2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)

Micro-architectural Characterization of Apache Spark on Batch and Stream Processing Workloads

Ahsan Javed Awan*, Mats Brorsson*, Vladimir Vlassov* and Eduard Ayguade†
*KTH Royal Institute of Technology,
{ajawan, matsbror, vladv}@kth.se
†Technical University of Catalunya (UPC),
Barcelona Super Computing Center,
eduard.ayguade@bsc.es

***Abstract*—While cluster computing frameworks are continuously evolving to provide real-time data analysis capabilities, Apache Spark has managed to be at the forefront of big data analytics for being a unified framework for both, batch and stream data processing. However, recent studies on micro-architectural characterization of in-memory data analytics are limited to only batch processing workloads. We compare the micro-architectural performance of batch processing and stream processing workloads in Apache Spark using hardware performance counters on a dual socket server. In our evaluation experiments, we have found that batch processing and stream processing has same micro-architectural behavior in Spark if the difference between two implementations is of micro-batching only. If the input data rates are small, stream processing workloads are front-end bound. However, the front end bound stalls are reduced at larger input data rates and instruction retirement is improved. Moreover, Spark workloads using DataFrames have improved instruction retirement over workloads using RDDs.**

I. Introduction

With a deluge in the volume and variety of data collecting, web enterprises (such as Yahoo, Facebook, and Google) run big data analytics applications using clusters of commodity servers. However, it has been recently reported that using clusters is a case of over-provisioning since most analytics jobs do not process really huge data sets and those modern scale-up servers are adequate to run analytics jobs [1]. Additionally, commonly used predictive analytics such as machine learning algorithms, work on filtered datasets that easily fit into the memory of modern scale-up servers. Moreover, the today's scale-up servers can have CPU, memory, and persistent storage resources in abundance at affordable prices. Thus we envision the small cluster of scale-up servers will be the preferable choice of enterprises in near future.

While Phoenix [2], Ostrich [3] and Polymer [4] are specifically designed to exploit the potential of a single scale-up server, they do not scale-out to multiple scale-up servers. Apache Spark [5] is getting popular in the industry because it enables in-memory processing, scales out to many of commodity machines and provides a unified framework for batch and stream processing of big data workloads. However, its performance on modern scale-up servers is not fully understood. Recent studies [6], [7] characterize the micro-architectural performance of in-memory data analytics with Spark on a scale-up server but they cover only batch processing workloads and they also do not quantify the impact of data velocity on the micro-architectural performance of Spark workloads. Knowing the limitations of modern scale-up servers for real-time streaming data analytics with Spark will help in achieving the future goal of improving the performance of real-time streaming data analytics with Spark on small clusters of scale-up servers.

Our contributions are:

- We characterize the micro-architectural performance of Spark-core, Spark MLlib, Spark SQL, GraphX and Spark Streaming.
- We quantify the impact of data velocity on the micro-architectural performance of Spark Streaming.

The rest of this paper is organized as follows. Firstly, we provide background and formulate the hypothesis in section 2. Secondly, we discuss the experimental setup in section 3, examine the results in section 4 and discuss the related work in section 5. Finally, we summarize the findings and give recommendations in section 6.

II. Background

A. Spark

Spark is a cluster computing framework that uses Resilient Distributed Datasets (RDDs) [5] which are immutable collections of objects spread across a cluster. Spark programming model is based on higher-order functions that execute user-defined functions in parallel. These higher-order functions are of two types: "Transformations" and "Actions". Transformations are lazy operators that create new RDDs, whereas Actions launch a computation on RDDs and generate an output. When a user runs an action on an RDD, Spark first builds a DAG of stages from the RDD lineage graph. Next, it splits the DAG into stages that contain pipelined transformations with narrow dependencies. Further, it divides each stage into tasks, where a task is a combination of data and computation. Tasks are assigned to executor pool of threads. Spark executes all tasks within a stage before moving on to the next stage. Finally, once all jobs are completed, the results are saved to file systems.

B. Spark MLlib

Spark MLlib [8] is a machine learning library on top of Spark-core. It contains commonly used algorithms related to

978-1-5090-3936-4/16 $31.00 © 2016 IEEE
DOI 10.1109/BDCloud-SocialCom-SustainCom.2016.20

IEEE computer society

collaborative filtering, clustering, regression, classification and dimensionality reduction.

C. Graph X

GraphX [9] enables graph-parallel computation in Spark. It includes a collection of graph algorithms. It introduces a new Graph abstraction: a directed multi-graph with properties attached to each vertex and edge. It also exposes a set of fundamental operators (e.g., aggregateMessages, joinVertices, and subgraph) and optimized variant of the Pregel API to support graph computation.

D. Spark SQL

Spark SQL [10] is a Spark module for structured data processing. It provides Spark with additional information about the structure of both the data and the computation being performed. This extra information is used to perform extra optimizations. It also provides SQL API, the DataFrames API, and the Datasets API. When computing a result the same execution engine is used, independent of which API/language is used to express the computation.

E. Spark Streaming

Spark Streaming [11] is an extension of the core Spark API for the processing of data streams. It provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. Internally, a DStream is represented as a sequence of RDDs. Spark streaming can receive input data streams from sources such as Kafka, Twitter, or TCP sockets. It then divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches. Finally, the results can be pushed out to file systems, databases or live dashboards.

F. Garbage Collection

Spark runs as a Java process on a Java Virtual Machine(JVM). The JVM has a heap space which is divided into young and old generations. The young generation keeps short-lived objects while the old generation holds objects with longer lifetimes. The young generation is further divided into eden, survivor1 and survivor2 spaces. When the eden space is full, a minor garbage collection (GC) is run on the eden space and objects that are alive from eden and survivor1 are copied to survivor2. The survivor regions are then swapped. If an object is old enough or survivor2 is full, it is moved to the old space. Finally when the old space is close to full, a full GC operation is invoked.

G. Spark on Modern Scale-up Servers

Our recent efforts on identifying the bottlenecks in Spark [6], [7] on Ivy Bridge machine shows that (i) Spark workloads exhibit poor multi-core scalability due to thread level load imbalance and work-time inflation, which is caused by frequent data access to DRAM and (ii) the performance of Spark workloads deteriorates severely as we enlarge the input data size due to significant garbage collection overhead. However, the scope of work is limited to batch processing workloads only, assuming that Spark streaming would have same micro-architectural bottlenecks. We revisit this assumption in this paper.

In this paper, we answer the following questions concerning real-time streaming data analytics running on modern scale-up servers using Apache Spark as a case study. Apache Spark defines the state of the art in big data analytics platforms exploiting data-flow and in-memory computing.

- Does micro-architectural performance remain consistent across batch and stream processing data analytics?
- How does data velocity affect the micro-architectural behavior of stream processing data analytics?

III. Methodology

Our study of micro-architectural characterization of real-time streaming data analytics is based on an empirical study of performance of batch and stream processing with Spark using representative benchmark workloads.

A. Workloads

This study uses batch processing and stream processing workloads, described in Table I and Table II respectively. Benchmarking big data analytics is an open research area, we, however, choose the workloads carefully. Batch processing workloads are the subset of BigdataBench [12] and HiBench [13], which are highly referenced benchmark suites in the big data domain. Stream processing workloads used in the paper are the superset of StreamBench [14] and also cover the solution patterns for real-time streaming analytics [15].

The source codes for Word Count, Grep, Sort, and NaiveBayes are taken from BigDataBench [12], whereas the source codes for K-Means, Gaussian, and Sparse NaiveBayes are taken from Spark MLlib examples available along with Spark distribution. Likewise, the source codes for stream processing workloads are also available from Spark Streaming examples. Big Data Generator Suite (BDGS), an open source tool is used to generate synthetic data sets based on raw data sets [16].

B. System Configuration

Table IV shows details about our test machine. Hyper-Threading and Turbo-boost are disabled through BIOS as per Intel Vtune guidelines to tune software on the Intel Xeon processor E5/E7 v2 family [17]. With Hyper-Threading and Turbo-boost disabled, there are 24 cores in the system operating at the frequency of 2.7 GHz.

Table V lists the parameters of JVM and Spark after tuning. For our experiments, we configure Spark in local mode in which driver and executor run inside a single JVM. We use HotSpot JDK version 7u71 configured in server mode (64 bit). The Hotspot JDK provides several parallel/concurrent GCs out of which we use Parallel Scavenge (PS) and Parallel Mark Sweep for young and old generations respectively as recommended in [7]. The heap size is chosen such that the memory consumed is within the system. The details on Spark internal parameters are available [18].

TABLE I: Batch Processing Workloads

Spark Library	Workload	Description	Input data-sets
Spark Core	Word Count (Wc)	counts the number of occurrence of each word in a text file	Wikipedia Entries
	Grep (Gp)	searches for the keyword "The" in a text file and filters out the lines with matching strings to the output file	
	Sort (So)	ranks records by their key	Numerical Records
	NaiveBayes (Nb)	runs sentiment classification	Amazon Movie Reviews
Spark MLlib	K-Means (Km)	uses K-Means clustering algorithm from Spark MLlib. The benchmark is run for 4 iterations with 8 desired clusters	Numerical Records
	Sparse NaiveBayes (Snb)	uses NaiveBayes classification algorithm from Spark MLlib	
	Support Vector Machines (Svm)	uses SVM classification algorithm from Spark MLlib	
	Logistic Regression (Logr)	uses Logistic Regression algorithm from Spark MLlib	
Graph X	Page Rank (Pr)	measures the importance of each vertex in a graph. The benchmark is run for 20 iterations	Live Journal Graph
	Connected Components (Cc)	labels each connected component of the graph with the ID of its lowest-numbered vertex	
	Triangles (Tr)	determines the number of triangles passing through each vertex	
Spark SQL	Aggregation (SqlAg)	implements aggregation query from BigdataBench using DataFrame API	Tables
	Join (SqlJo)	implements join query from BigdataBench using DataFrame API	

TABLE II: Stream Processing Workloads

Workload	Description	Input data stream
Streaming Kmeans (Skm)	uses streaming version of K-Means clustering algorithm from Spark MLlib.	Numerical Records
Streaming Linear Regression (Slir)	uses streaming version of Linear Regression algorithm from Spark MLlib.	
Streaming Logistic Regression (Slogr)	uses streaming version of Logistic Regression algorithm from Spark MLlib.	
Network Word Count (NWc)	counts the number of words in text received from a data server listening on a TCP socket every 2 sec and print the counts on the screen. A data server is created by running Netcat (a networking utility in Unix systems for creating TCP/UDP connections)	Wikipe-dia data
Network Grep (Gp)	counts how many lines have the word "the" in them every sec and prints the counts on the screen.	
Windowed Word Count (WWc)	generates every 10 seconds, word counts over the last 30 sec of data received on a TCP socket every 2 sec.	
Stateful Word Count (StWc)	counts words cumulatively in text received from the network every sec starting with initial value of word count.	
Sql Word Count (SqWc)	uses DataFrames and SQL to count words in text received from the network every 2 sec.	
Click stream Error Rate Per Zip Code (CErpz)	returns the rate of error pages (a non 200 status) in each zipcode over the last 30 sec. A page view generator generates streaming events over the network to simulate page views per second on a website.	Click streams
Click stream Page Counts (CPc)	counts views per URL seen in each batch.	
Click stream Active User Count (CAuc)	returns number of unique users in last 15 sec	
Click stream Popular User Seen (CPus)	look for users in the existing dataset and print it out if there is a match	
Click stream Sliding Page Counts (CSpc)	counts page views per URL in the last 10 sec	
Twitter Popular Tags (TPt)	calculates popular hashtags (topics) over sliding 10 and 60 sec windows from a Twitter stream.	Twitter Stream
Twitter Count Min Sketch (TCms)	uses the Count-Min Sketch, from Twitter's Algebird library, to compute windowed and global Top-K estimates of user IDs occurring in a Twitter stream	
Twitter Hyper Log Log (THll)	uses HyperLogLog algorithm, from Twitter's Algebird library, to compute a windowed and global estimate of the unique user IDs occurring in a Twitter stream.	

C. Measurement Tools and Techniques

We use Intel Vtune Amplifier [19] to perform general micro-architecture exploration and to collect hardware performance counters. All measurement data are the average of three measure runs; Before each run, the buffer cache is cleared to avoid variation in the execution time of benchmarks. Through concurrency analysis in Intel Vtune, we find that executor pool threads in Spark start taking CPU time after 10 seconds. Hence, hardware performance counter values are collected after the ramp-up period of 10 seconds. For batch processing workloads, the measurements are taken for the entire run of the applications and for stream processing workloads, the measurements are taken for 180 seconds as the sliding interval and duration of windows in streaming workloads considered are much less than 180 seconds.

We use top-down analysis method proposed by Yasin [20] to study the micro-architectural performance of the workloads. Earlier studies on profiling of big data workloads show the efficacy of this method in identifying the micro-architectural bottlenecks [6], [21], [22]. Super-scalar processors can be conceptually divided into the "front-end" where instructions are fetched and decoded into constituent operations, and the "back-end" where the required computation is performed. A pipeline slot represents the hardware resources needed to process one micro-operation. The top-down method assumes that for each CPU core, there are four pipeline slots available per clock cycle. At issue point, each pipeline slot is classified into one of four base categories: Front-end Bound, Back-end Bound, Bad Speculation and Retiring. If a micro-operation is issued in a given cycle, it would eventually either get retired or canceled. Thus it can be attributed to either Retiring or Bad Speculation respectively. Pipeline slots that could not be filled with micro-operations due to problems in the front-end are attributed to Front-end Bound category whereas pipeline slot where no micro-operations are delivered due to a lack of required resources for accepting more micro-operations in the back-end of the pipeline are identified as Back-end Bound.

The top-down method requires the metrics described in Table VI, whose definition are taken from Intel Vtune on-line help [19].

IV. Evaluation

A. Does micro-architectural performance remain consistent across batch and stream processing data analytics?

As stream processing is micro-batch processing in Spark, we hypothesize batch processing and stream processing to exhibit same micro-architectural behavior. Figure 1a shows the IPC values of batch processing workloads range between 1.78 to 0.76, whereas IPC values of stream processing workloads also range between 1.85 to 0.71. The IPC values of word

TABLE III: Converted Spark Operations in Workloads

Workload	Converted Spark Operation
Wc	Map, ReduceByKey, SaveAsTextFile
Gp	Filter, SaveAsTextFile
So	Map, SortByKey, SaveAsTextFile
Nb	Map, Collect, SaveAsTextFile
Snb	Map, RandomSplit, Filter, CombineByKey
Km	Map, MapPartitions, MapPartitionsWithIndex, FlatMap,Zip, Sample, ReduceByKey,
Svm Logr	Map, MapPartitions, MapPartionswithIndex, Zip, Sample, RandomSplit,Filter,MakeRDD,Union, TreeAggregate, CombineByKey, SortByKey
Pr Cc Tr	Coalesce, MapPartitionswithIndex, MapPartitions, Map, PartitionBy, ZipPartitions
SqlAgg	Map, MapPartitions, TungstenProject, TungstenExchange, TungstenAggregate, ConvertToSafe
SqlJo	Map, MapPartitions, SortMergeJoin, TungstenProject, TungstenExchange, TungstenSort, ConverToSafe
SqWc	FlatMap, ForeachRDD, TungstenExchange, TungstenAggregate, ConvertToSafe
NWc	FlatMap, Map, ReduceByKey
NGp	Filter, Count
WWc	FlatMap, Map, ReduceByKeyAndWindow
StWc	FlatMap, Map, UpdateStateByKey
CErPz	FlatMap, Map, Window, GroupByKey
CAuc	FlatMap, Map, Window, GroupByKey, Count
CPus	FlatMap, Map, Parallelize, ForeachRDD
CPc	FlatMap, Map, CountByValue
CSPc	FlatMap, Map, CountByValueAndWindow
Tpt Tcms	FlatMap, Map, ReduceByKeyAndWindow, Transform Map, MapPartitions, Reduce, ForeachRDD, ReduceByKey,
Thll	Map, MapPartitions, Reduce

TABLE IV: Machine Details.

Component	Details	
Processor	Intel Xeon E5-2697 V2, Ivy Bridge micro-architecture	
	Cores	12 @ 2.7GHz (Turbo up 3.5GHz)
	Threads	2 per Core (when Hyper-Threading is enabled)
	Sockets	2
	L1 Cache	32 KB for Instruction and 32 KB for Data per Core
	L2 Cache	256 KB per core
	L3 Cache (LLC)	30MB per Socket
Memory	2 x 32GB, 4 DDR3 channels, Max BW 60GB/s per Socket	
OS	Linux Kernel Version 2.6.32	
JVM	Oracle Hotspot JDK 7u71	
Spark	Version 1.5.0	

count (Wc) and grep (Gp) are very close to their stream processing equivalents, i.e. network word count (NWc) and network grep (NGp). Likewise, the pipeline slots breakdown in Figure 1b for the same workloads are quite similar. This implies that batch processing and stream processing will have same micro-architectural behavior if the difference between two implementations is of micro-batching only.

Sql Word Count(SqWc), which uses the Dataframes has better IPC than both word count (Wc) and network word count (NWc), which use RDDs. Aggregation (SqlAg) and Join (SqlAg) queries which also use DataFrame API have IPC values higher than most of the workloads using RDDs. One can see the similar pattern for retiring slots fraction in Figure 1b. Sql Word Count (SqWc) exhibits 25.56% less back-

TABLE V: Spark and JVM Parameters for Different Workloads.

Parameters	Batch Processing Workloads		Stream Processing Workloads
	Spark-Core, Spark-SQL	Spark MLlib, Graph X	
spark.storage.memoryFraction	0.1	0.6	0.4
spark.shuffle.memoryFraction	0.7	0.4	0.6
spark.shuffle.consolidateFiles	true		
spark.shuffle.compress	true		
spark.shuffle.spill	true		
spark.shuffle.spill.compress	true		
spark.rdd.compress	true		
spark.broadcast.compress	true		
Heap Size (GB)	50		
Old Generation Garbage Collector	PS Mark Sweep		
Young Generation Garbage Collector	PS Scavenge		

TABLE VI: Metrics for Top-Down Analysis of Workloads

Metrics	Description
IPC	average number of retired instructions per clock cycle
DRAM Bound	how often CPU was stalled on the main memory
L1 Bound	how often machine was stalled without missing the L1 data cache
L2 Bound	how often machine was stalled on L2 cache
L3 Bound	how often CPU was stalled on L3 cache, or contended with a sibling Core
Store Bound	how often CPU was stalled on store operations
Front-End Bandwidth	fraction of slots during which CPU was stalled due to front-end bandwidth issues
Front-End Latency	fraction of slots during which CPU was stalled due to front-end latency issues
ICache Miss Impact	fraction of cycles spent on handling instruction cache misses
Cycles of 0 ports Utilized	the number of cycles during which no port was utilized.

end bound slots than streaming network word count (NWc) because sql word count (SqWc) shows 64% less DRAM bound stalled cycles than network word count (NWc) and hence consumes 25.65% less memory bandwidth than network word count (NWc). Moreover, the execution units inside the core are less starved in sql word count as the fraction of clock cycles during which no ports are utilized, is 5.23% less than in network wordcount. The difference in performance is because RDDs use Java objects based row representation, which have high space overhead whereas DataFrames use new Unsafe Row format where rows are always 8-byte word aligned (size is multiple of 8 bytes) and equality comparison and hashing are performed on raw bytes without additional interpretation. This implies that Dataframes have the potential to improve the micro-architectural performance of Spark workloads.

The DAG of both windowed word count (Wwc) and twitter

popular tags (Tpt) consists of "map" and "reduceByKeyAndWindow" transformations (see Table III) but the breakdown of pipeline slots in both workloads differ a lot. The back-end bound fraction in windowed word count (Wwc) is 2.44x larger and front-end bound fraction is 3.65x smaller than those in twitter popular tags (Tpt). The DRAM bound stalled cycles in windowed word count (Wwc) are 4.38x larger and L3 bound stalled cycles are 3.26x smaller than those in twitter popular tags (Tpt). The fraction of cycles during which 0 port is utilized, however, differ only by 2.94%. Icache miss impact is 13.2x larger in twitter popular tags (Tpt) than in windowed word count (Wwc). The input data rate in windowed word count (Wwc) is 10,000 events/s whereas in twitter popular tags (Tpt), it is 10 events/s. Since the sampling interval is 2s, the working set of a windowing operation in windowed word count (Wwc) with 30s window length is 15 x 10,000 events where the working set of a windowing operation in twitter popular tags (Tpt) with 60s window length is 30 x 10 events. The working set in windowed word count (Wwc) is 500x larger than that in twitter popular tags (Tpt), The 30 MB last level cache is sufficient enough for the working set of Tpt but not for windowed word count (Wwc). That's why windowed word count (Wwc) also consumes 24x more bandwidth than twitter popular tags (Tpt).

Click stream sliding page count (CSpc) also uses similar "map" and "countByValueAndWindow" transformations (see Table III) and the input data rate is also the same as in windowed word count (Wwc) but the back-end bound fraction and DRAM bound stalls are smaller in click stream sliding page count (CSpc) than in windowed word count (Wwc). Again the working set in Click stream sliding page count (CSpc) with 10s window length is 5 x 10,000 events which three times less than the working set in windowed word count (Wwc).

CErpz and CAuc both use "window", "map" and "groupbyKey" transformations (see Table III) but the front-end bound fraction and icache miss impact in CAuc is larger than in CErpz. However, back-end bound fraction, DRAM bound stalled cycles, memory bandwidth consumption are larger in CErpz than in CAuC. The retiring fraction is almost same in both workloads. The difference is again the working set. The working set in CErpz with the window length of 30 seconds is 15 x 10,000 events which are 3x larger than in CAuc with the window length of 10 seconds. This implies that with larger working sets, Icache miss impact can be reduced.

B. How does data velocity affect micro-architectural performance of in-memory data analytics with Spark?

In order to answer the question, we compare the micro-architectural characteristics of stream processing workloads at input data rates of 10, 100, 1000 and 10,000 events per second. Figure 2a shows that CPU utilization increases only modestly up to 1000 events/s after which it increases up to 20%. Likewise IPC in figure 2b increases by 42% in CSpc and 83% in CAuc when input rate is increased from 10 to 10,000 events per second.

The pipeline slots breakdown in Figure 2c shows that when the input data rates are increased from 10 to 10,000 events/s, fraction of pipeline slots being retired increases by 14.9% in CAuc and 8.1% in CSpc because in CAuc, the fraction of front-end bound slots and bad speculation slots decrease by 9.3% and 8.1% respectively and the back-end bound slots increase by only 2.5%, whereas in CSpc, the fraction of front-end bound slots and bad speculation slots decrease by 0.4% and 7.4% respectively and the back-end bound slots increase by only 0.4%. The memory subsystem stalls break down in Figure 2d show that L1 bound stalls increase, L3 bound stalls decrease and DRAM bound stalls increase at high data input rate, e.g in CErpz, L3 bound stalls and DRAM bound stalls remain roughly constant at 10, 100 and 1000 events/s because the working sets are still not large enough to create an impact but at 10,000 events/s, the working sets does not fit into the last level cache and thus DRAM bound stalls increase by approximately 20% while the L3 bound stalls decrease by the same amount. This is also evident from Figure 2f, where the memory bandwidth consumption is constant at 10, 100 and 1000 events/s and then increases significantly at 10,000 events/s. Larger working sets translate into better utilization of functional units as the number of clock cycles during which no ports are utilized decrease at higher input data rates. Hence input data rates should be high enough to provide working sets large enough to keep the execution units busy.

V. Related Work

Several studies characterize the behaviour of big data workloads and identify the mismatch between the processor and the big data applications [12], [21], [23]–[27]. Ferdman et al. [23] show that scale-out workloads suffer from high instruction cache miss rates. Large LLC does not improve performance and off-chip bandwidth requirements of scale-out workloads are low. Zheng et al. [28] infer that stalls due to kernel instruction execution greatly influence the front end efficiency. However, data analysis workloads have higher IPC than scale-out workloads [24]. They also suffer from notable front-end stalls but L2 and L3 caches are effective for them. Wang et al. [12] conclude the same about L3 caches and L1 I-Cache miss rates despite using larger data sets. Deep dive analysis [21] reveal that big data analysis workload is bound on memory latency but the conclusion can not be generalized. None of the above-mentioned works consider frameworks that enable in-memory computing of data analysis workloads.

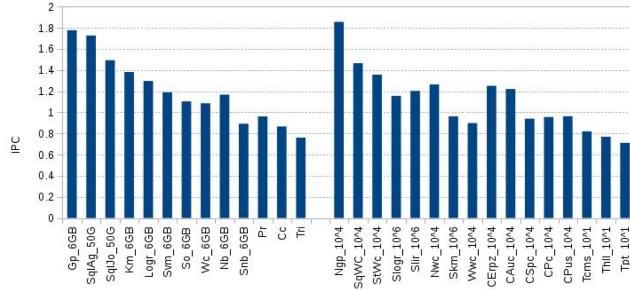
Ruirui et-al [14] have compared throughput, latency, data reception capability and performance penalty under a node failure of Apache Spark with Apache Storm. Miyuru et-al [29] have compared the performance of five streaming applications on System S and S4. Jagmon et-al [30] have analyzed the performance of S4 in terms of scalability, lost events, resource usage, and fault tolerance. Our work analyzes the micro-architectural performance of Spark Streaming.

VI. Conclusion

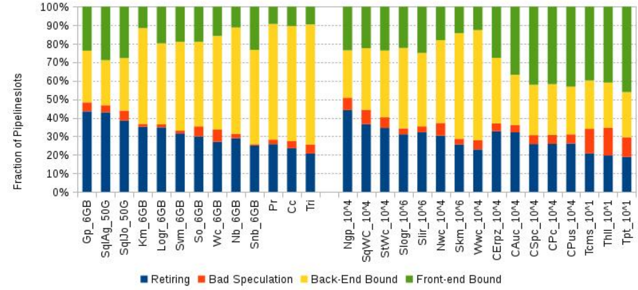
We have reported a deep dive analysis of in-memory data analytics with Spark on a large scale-up server.

The key insights we have found are as follows:

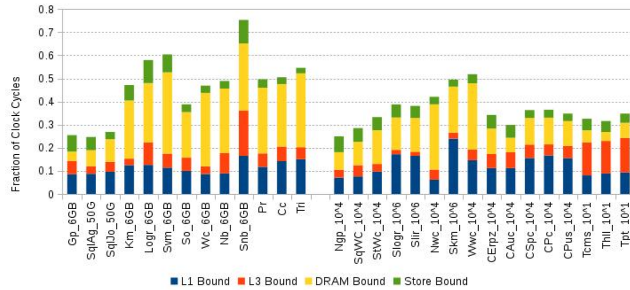
- Batch processing and stream processing has same micro-architectural behavior in Spark if the difference between two implementations is of micro-batching only.



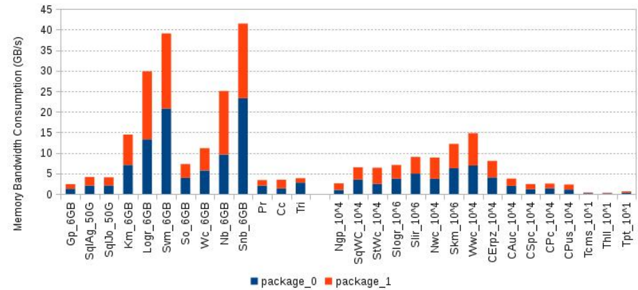
(a) IPC values of stream processing workloads lie in the same range as of batch processing workloads



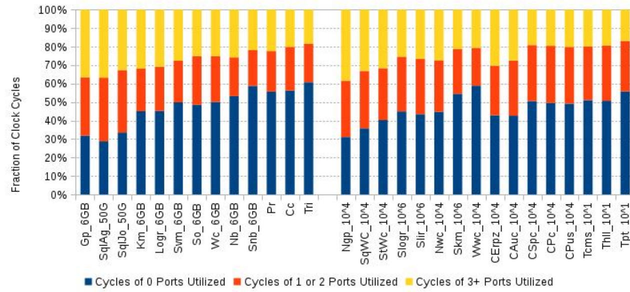
(b) Majority of stream processing workloads are back-end bound as that of batch processing workloads



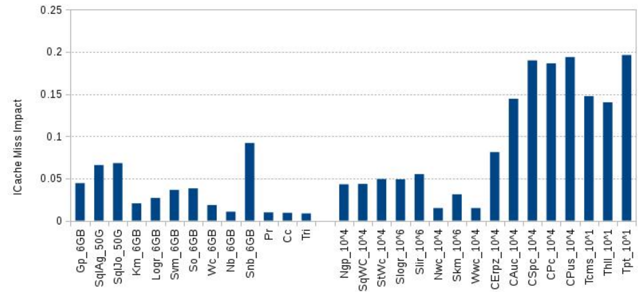
(c) Stream processing workloads are also DRAM bound but their fraction of DRAM bound stalled cycles is lower than that of batch processing workloads



(d) Memory bandwidth consumption of machine learning based batch processing workloads is higher than other Spark workloads



(e) Execution units starve both in batch in stream processing workloads



(f) ICache miss impact in majority of stream processing workloads is similar to batch processing workloads

Fig. 1: Comparison of micro-architectural characteristics of batch and stream processing workloads

- Spark workloads using DataFrames have improved instruction retirement over workloads using RDDs.
- If the input data rates are small, stream processing workloads are front-end bound. However, the front end bound stalls are reduced at larger input data rates and instruction retirement is improved.

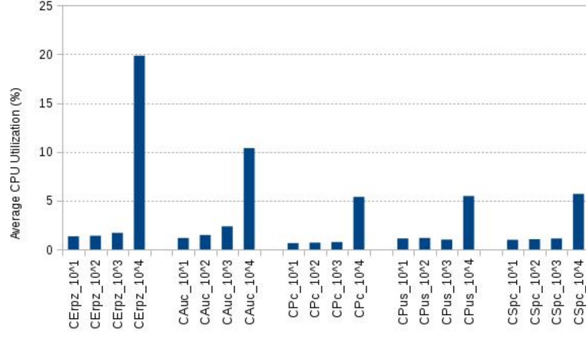
We recommend Spark users to prefer DataFrames over RDDs while developing Spark applications. Computer architects rely heavily on cycle accurate simulators to evaluate novel designs for processor and memory. Since simulators are quite slow, computer architects tend to prefer smaller input data sets. Due to large inconsistencies in the micro-architectural behavior with data velocity, computer architects need to simulate their proposals for stream processing workloads at large input data rates.

ACKNOWLEDGMENTS

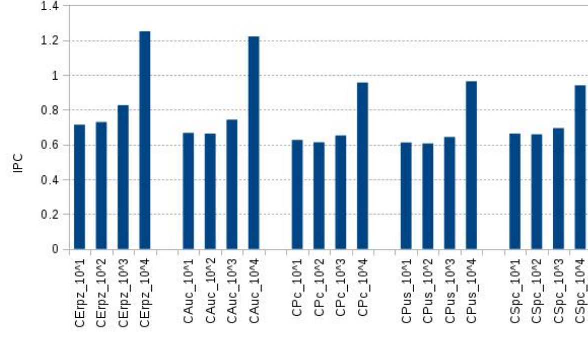
This work is supported by Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) program funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission. It is also supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology through TIN2015-65316-P project and by the Generalitat de Catalunya (contract 2014-SGR-1051). We thank Ananya Muddukrishna for his comments on the first draft of the paper. We also thank the anonymous reviewers for their constructive feedback.

REFERENCES

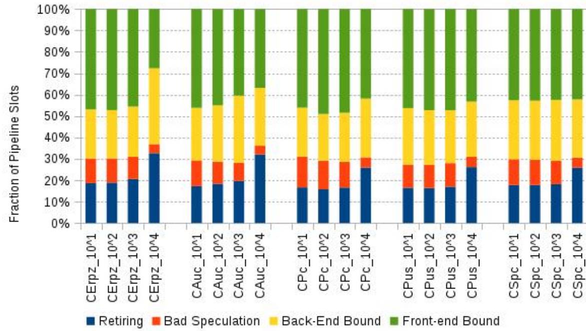
[1] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. I. T. Rowstron, "Scale-up vs scale-out for hadoop: time to rethink?" in *ACM*



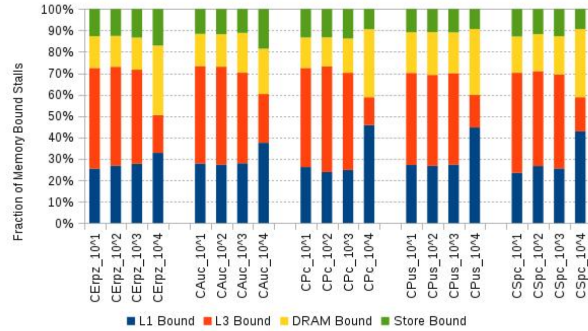
(a) CPU utilization increases with data velocity



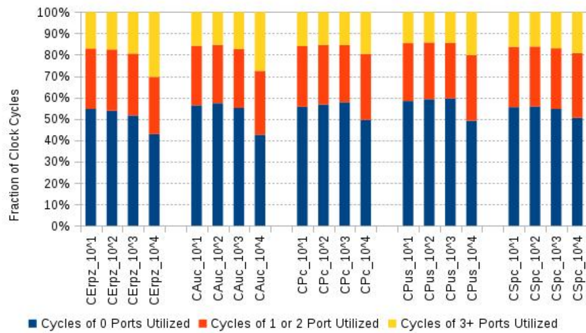
(b) Better IPC at higher data velocity



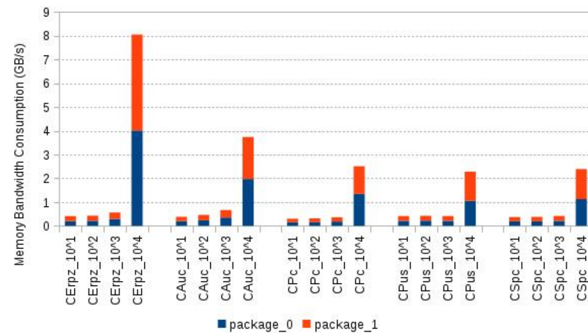
(c) Front-end bound stalls decrease and fraction of retiring slots increases with data velocity



(d) Fraction of L1 Bound stalls increases, L3 Bound stalls decreases and DRAM bound stalls increases with data velocity



(e) Functional units inside exhibit better utilization at higher data velocity



(f) Memory bandwidth consumption increases with data velocity

Fig. 2: Impact of Data Velocity on Micro-architectural Performance of Spark Streaming Workloads

Symposium on Cloud Computing, SOCC, 2013, p. 20.

[2] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system," in *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 198–207.

[3] R. Chen, H. Chen, and B. Zang, "Tiled-mapreduce: Optimizing resource usages of data-parallel applications on multicore with tiling," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10, 2010, pp. 523–534.

[4] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2015, pp. 183–193.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, San Jose, CA, 2012, pp. 15–28.

[6] A. Javed Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Performance characterization of in-memory data analytics on a modern cloud server," in *Big Data and Cloud Computing (BDCloud), 2015 IEEE Fifth International Conference on*. IEEE, 2015, pp. 1–8.

[7] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, *Big Data Benchmarks, Performance Optimization, and Emerging Hardware: 6th Workshop, BPOE 2015, Kohala, HI, USA, August 31 - September 4, 2015. Revised Selected Papers*. Springer International Publishing, 2016, ch. How Data Volume Affects Spark Based Data Analytics on a Scale-up Server, pp. 81–92.

[8] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning

in apache spark," *arXiv preprint arXiv:1505.06807*, 2015.

[9] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 599–613.

[10] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1383–1394.

[11] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012, pp. 10–10.

[12] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services," in *20th IEEE International Symposium on High Performance Computer Architecture, HPCA*, 2014, pp. 488–499.

[13] 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.

[14] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*. IEEE, 2014, pp. 69–78.

[15] S. Perera and S. Suhothayan, "Solution patterns for realtime streaming analytics," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. ACM, 2015, pp. 247–255.

[16] Z. Ming, C. Luo, W. Gao, R. Han, Q. Yang, L. Wang, and J. Zhan, "BDGS: A scalable big data generator suite in big data benchmarking," in *Advancing Big Data Benchmarks*, ser. Lecture Notes in Computer Science, 2014, vol. 8585, pp. 138–154.

[17] Using Intel VTune Amplifier XE to Tune Software on the Intel Xeon Processor E5/E7 v2 Family. https://software.intel.com/en-us/articles/using-intel-vtune-amplifier-xe-to-tune-software-on-the-intel-xeon-processor-e5e7-v2-family.

[18] Spark configuration. https://spark.apache.org/docs/1.5.1/configuration.html.

[19] Intel Vtune Amplifier XE 2013. http://software.intel.com/en-us/node/544393.

[20] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, 2014, pp. 35–44.

[21] A. Yasin, Y. Ben-Asher, and A. Mendelson, "Deep-dive analysis of the data analytics workload in cloudsuite," in *Workload Characterization (IISWC), IEEE International Symposium on*, Oct 2014, pp. 202–211.

[22] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, D. Brooks, S. Campanoni, K. Brownell, T. M. Jones *et al.*, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015, pp. 158–169.

[23] 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 *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, 2012, pp. 37–48.

[24] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers," in *Workload Characterization (IISWC), IEEE International Symposium on*, 2013, pp. 66–76.

[25] T. Jiang, Q. Zhang, R. Hou, L. Chai, S. A. McKee, Z. Jia, and N. Sun, "Understanding the behavior of in-memory computing workloads," in *Workload Characterization (IISWC), IEEE International Symposium on*, 2014, pp. 22–30.

[26] Z. Jia, J. Zhan, L. Wang, R. Han, S. A. McKee, Q. Yang, C. Luo, and J. Li, "Characterizing and subsetting big data workloads," in *Workload Characterization (IISWC), IEEE International Symposium on*, 2014, pp. 191–201.

[27] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. Swift, "Performance analysis of the memory management unit under scale-out workloads," in *Workload Characterization (IISWC), IEEE International Symposium on*, Oct 2014, pp. 1–12.

[28] C. Zheng, J. Zhan, Z. Jia, and L. Zhang, "Characterizing OS behavior of scale-out data center workloads," in *The Seventh Annual Workshop on the Interaction amongst Virtualization,Operating Systems and Computer Architecture(WIVOSCA2013) held in conjunction with The 40th International Symposium on Computer Architecture*, 2013.

[29] M. Dayarathna and T. Suzumura, "A performance analysis of system s, s4, and esper via two level benchmarking," in *Quantitative Evaluation of Systems*. Springer, 2013, pp. 225–240.

[30] J. Chauhan, S. A. Chowdhury, and D. Makaroff, "Performance evaluation of yahoo! s4: A first look," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012 Seventh International Conference on*. IEEE, 2012, pp. 58–65.