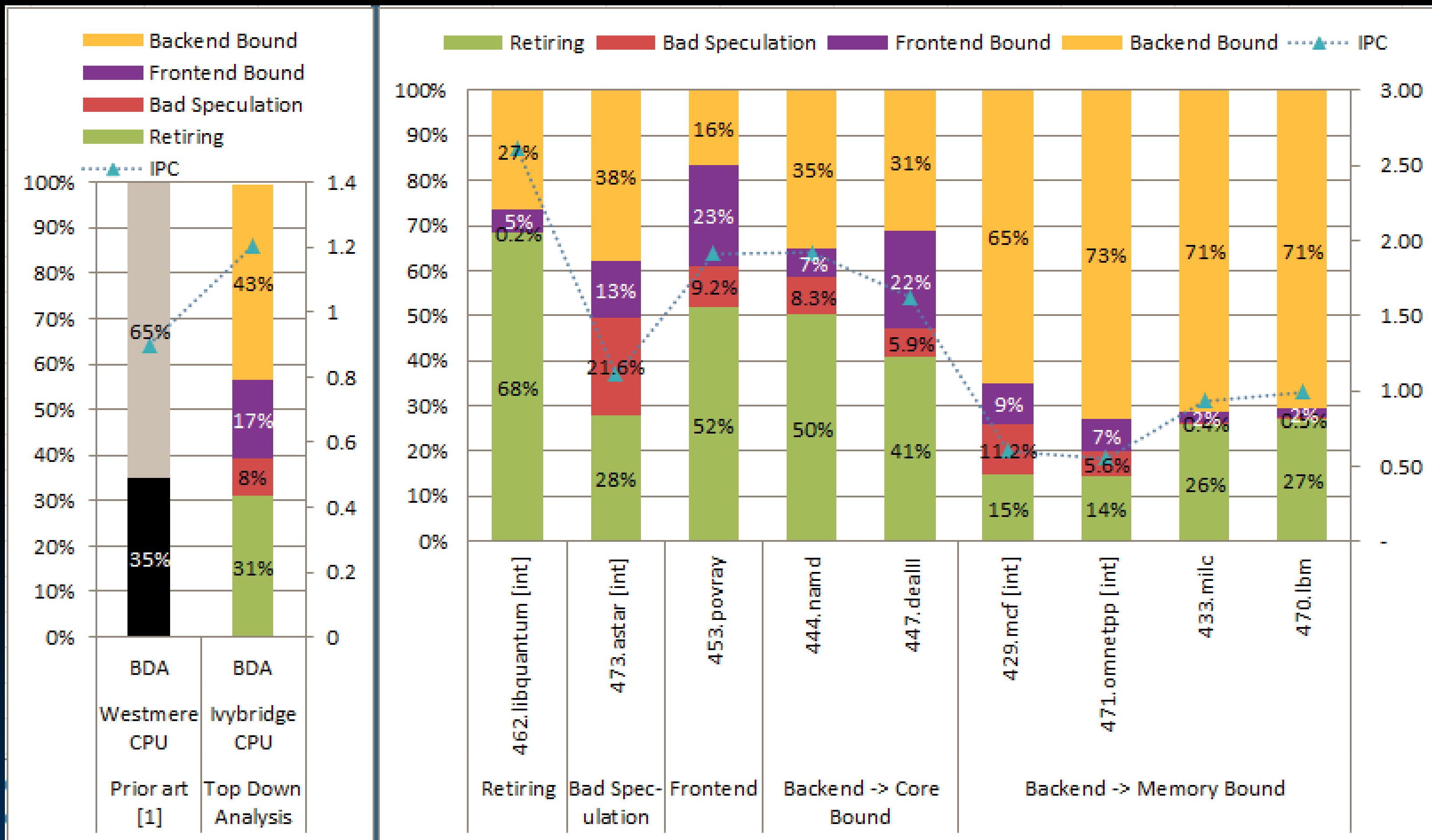
- ✓ Introduction
- ✓ The workload
- ✓ Threefold Analysis
- ✓ Findings
- Vs other workloads
- Conclusions

# 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

# 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