Deep-dive Analysis of the Data Analytics Workload in CloudSuite

Ahmad Yasin[1,2] Yosi Ben-Asher[2] Avi Mendelson[3]
[1] Intel Corporation [2] University of Haifa [3] Technion

Abstract

Exponential growth of digital data has introduced massively-parallel systems, special orchestration layers, and new scale-out applications. While recent works suggest characteristics of scale-out workloads are different from those of traditional ones, their root causes are not understood. Such understanding is extremely important to improve efficiency; even a 1% performance gain for a core can have a large impact on the datacenter as a whole.

This paper studies the characteristics of a Big Data Analytics (BDA) workload on a modern cloud server. It is intentionally focused on a single workload-platform in order to enable deep-dive analysis that aims to understand the root causes of the CPU bottlenecks which this paper identify. We choose the Data Analytics benchmark from CloudSuite[1] as a representative of a growing family of important applications.

This paper describes a customization of a comprehensive threefold analysis method. The method consists of a System level, where sensitivity to system parameters is examined, as well as Application and Architectural levels; where bottlenecks are attributed back to the application and runtime codes, respectively. The paper also adopts a proof-by-optimization approach to prove bottlenecks' validity. Overall, 65% net speedup is measured with significant power reduction.

The paper reveals that BDA workloads suffer from overheads related to *managing* the data rather than *accessing* the data. For example, Hash index lookup is found to be a key performance limiter. Inefficiencies leading to such unexpected behavior are demonstrated, including JVM selection and heavily unoptimized application code, both of which have a big impact. Suboptimal microarchitecture areas are demonstrated too, in addition to programming styles that limit exploitation of upcoming JVM and CPU parallelization features.

1. Introduction

Background: The focus of computer systems has gradually moved from the end-point, such as a processor core, to full systems; and from general-purpose systems to domain-specific applications. A major factor contributing to this change is the exponential growth of digital data which lead to the prevalence of huge datacenters[6][7].

New software layers were developed for these massively-parallel systems such as MapReduce[8], – the leading framework for large-scale data processing, and its popular open source implementation, Hadoop[9]. Such frameworks have become popular because they allow ordinary developers, not just those skilled in high-performance computing, to exploit large systems without worrying about the complex parallelism and warehouse-scale system details such as data-partitioning and failure handling[4].

Problem: Understanding the characteristics of scale-out workloads is extremely important because any improvement of some component, such as an end-point compute engine, impacts a large number of units and makes a sizable impact on the entire system. A performance improvement of 1% can result in millions of dollars saved at datacenter scale[5].

Recent work concludes that characteristics of scale-out workloads are completely different from those of traditional ones[1]. However, the root causes leading to such characteristics are not understood. Unfortunately, field analysis of scale-out workloads is a complicated task. Some researchers [1][11] use VTune[3] for profiling. Using such tools requires a deep understanding of both the architecture and the software environment in order to guarantee the quality of the results. New challenges appear due to the nature of scale-out workloads. For example, the workloads are sensitive to being run on appropriately sized platforms. Analysis challenges specific to BDA are enumerated in section 2.

Scope: This paper focuses on understanding the actual bottlenecks of Data Analytics workload on a modern cloud server. The workload is part of CloudSuite 2.0 – a CPU-centric benchmark suite for scale-out applications[1].

We choose the data analytics benchmark because it relies on two important packages for data analytics: Hadoop and Mahout[10], hence it represents a growing family of applications designed to handle vast amounts of data. BDA itself performs significant mathematical work when manipulating the data and so it balances between the I/O and compute demands. Similar to our choice, HiBench – another benchmark suite for Hadoop characterization – has included Mahout's Bayesian Classification as a representative real-world application with heavy computation[12]. Lastly, [4] reported Mahout has clear advantage in terms of load-balancing jobs in comparison to other applications in Hadoop production deployments.

This paper intentionally focuses on a single-workload single-machine, rather than a set of applications, in order to enable deep-dive analysis for the purpose of understanding the root causes behind the bottlenecks. The workload itself is CPU bound as it uses in-memory database and we reveal enough issues in a single node setup (a typical Hadoop setup has thousands of nodes). The paper not only identifies the

978-1-4799-6454-3/14/$31.00 ©2014 IEEE

bottlenecks, but also adopts a proof-by-optimization approach to verify the bottlenecks are indeed critical.

Contributions: Our key contributions include:

- We *customize a threefold analysis process* to suit scale-out workloads. The threefold process itself, consisting of system, application and architecture levels, is not new. We illustrate the challenges a naive user may face and consequently customize the legacy process in specific ways to suit our framework. (See Section 2).
  We *demonstrate how to use the customized method* on BDA and successfully report multiple findings (summarized in *Table 4*). The main findings are highlighted below.
- At the System level, we identify that *JVM efficiency has a major impact*. For example, OpenJDK has basic macro-level inefficiencies leading to significant slowdown as compared with other JVMs. This is important as widespread packages are written in Java, and OpenJDK is the default in Linux distributions. (See section 3.)
  We contribute two generic metrics to call-out runtime performance anomalies at the macro and micro levels.
- At the Application level, we reveal that *BDA has inefficient application code*. For example, one optimization in the main loop gets 50% speedup. We also report programming styles that limit exploitation of new features in upcoming JVM and CPU releases. These insights lead us to believe there is much room for optimization in scale-out workloads in general, as [4] reported. (See section 4.)
- At the μ/Architectural level, we identify *hash index lookup as a key limiter*. We point to concrete deficiencies at the runtime generated code and microarchitecture implementation. (See section 5.)
- We *conduct a detailed comparison* of BDA characteristics vs. traditional workloads and analysis methods. The main conclusion is that BDA has *fairly distributed bottlenecks* which call for accurate bottleneck identification methods at the lowest level. (See section 6.)

Overall, we *measured* 65% speedup via ad-hoc optimizations. We notice one commonality among the identified issues – they all deal with maintaining and traversing the data (as opposed to, for example, the data working set not fitting into cache). Hence we summarize that BDA suffers from overheads related to *managing* the data rather than overheads related to *accessing* the data itself.

2. Analysis Methodology

We adopt a threefold analysis process. Because the process itself is commonly used[3][13], the focus of this section is to highlight relevant aspects and emphasize our contributions, showing how we customized the existing process.

2.1. Threefold Process Background

A typical analysis process has system, application and architecture levels. External memory setup, algorithm complexity and cache misses are sample issues at the three levels, respectively. An analysis should follow the specified order, because each level has a bigger potential than its successor. For example, reducing an algorithm's complexity from $O(n^2)$ to $O(n\log n)$ (application level) would incur more benefits than vectorizing it (architecture level).

Our workload has multiple software layers: the operating-system, JVM, Hadoop system, Mahout Library (algorithms and data structures) and the user application. An average developer is unlikely to be familiar with all these entities; hence it is important to conduct a solid and systematic analysis. We adopt a customized analysis process to identify performance inefficiencies, and then use a proof-by-optimization approach to verify the identified issues are indeed critical.

2.2. System Analysis

The System level characterizes issues related to hardware and software system-components. The latter include JVM and Hadoop in our case, each with plenty of configuration parameters. On the other hand, modern processors include more and more configurable features such as SMT and Turbo. Datacenter vendors configure such features to best fit workloads' demands. [1] has concluded that scale-out workloads have completely different characteristics than traditional workloads. This further increases the importance of insightful system level analysis.

While Hadoop enables ordinary users to exploit multicores throughout inherent parallelization, its abstraction complicates resource utilization judgment. For example, CPU core-count, Hadoop configuration and dataset partitioning all impact CPU utilization. To address this we facilitate a new metric called *Effective CPU*s which can be checked against an expected CPU Utilization. This metric was key to revealing macro level inefficiency in one of the experimental configurations, as discussed in section 3.3. *UPI* is another metric that can point to generated code inefficiency at the micro level.

Note we had to carefully judge whether legacy BKMs suit our workload. For example, some tuning guides advise to disable Turbo[3]. We performed all our analysis with Turbo on as it heavily impacts MapReduce-style jobs (see section 3.4).

We adopted and defined the following traditional and new-custom characterization metrics, respectively:

Legacy Metrics

- CPU Utilization
- IPC – Instructions Per Cycle
- Mem BW – Memory Bandwidth [GB/s]
- MLP – Miss-Level Parallelism
- Miss Ratio (at last level cache)

New Metrics

- **Effective CPUs** – The average number of CPUs that would be required in order to complete the workload. E.g. with full CPU utilization we should expect this number to equal *M* – the number of mappers – because map tasks dominate execution time (see section 3.2).
- **UPI** – Uops Per Instruction – The average number of micro operations (uops) to which an ISA instruction is translated. This metric serves as an indicator for how efficient

the code-generation and instruction-selection were for the given microarchitecture.

- **Off-core Bound** – A TopDown oriented metric which measures the fraction of time the core execution is stalled while waiting for off-core data (i.e., aggregation of L3 Bound and Ext. Memory Bound[2]).

2.3. Application Analysis

The Application level analyzes application code when performance data is mapped back to source code. Hadoop's Adolescence[4] observed wide inefficiencies in scientific applications at three different Hadoop clusters. The deep software layers are a key contributor to this observation. Hence this level double-checks the bulk of time spent at expected areas – a check that is not trivial anymore with nested 3rd party packages.

Many profilers offer event-based sampling (EBS) to associate performance counters back with the originating code[3][14]. While EBS offers low overhead compared to instrumentation-based methods, EBS alone is insufficient. We found that collecting **call-stacks** with event samples is a critical factor. The complexity is increased not only by multi-layer software stacks, but also by heavy abstractions use. Our BDA's map job makes excessive use of *abstracted data-structures* to maintain various objects. It relies on the Mahout-Collections library which has a hash-map with generic key-types and value-types. For example, an `Open<String><int>HashMap` instance is used to maintain the frequency of words.

JIT: Because the application is CPU bound, the JVM should detect and compile the hot methods Just-In-Time. This challenges the profiler's EBS engine because the binary code is dynamically generated.

We have used VTune Amplifier XE 2013[3] – the rich profiler – with collect-stacks mode for Application level analysis. We want to bring to the reader's attention that JIT and call-stacks are crucial capabilities for BDA and Hadoop workloads. JIT is often missed in mainstream industry profilers such as the Linux perf utility and Gooda[14]. Even in VTune it is featured for certain deployments.

2.4. µ/Architectural Analysis

The Architectural and Microarchitectural (µ/Architectural) level further analyzes inefficiencies due to interactions among runtime, architecture and microarchitecture at the lowest level.

For accurate bottlenecks attribution, we used the Top Down Analysis method (TopDown), recently published[2] and released with VTune. TopDown adopts a hierarchical approach to identify *critical* bottlenecks in out-of-order CPUs. The hierarchy depicted in *Figure 1* systematically addresses stalls overlap, speculative execution, variable miss penalties and superscalar inaccuracies; to name key challenges that preceding methods had struggled to deal with. In addition, the hierarchy design does not restrict the model to a predefined set of common miss-events. Identifying the right bottleneck is especially important due to the *fairly distributed bottlenecks* nature of BDA.

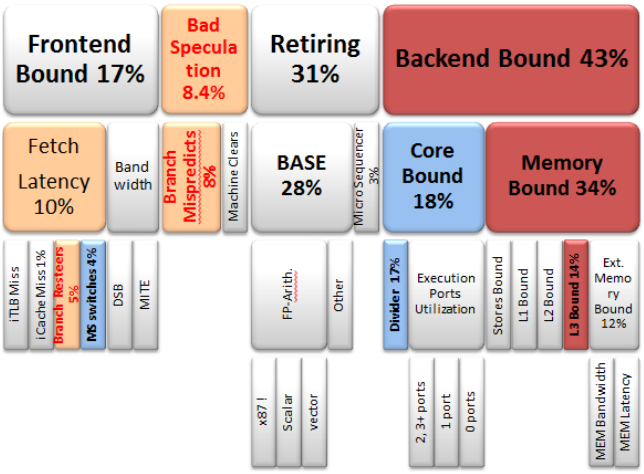

Figure 1: Top-Down Analysis hierarchy for BDA

An example is the best way to illustrate how critical the analysis method is. In our workload, 17% of the pipeline issue-slots are classified as Frontend Bound[2], meaning that the frontend portion of the CPU undersupplies its backend in 17% of the cases. We find the workload is limited by uncommon limitations leading to an undersupply of the fetch bandwidth as caused by IDIV (the Integer Divide of x86 instructions set), for example. Traditional analysis methods, such as the one used by [1], correctly concluded this workload has no *familiar* i-cache and i-TLB misses but has missed *uncommon* fetch issues that can limit performance of superscalar cores. Unrestricting the model to a predefined set of common miss-events is a key differentiation adopted by our analysis framework. Note we interpret the whole hierarchy of *Figure 1* later in section 5.1. Note a fraction of some node is comparable to its sibling nodes, and may not equal the sum of its children[2].

3. System Level

This section characterizes the workload "horizontally", examines its sensitivity to some parameters, and suggests a few guidelines for BDA vendors.

3.1. System Setup

We conduct our analysis and optimizations on a dual-socket Intel Xeon® E5-2697 v2 (codenamed Ivytown or Ivy Bridge EP). Each socket includes 12 out-of-order processor cores with a three-level cache hierarchy: the L1 and L2 caches are private to each core, while the LLC (L3) is shared among all cores in same socket. *Table 1* has system setup details.

Baseline configuration: Hadoop is configured with 16 mappers and 2 reducers executed by Oracle's HotSpot JVM. The CPU has both sockets enabled and Turbo enabled. Note we do not disable unused cores as we noticed certain workload phases make use of more than *M* CPUs as suggested by *Figure 2*, *M* being the number of mappers. All performance and counter data is reported as an average of three runs in

order to deal with run-to-run variation[1]. Hadoop setup is appropriately sized for this workload[1].

Table 1: System Setup Details

Hardware	CPU	Name	Intel Xeon E5-2697 v2, Ivy Bridge µarch, 30MB LLC
		Frequency	2.7 GHz (Turbo up to 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.

Consider the Baseline row in *Table 2*; the data tells us the workload has unsaturated CPU utilization, a modest IPC of 1.17, high LLC miss ratio and low memory bandwidth. This largely corroborates with [1], who also reported high miss ratio, low memory bandwidth, and an IPC of 0.9 on Westmere (a two-generation older platform). It is noteworthy that the same BDA workload has sped up by 33% Westmere to Ivytown. This means Xeon processors have sizable benefits, unlike what [1] and [15] concluded.

3.2. Workload Description

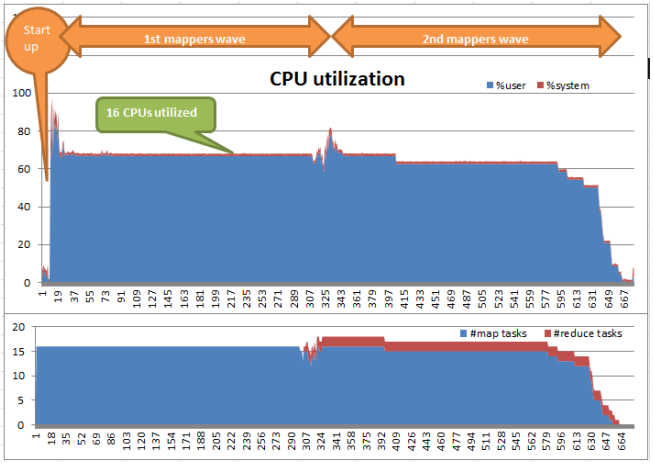

Figure 2: Overtime Hadoop CPU Utilization

The BDA program performs classification using the Mahout machine-learning library[10], where it guesses a category for each page in a 4.5 GB dataset of Wikipedia pages. Hadoop "orchestrates" the work, with the data divided into 32 partitions distributed among map tasks. The map-function, within Mahout, models the Naive Bayesian classification algorithm, where it emits for each page a *predicted-label*; whereas the reduce-function is a simple summing operation which counts tuples of <correct-label, *predicted-label*>. Both Hadoop and Mahout are in Java.

Figure 2 illustrates the overtime execution of the whole workload for the baseline configuration with 16 mappers. The 32 data-partitions are processed in two "waves" accounting for the major bulk of execution time. The reduce tasks account for minimal execution time. All of the following have negligible impact: operating-system (as *Figure 2* shows), disk I/O and Hadoop system (not shown).

3.3. JVM efficiency

Table 2 compares key metrics for the three JVMs we tested. Such a comparison is effective for spotting global inefficiencies or unexpected behavior.

HotSpot is 1.43 times faster than OpenJDK. This can be attributed to inefficiency at the macro level where OpenJDK used many more *Effective CPUs*: A net of 12.4 vs. 20.5 logical CPUs were utilized on average to complete the same task by HotSpot vs. OpenJDK, respectively. We observed OpenJDK java processes somehow consume over 100% CPU utilization which triggered deeper analysis. 63% of OpenJDK time was spent inside the JVM runtime library (libjvm.so) vs. 15% in the HotSpot case. OpenJDK's extra time was due to inefficiencies in scheduling of (parallel) garbage-collector threads – we observed much time was wasted in spin-wait loops using call-stacks.

Table 2: Characterization Metrics for Tested JVMs

		System			Memory		Core	
JVM type	Speed up	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%

A micro level check suggests OpenJDK generated suboptimal x86 instructions – an instruction generated by OpenJDK is decoded into 1.4 uops on average (i.e. UPI), while the same metric is closer to 1 for both Hotspot and J9. OpenJDK uses suboptimal CISC instructions with multiple uops. For example, the use of the PAUSE instruction in the aforementioned spin-wait loops. This is backed up by the *MS Switches* TopDown metric[3], where the penalty due to switches from fast fetch units to the Micro-Sequencer – with a cost of 3 cycles-per-instance – accounts for 25% of runtime in OpenJDK vs ~5% in HotSpot/J9.

To summarize, the JVM implementation heavily impacts performance. This observation is especially important as OpenJDK becomes a de-facto default in Linux distributions, while it is a common misconception that it performs at HotSpot's performance level.

[1] We used 16 cores/mappers settings, while [1] used 4 cores/mappers. *Table 3* shows 4- and 16-mappers have similar characteristics.

3.4. Turbo

Turbo dynamically varies the frequency per workload's instantaneous demands and available power budget[16]. When Turbo exceeds nominal frequency, it accelerates sequential application phases hence improve overall performance. We find this is a key attribute for boosting BDA and MapReduce workloads in general ([4] has reported more workloads run with underutilized system waiting for straggler jobs).

With Turbo, a speedup of 9% and 15% is measured for a 16 (baseline) and 32 mappers setting. This is above average Turbo speedups of general enterprise workloads for Ivy Bridge (Intel internal measurements). It can be explained as follows: towards the end of the workload execution, the few remaining map-jobs and reduce-jobs typically utilize low numbers of CPUs which leaves plenty of power budget for Turbo to kick-in, as was observed looking at the VTune overtime profile. The straggler jobs phenomena manifests more with 32 mappers as all jobs are run in single phase.

3.5. SMT

In order to improve core utilization, the Xeon architecture utilizes SMT (Simultaneous Multi-Threading) technology. SMT enables two software threads to run simultaneously in the same core. The additional thread improves core resources' utilization as the reorder window serves more instructions from independent streams. It is also power-efficient when compared to two physical cores. While SMT can benefit parallel workloads, performance may be limited due to possible contention on shared core resources. With that said, we expect SMT to speed up our inherently-parallel workload, because twice as many mappers can run in parallel. Furthermore, the modest IPC and *fairly distributed bottlenecks* characteristics hint at good speedup chances.

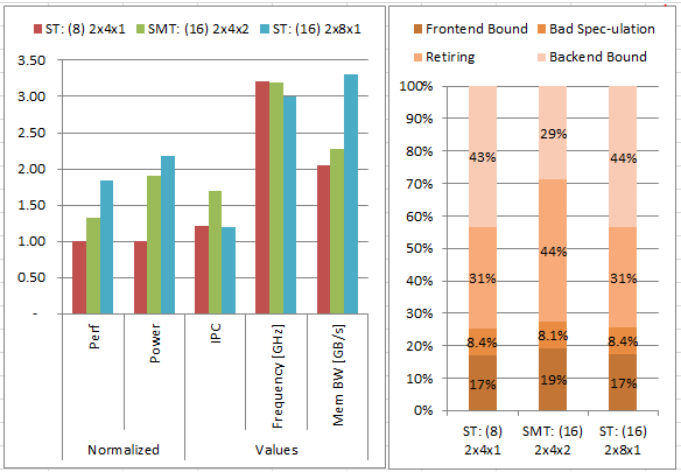

Figure 3: Multithread vs. Multicore Comparison

The two-fold *Figure 3* shows results for three configurations:

I) **2x4x1** (2 sockets, 4 cores, 1 thread) – the baseline Single Thread (ST) configuration with 8 mappers

II) **2x4x2** – An SMT configuration with 16 mappers

III) **2x8x1** – An ST configuration with 16 mappers. It serves as the SMT performance upper-bound for this workload

SMT provides 33% speedup over baseline, while the upper-bound configuration provided 83% (when additional single-threaded cores were added). The speedup comes with *a mere* 14% power reduction over upper-bound configuration. The smaller power reduction is (partly) due to the fact the SMT-on system ran with higher frequency; there is room for better efficiency though. Note that the upper-bound configuration runs at 200 MHz lower frequency, because the power management firmware has to reserve power consumption guard bands for twice as many cores[16].

The net Core's IPC has indeed improved with SMT. The TopDown breakdown on the right suggests the out-of-order core was able to shift a big portion of the Backend Bound bucket – pipeline stalls cause by lack of backend's memory or execution resources – into the Retiring bucket, which are pipeline slots utilized for useful work – as Retiring went from 31% up to 44%. Frontend Bound –actual stalls due to the fetch pipeline – slightly increased. This can be attributed to higher contention on the fetch bandwidth required to decode instructions with long flows (see section 5.3).

In section 5 we introduce a few targeted optimizations to mitigate some BDA bottlenecks. It is noteworthy that the potential of the presented optimizations is bigger when SMT is on. That is, the optimized- vs. baseline-version with SMT-on has ~5% *additional* speedup when compared with optimized vs. baseline with SMT-off. This is good news as most datacenters have SMT enabled[17]; hence our targeted optimizations will give more benefits in production systems. This also corroborates the identified bottlenecks are critical because the 2nd HW thread converts more of the alleviated stalls into speedup.

4. Application Level

VTune profiles suggest that 67% of the time is attributed to the `classifyDocument` function. With call-stacks, one can also navigate through the deep layers of software stacks[2].

4.1. Redundant Code

Let us examine the source code for `classifyDocument`. Mahout performs Bayesian classification, where each Wikipedia page (document) is classified into a category. The primary loop's pseudo code is depicted in *Listing 1*.

Issue: The code in red (lines 4-6) calculates the frequency of each word in the page (WordCount). This operation is independent of the current category, that is, that code is invariant with regard to the main loop at line 3.

Optimization: A quite simple optimization is needed to fix this issue: move the WordCount code outside the main loop. By doing so a speedup of 50% is achieved.

Using counters data and metrics, Instruction count was reduced by 48% and memory demand (total LLC misses) was

[2] A screenshot was omitted due to page count limit. E.g., the JVM executes Hadoop, which invokes user's map-function, which invokes Mahout.

reduced by a factor of two. The latter is backed up by a Miss Ratio reduction from 12% down to 7%, as well as a Mem BW reduction. This makes sense since data reuse was increased because the WordCount hash-structure is populated *once* and reused *many* times for all categories, unlike the original code.

Listing 1: Main loop pseudo-code

```
(1) Label classifyDocument(document) {
(2)   label = default_label;
(3)   foreach (category : categories) {
(4)     hash = new HashMap<String><int>;
(5)     foreach (word : document)
(6)       hash.update(word, 1);
(7)
(8)     result = 0;
(9)     foreach (pair-of [word, frequency] : hash)
(10)      result += frequency *
            featureWeight(category, word);
(11)
(12)    if (result is a maximum)
(13)      label = category;
(14)  }
(15)  return label;
(16) }
  where featureWeight is:
(17) double featureWeight(label, word) {
(18)   return - log[(getW("weight", label, word) +
    getW("params","αi")) /(getW("labelWeight",label) +
    getW("sumWeight","vocabCount"))];
(19) }
```

Note: each of these four `getWeight (getW)` calls is a lookup in a matrix in the Datastore pre-populated by the trained model.

4.2. Too-abstracted code

Vectorization can speed up a workload when multiple elements are processed with a single instruction. Furthermore, new extensions like the recent/upcoming AVX2/AVX512[18], feature a gather instruction to load multiple (non-adjacent) memory locations, thus increasing MLP. Such a property is desired for BDA-like workloads with big datasets and low Mem BW where the memory latency of multiple elements can be potentially overlapped.

Theoretically, vectorizing the loop in *Listing 1* should be doable by a compiler; because calculating the feature-weight of some word is independent from other words. Unfortunately, the source code has multiple layers of abstractions for getting the data elements, as shown in `featureWeight` body (lines 17-18). For example, in order to perform `getW("weight", label, word)`, instead of a simple two-dimension array lookup, the `getWeight` function receives everything as strings, converts strings to the matrix object, determines row- and column-indices, gets the vector object of that row, and only then does it obtain the specified element. There are more checks on the way that we did not mention for simplicity.

Note that such code is also unfriendly for upcoming programing language parallelization contracts such as Closures coming with JDK8[19], where the language permits a data-structure object to efficiently execute a user-defined operation over its elements.

5. µ/Architectural Level

In order to understand the causes of BDA bottlenecks, VTune and its recently featured TopDown method[2] are used. We examine each of the top hotspots, where we map bottlenecks back to the source/assembly code, reason what issues caused them, and mitigate the issues through ad-hoc optimizations in source-code.

Clarification note: We adopt a proof-by-optimization approach to confirm the identified issues are indeed critical. The ad-hoc optimizations are not a goal in themselves.

Figure 4 shows a schematic overview of the TopDown hierarchy applied to the biggest hotspots. The top three functions, namely `BayesAlgorithm::apply`, `*HashMap::indexOfKey` and `*HashMap::indexOfInsertion` account for over 65% of overall runtime. They are aggregated under the "Compiled Java Code" module, meaning these functions are JITed (as expected).

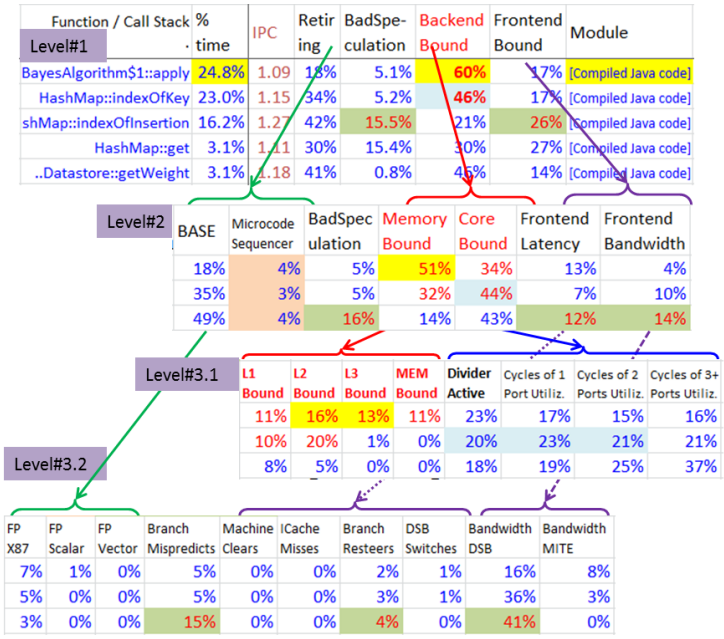

Figure 4: Top-Down Analysis for Top Hotspots

The one outstanding observation from this diagram is that a bottlenecks mixture is distributed over the top hotspots. That is, there is no one major bottleneck as in traditional HPC workloads; instead, each hotspot has a different bottleneck: the first hotspot is Memory Bound, the 2nd is Core Bound, and the 3rd has high Bad Speculation. This might tell us several things:

- The microarchitecture parameters are well-balanced.
- The Performance Analysis task of determining the true bottleneck is harder.
- Even if the correct bottleneck is detected, fixing it may not necessary result in a pure speedup.

5.1. Overview

Ivytown has a theoretical IPC of 4 (a four-wide superscalar core) while BDA has a modest IPC of 1.17. Let us examine data to see why higher IPC is not being reached. *Figure 1*

shows a "global application" of TopDown. That figure suggests *fairly distributed bottlenecks*, specifically:

(I) Reasonable[3] performance where 31% of the pipeline slots are used for retiring useful instructions. This yields the measured 1.17 IPC. Note that 10% of this bucket is delivered by the Micro-Sequencer slow fetch unit which handles "rarely-used" complex instructions.

(II) The main bottleneck is in the Backend under Memory subsystem with high fractions in L3 cache (highest) and External Memory. While waiting for memory is somewhat reasonable in a big data application; being L3-cache Bound needs understanding.

(III) The 2nd bottleneck is in the Backend as well; though the workload is Core (computation) Bound this time and the divider unit seems contended.

(IV) The 3rd bottleneck is Bad Speculation where a significant fraction of executed uops are thrown away due to Branch Mispredictions. Note that this causes Branch Resteers, which denotes penalty as the machine attempts to fetch from the corrected path[2].

In the remaining subsections, we describe the deep dive we performed to map these bottlenecks to their originating hotspots.

5.2. Cache misses in hashing

Issue: TopDown tags the 1st hotspot as Backend Bound → Memory Bound → {L2_Bound, L3_Bound}. The hotspot is mapped to this source code:

Listing 2: `Open<Object><Int>HashMap source code`

```
(1)     int indexOfKey(T key) {
(2)       int length = table.length;
(3)       int hash = key.hashCode() & 0x7FFFFFFF;
(4)       int i = hash % length;
(5)       int decrement=hash % (length - 2);
(6)       if (decrement == 0)
(7)         decrement = 1;
(8)       while ((state[i] != FREE) && …) {
(9)         i -= decrement;
(10)        if (i < 0)      i += length;
(11)      }
(12)      if (state[i] == FREE)    return -1;
(13)      return i;
(14)    }
```

A hash table is used to maintain the data. First, a hash value has to be determined – simply a modulo of a key-hash by the current hash table size. The critical cache miss is to retrieve the hash table size. The Java compiler translates the red line 2 into two bytecode instructions:

- `*getfield table`
- `*arraylength`

While the JVM implements them in assembly with:

- `mov r13d, dword ptr [r12+r9*8+0x2c]`
- `mov ecx, dword ptr [r12+r13*8+0xc]`

[3] SPEC CPU2006 average at 1.4 IPC on same microarchitecture [2]

VTune suggests 5% of runtime is spent on these two loads, where two-thirds of the accesses are hitting the L3 cache. The load-dereferencing using `r13d`, where the 2nd load address uses the data result of the 1st load, severely limits speculative execution and hardware prefetching.

Optimization: To mitigate this issue, we introduced a cached-copy of `table.length` as an attribute for the `Open<KeyType><ValueType>HashMap` class. The cached-copy is updated on re-hash operations. We applied the optimization to all instances and functions of this hashing construct, including `indexOfInsertion` with similar code.

A speedup of ~5% was achieved over baseline. A close examination shows CPI was reduced in the top 3 hotspots: Backend Bound was reduced and most of its fraction moved to the Retiring bucket; hence the speedup. Some fraction moved to other non-beneficial buckets of Frontend Bound and Bad Speculation.

Comparison to other methods: It is noteworthy that such a case would pitfall legacy analysis methods. For example, CycleAccounting's Load Latency bucket in Gooda[14] is calculated as:

$$\texttt{Load\_Latency} = \Sigma\ \texttt{Penalty}_i\ *\ \texttt{HitCacheLevel}_i$$

That is, a fixed cost is assigned per cache level. This would fail here as the total number of loads missing the L2 cache increased in the optimized version (due to the cached-copy); hence Gooda would tag a higher cost of Load Latency to the optimized version despite being faster.

5.3. Computation in hashing

Issue: TopDown tags the 2nd hotspot as Backend Bound → Core Bound → Divider. It is mapped to the previous same source code of `Open<KeyType><ValueType>HashMap`, though with a different template-types this time. The modulo operation of the hash calculation is translated into IDIV[18].

The code applies double-hashing to deal with collisions where *i* is the hash of the default index, and *decrement* is a 2nd hash used should there be a collision. Note the calculation of *decrement* is redundant if there was no collision. Due to pressure on the divider this slows down the program for this compute-bound code.

Optimization: To mitigate this issue, the blue lines 5-7 in *Listing 2* are moved to inside the *while* loop. That is, the 2nd hash computation is done only when required. A speedup of ~2% on top of optimization 5.2 was achieved.

Through close review comparing TopDown breakdown before and after; we observe the following (see *Table 3*):

- The net instruction count was reduced by 3%. This confirms the 2nd hash computation was redundant. Consequently, cycles when the divider was busy were reduced from 17% down to 13%.
- The net uop count was reduced by 6%. This makes sense as an IDIV is (inefficiently) translated into nine uops[13]. Our UPI metric was reduced from 1.03 down to 1.
- Frontend Bound cycles were reduced from 20% down to 16%. This can be explained by the fact that an instruction

decoded into >4 uops is provided by the MicroSequencer in the Core microarchitecture[13]. The fetch efficiency is also hurt by penalty cycles when switching from the fast fetch units to the MicroSequencer and vice versa.

Note: A 2% speedup might look too small. Recall that a small improvement at the end-point compute engine may have a large impact on the datacenter as a whole.

5.4. Polymorphic Objects

Recall performance optimization is an iterative process. When applying solutions for the first two issues, the workload remained Memory Bound where the 1st hotspot is tagged as Backend Bound → Memory Bound → L3_Bound.

Issue: Closely analyzing VTune data as well as JVM's own profiling information, we observe that the biggest samples are due to memory accesses that check the object-type of the parameters given to the `Open<K><V>HashMap` template. We find the JVM produces optimized code that targets the most frequent pair of `<String><int>` on one hand, but on the other hand, it kept asserting that those are the runtime types in every method invocation (for functional correctness of the optimized code). As this big data workload has many elements to lookup, process and compute in every main Bayesian classification step; the temporal locality was poor, causing some of the cache-lines holding the objects' types to be evicted outside core private caches.

Optimization: To mitigate this problem we generated a new class: `OpenStringIntHashMap`; functionally it is equivalent to `Open<String><int>HashMap` of the original generic-and-polymorphic code in mahout-collections[20]. Methods of new class are made `final`; it informs the Java runtime that the methods cannot be overridden by a subclass which shall enable further optimizations.

As a result of these optimizations, we measure: net instructions count reduction by 25%, a speedup of 7%, and energy saving of 6.3 Kilo Joules.

Summary: *Table 3* summarizes the potential and incremental speedup of the suggested optimizations.

Table 3: µ/Arch level Optimizations Summary

	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%

6. Comparison to Traditional Workloads and Analysis Methods

A subset of SPEC CPU2006 benchmarks covering the bottleneck domain is configured in rate 4-copy mode to exercise multicore and memory subsystem. Refer to [2] for a characterization of all SPEC benchmarks. Because enterprises serve a wide set of domains, five key workloads were chosen and measured on Sandy Bridge EP.

The CPU2006 workloads are grouped by their primary bottleneck to the left of BDA in *Figure 5*, while enterprise workloads are on the right. Memory BW, IPC and MLP are key legacy metrics. Two more memory metrics are overlaid with MLP: Miss Ratio and Off-core Bound, in order to contrast traditional- and accurate-methods. All metrics but the last two are plotted on the primary (left) Y-axis.

On the one hand, BDA performs better than enterprise workloads – it has at least 2x IPC of all but Search Engine. However, BDA has a much lower Memory BW demand with merely 2.9 GB/s vs. a range of 5.3–22.1. Ditto when BDA is compared to the Memory Bound SPEC benchmarks. On the other end, BDA is midway when compared to SPEC, considering IPC and MLP metrics. All SPEC workloads have decent MLP; we should consider this carefully though.

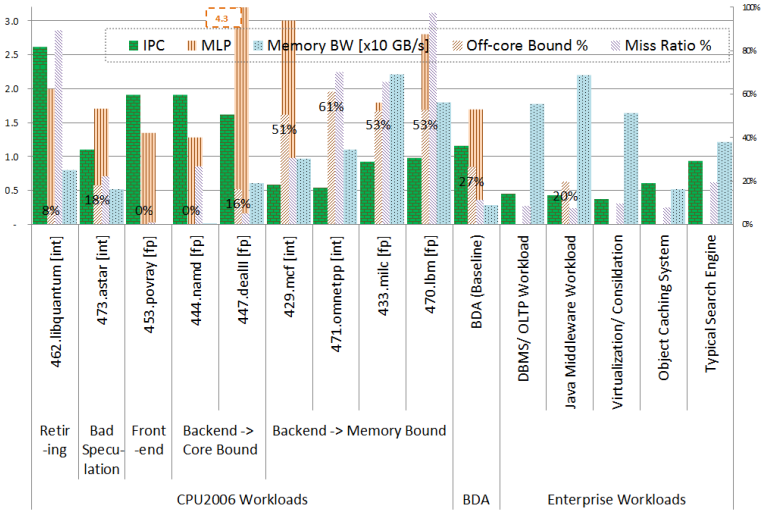

Figure 5: BDA vs Traditional Apps Comparison

The *Off-core Bound* metric – plotted also in % digits – suggests the Memory Bound SPEC workloads spend over half their time stalled on the memory subsystem outside the core. Indeed this metric inversely correlates well with performance (IPC). Interestingly, the legacy *Miss Ratio* metric has poor correlation with performance. This can be explained because the aggressive out-of-order core hides the cache misses' penalties[2]. For example, 462.libquantum has the best IPC with the worst Miss Ratio of 90% and a negligible Off-core Bound (this occurs as the benchmark has prefetch-friendly access pattern). In contrast, 429.mcf – a known memory troublemaker – has low IPC and a Miss-Ratio of 0.55 and 30%, respectively; while it is Off-core Bound 51% of its time, correctly reflecting its poor performance.

BDA has a modest performance. It has Off-core Bound of 27%, which is not an especially high fraction. This corroborates with our *fairly distributed bottlenecks* observation, and hints non-memory bottlenecks do exist. Note BDA has MLP of 1.7 similar to the MLP reported on older platform[1]. This is due to limited memory parallelism in code generation (the same setup was used in both works).

Next we perform a microarchitectural comparison. The BDA workload is analyzed using two methods on the left side of *Figure 6*: TopDown in color and analysis by [1] in grayscale. IPC increase is due to the newer platform. The same SPEC benchmarks are plotted on the right side.

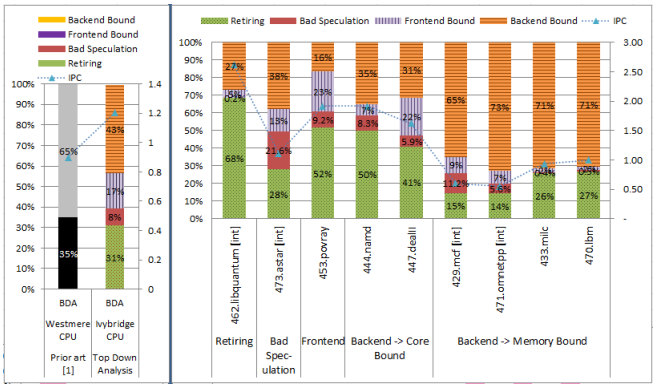

Figure 6: BDA vs. Traditional Apps at µArchitectural Level

While BDA is *partly* Backend Bound, it is not a solely-dominant bottleneck like the two rightmost applications (known for high memory BW demands[2]). In contrast, BDA is bound on memory latency, letting its signature be closer to 429.mcf and 471.omnetpp. The latter two have pointer-chasing patterns analogous to the BDA's hash traversal we observed. 453.povray has hard-to-predict data-dependent branches causing sizable Bad Speculation like in BDA (whose mispredictions are due to hash collision detection – see footnote 4). BDA has frontend issues due to the use of fetch-inefficient instructions.

7. Related Work

Ferdman et al.[1] introduced CloudSuite – a benchmark suite for scale-out workloads – and compared it to traditional benchmarks using raw counters. They concluded BDA in particular has low memory BW and no instruction cache misses – we observe the same. However, stalls overlap, speculate execution and superscalar issues were not handled well due to the use of traditional methods. This work reveals BDA has Bad Speculation & Frontend Bandwidth issues. We relate to other items along with [21] – their follow-up work.

HiBench[12] is a benchmark suite that characterizes Hadoop framework using coarse metrics; e.g., throughput, CPU or I/O utilization. A naive Bayesian algorithm with the same Wikipedia dataset is included as a machine-learning representative. HiBench reports its *training* part is mostly disk I/O. We analyzed the *classification* part of the same workload which is CPU Bound and provided broad and deep-dive analysis for its bottlenecks.

An analysis using trace-based system tool and performance counters on a Web search workload was done by [11]. They acknowledged the software stack increased complexity and used *CPI-breakdown* to show the workload characteristics resemble SPECcpu2000. While a single system-call optimization was tried, no deep-analysis was performed to explain the suboptimal performance. *CPI-breakdown* is a PowerPC bottleneck identification method that relies on a restricted set of predefined events. It was reported to underestimate superscalar and frontend issues[2].

A Pod *new processor theoretical design* that balances core count, cache and interconnect resources is proposed by [21] to fir characteristics of six scale-out workloads. While a 1-3x *aggregate* speedup is projected, [21] explicitly mentioned BDA was an exception where larger caches did help to capture a "secondary working set". Through a deep-dive analysis, our work identified the specific application constructs that do benefit from more cache, and suggested possible application, JVM and microarchitecture optimizations to *optimize in-production field platforms*.

More recently, [15] identified hash index lookups as the performance limiter in *traditional* DBMSs, and proposed a new accelerator to offload certain hash-join algorithms with variable *projected* speedups at *query-level*. Their custom accelerator does not suit BDA as it does not adopt a hash-join algorithm. Our work analyzed a different workload yet revealed similar findings (see sections 5.1 & 5.3). However, we proposed optimizations at today's platform stack hitting *measured* comparable speedups at *workload-level*.

Hadoop's Adolescence[4] surveyed usage of Hadoop in scientific workloads on three different clusters over long periods. They reported significant opportunities for optimizations and indicated penalties arise from over-reliance on default parameters. Our findings and optimizations corroborate that, and demonstrated concrete inefficiencies such as those shown in sections 3.3 and 4.1.

[12][22][23] developed techniques to improve utilization of Google datacenters. Machine heterogeneity in such datacenters – built of commodity multicores with shared resources – leads to unpredictable performance when collocating applications. [12] presents heuristics to find optimal thread-to-core mapping. CPI^2[22] uses CPI history to detect situations with victim-app (underperforming user-facing application), identify antagonist co-runner app, and avoid such collocations in future. Bubble-Up[23] predicts performance degradation and selects safe collocations that meet prescribed QoS constraints. Our findings optimize the application itself, thus complementing these approaches and improving the efficiency of the datacenter as a whole.

8. Conclusions

This work studied the characteristics of a Big Data Analytics (BDA) workload on a state-of-the-art cloud server. It intentionally focused on a single workload-platform that uses an in-memory database, in order to enable a deep-dive analysis to understand the root causes of the identified CPU bottlenecks. Data Analytics of CloudSuite[1] was chosen as a representative workload of a growing family of important applications. To perform the analysis, a comprehensive analysis method was customized and successfully used in a real setup.

The method is threefold, covering System, Application and µ/Architecture levels as depicted in *Table 4*. BDA–specific

new insights reported by this work are highlighted in light green. As can be seen, the use of a solid systematic method provides key insights into system behavior at all levels. A user provides only a fraction of the entire code when dealing with complicated systems such as Hadoop. Thus it is hard for her to locate where and how many inefficiencies the system experiences in terms of performance and power.

Table 4: Insights and Observations Summary

Level	*Parameter*	*Observation/ Optimization-potential*
System Introduced New Metrics	General Classification	CPU-Bound in-memory DB, no I/O or Hadoop system overhead
	JVM selection	Has major impact; e.g., Hotspot has 1.43x speedup over OpenJDK
	SMT	MT vs. CMP: 40% speedup, 20% power reduction
	Turbo	9-15% speedups. Benefits reduce- and straggler map-jobs
Application Uses call-stacks	Algorithm efficiency	Significant room for optimization. 50% sample speedup.
	Polymorphic objects	Big overheads. 25% uop reduction, 6% sample speedup
	Too abstracted code	Limits exploiting upcoming JVM and CPU parallelization features
µ/Architecture TopDown Analysis	JVM code generation	Memory dereferenceing limits MLP [Java generic]. 6% sample speedup
	CPU inefficiency for an ISA subset	Fetch bandwidth, contended Execution Units. E.g., Integer Divide
	µarch speculation[4]	About 16% uops waste power on miss-speculated paths

The *System Level* characterizes global parameters such as hardware multithreading or JVM setting. The *Application Level* analyzes the application code and use of abstracted programming language constructs. It double-checks the bulk of time is spent at expected areas – a check that is not trivial anymore for aforementioned systems with multiple software layers. The *µ/Architectural Level* further analyzes inefficiencies due to interactions among runtime, architecture and microarchitecture at the lowest level.

This paper revealed BDA suffers from overheads related to *managing* the data rather than *accessing* the data itself. Bottlenecks leading to this unexpected behavior are identified. Wide application-code inefficiency (or immaturity) are also reported, similar to the conclusions reached by [4] regarding other applications. Overall, *65% performance speedup* with significant power reduction was *measured*. Looking at the insights makes it clear that although Java is the language of choice for many users; its efficiency has to be improved.

As BDA uses in-memory database, we have high confidence the bottlenecks should hold for real setups. In fact, sections 3.4 and 5.3 show their cost is bigger in datacenters with SMT enabled[6][17]. With that said, multi-node system characterization is in our near future work radar.

[4] A subsection on Bad Speculation is skipped due to page count limit

9. Acknowledgements

This work would not have been possible without the help of colleagues at Intel. In particular, we thank Stas Bratanov and Sandhya Viswanathan for support using VTune and HotSpot, and Tal Katz for lab support. We also thank the anonymous reviewers for their valuable feedback.

10. References

[1] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ACM SIGARCH Computer Architecture News*, 2012, vol. 40, pp. 37–48.

[2] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," in Performance Analysis of Systems and Software (ISPASS), IEEE International Symposium on, 2014.

[3] Intel Corporation, "Intel® VTune™ Amplifier XE 2013." [Online].

[4] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence: an analysis of Hadoop usage in scientific workloads," *Proceedings of the VLDB Endowment*, pp. 853–864, 2013.

[5] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, 2011, pp. 283–294.

[6] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The Google cluster architecture," *Micro, Ieee*, , pp. 22–28, 2003.

[7] J. Varia, "Architecting for the cloud: Best practices," *Amazon Web Services*, 2010.

[8] J. Dean and Ghemawat, Sanjay, "Map-Reduce for Parallel Computing," in *6th Symposium on Operating Systems Design and Implementation*, 2004.

[9] "Apache Hadoop." [Online].

[10] "Apache Mahout: Scalable machine learning and data mining." [Online].

[11] R. W. Wisniewski, R. Azimi, M. Desnoyers, M. M. Michael, J. Moreira, D. Shiloach, and L. Soares, "Experiences understanding performance in a commercial scale-out environment," in *Euro-Par 2007 Parallel Processing*, Springer, 2007, pp. 139–149.

[12] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, 2010, pp. 41–51.

[13] Intel Corporation, "Intel® 64 and IA-32 Architectures Optimization Reference Manual," *Intel*. [Online].

[14] Google, "Gooda - PMU event analysis package" [Online].

[15] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: accelerating index traversals for in-memory databases," 2013, pp. 468–479.

[16] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova, "Evaluation of the Intel® Core™ i7 Turbo Boost feature," in *Workload Characterization, 2009. IISWC 2009.*

[17] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[18] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer Manuals," 2014. [Online].

[19] "JEP 126: Lambda Expressions & Virtual Extension Methods." [Online]. Available: http://openjdk.java.net/jeps/126.

[20] "mahout-collections - Apache Mahout - Apache Software Foundation." [Online].

[21] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, and E. Ozer, "Scale-out processors," in *Proceedings of the 39th International Symposium on Computer Architecture*, 2012, pp. 500–511.

[22] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI 2: CPU performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 379–391.

[23] J. Mars, L. Tang, K. Skadron, M. Soffa, and R. Hundt. "Increasing Utilization in Modern Warehouse-Scale Computers Using Bubble-Up."in *Micro, IEEE 32*, no. 3 (2012): 88-99.