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

Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov and Eduard Ayguade

--Characterize the micro-architectural performance of Spark-core, Spark MLlib, Spark SQL, GraphX and Spark Streaming.

---Quantify the impact of data velocity on the micro- architectural performance of Spark Streaming.

Background

Spark

---Actions launch a computation on RDDs and generate an output

---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

Spark MLib

---a machine learning library on top of Spark-core

---It contains commonly used algorithms related to collaborative filtering, clustering, regression, classification and dimensionality reduction

Graph X

---enables graph-parallel computation in Spark. It includes a collection of graph algorithms

Spark SQL

---a Spark module for structured data processing

Spark Stream

---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.

Background

Garbage Collection

---The JVM has a heap space which is divided into young and old generations

---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.

---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

Spark on Modern Scale-up Servers

---Bottleneck

---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

---the performance of Spark workloads deteriorates severely as we enlarge the input data size due to significant garbage collection overhead

---the scope of work is limited to batch processing workloads only, assuming that Spark streaming would have same micro-architectural bottlenecks

Methodology ---Workloads

Batch processing workloads are the subset of BigdataBench and HiBench

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	

Methodology ---Workloads

Stream processing workloads used in the paper are the superset of StreamBench and also cover the solution patterns for real-time streaming analytics

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)	Wikipedia 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.	

---All measurement data are the average of three measure runs

---executor pool threads in Spark start taking CPU time after 10 seconds
---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

---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

---Intel Vtune Amplifier to perform general micro-architecture exploration and to collect hardware performance counters

Top-down analysis method

Front-end: where instructions are fetched and decoded into constituent operations

Back-end: where the required computation is performed

Pipeline slot: hardware resources needed to process one micro-operation

Assume each CPU core has four pipeline slots available per clock cycle

At issue point, each pipeline slot is classified into one of four base categories

---Front-end Bound: Pipeline slots that could not be filled with micro-operations due to problems in the front-end

---Back-end Bound: no micro-operations are delivered due to a lack of required resources for accepting more micro-operations in the back-end of the pipeline

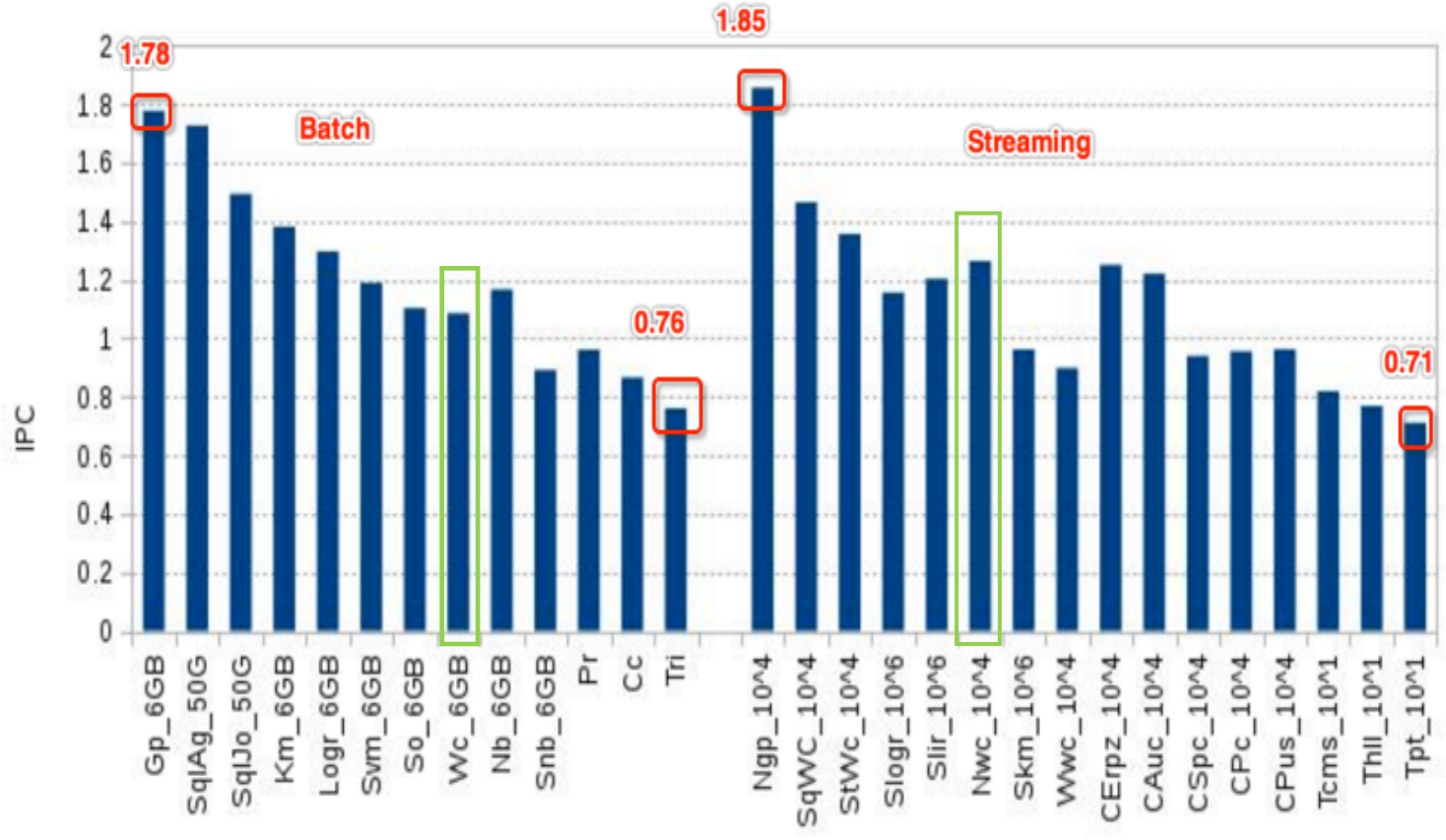
---Bad Speculation: a micro-operation is issued in a given cycle, it would eventually canceled

---Retiring: a micro-operation is issued in a given cycle, it would eventually canceled

TABLE VI: Metrics for Top-Down Analysis of Workloads

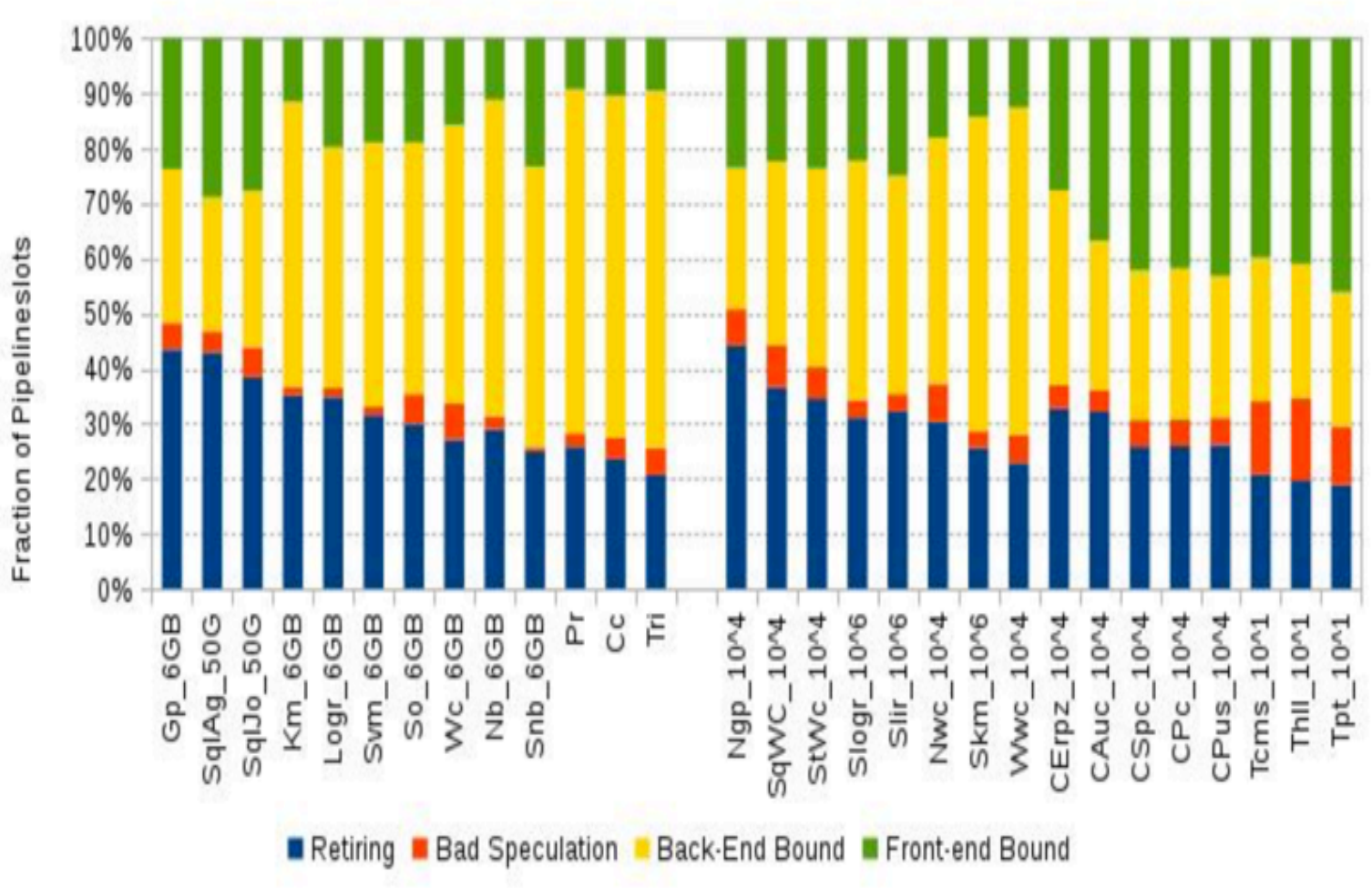
Metrics	**Description**
IPC	average number of retired instructions per clock cycle
DRAM Bound	how often CPU was stalled on the main memory
L1 Bound	how often machine was stalled without missing the L1 data cache
L2 Bound	how often machine was stalled on L2 cache
L3 Bound	how often CPU was stalled on 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.

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



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

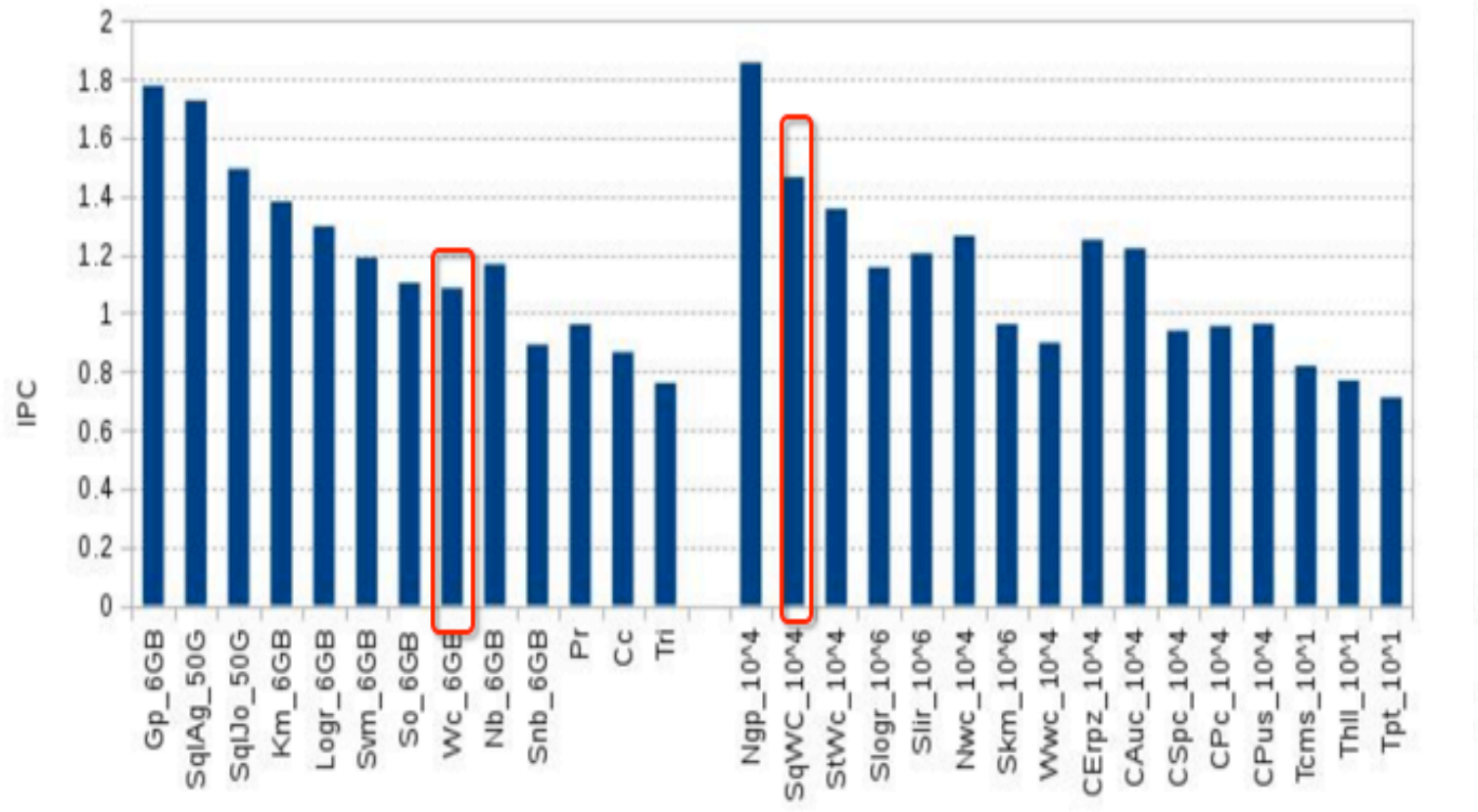
The IPC values of word count (Wc) and grep (Gp) are very close to their stream processing equivalents, i.e. network word count (NWc) and network grep (NGp).



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

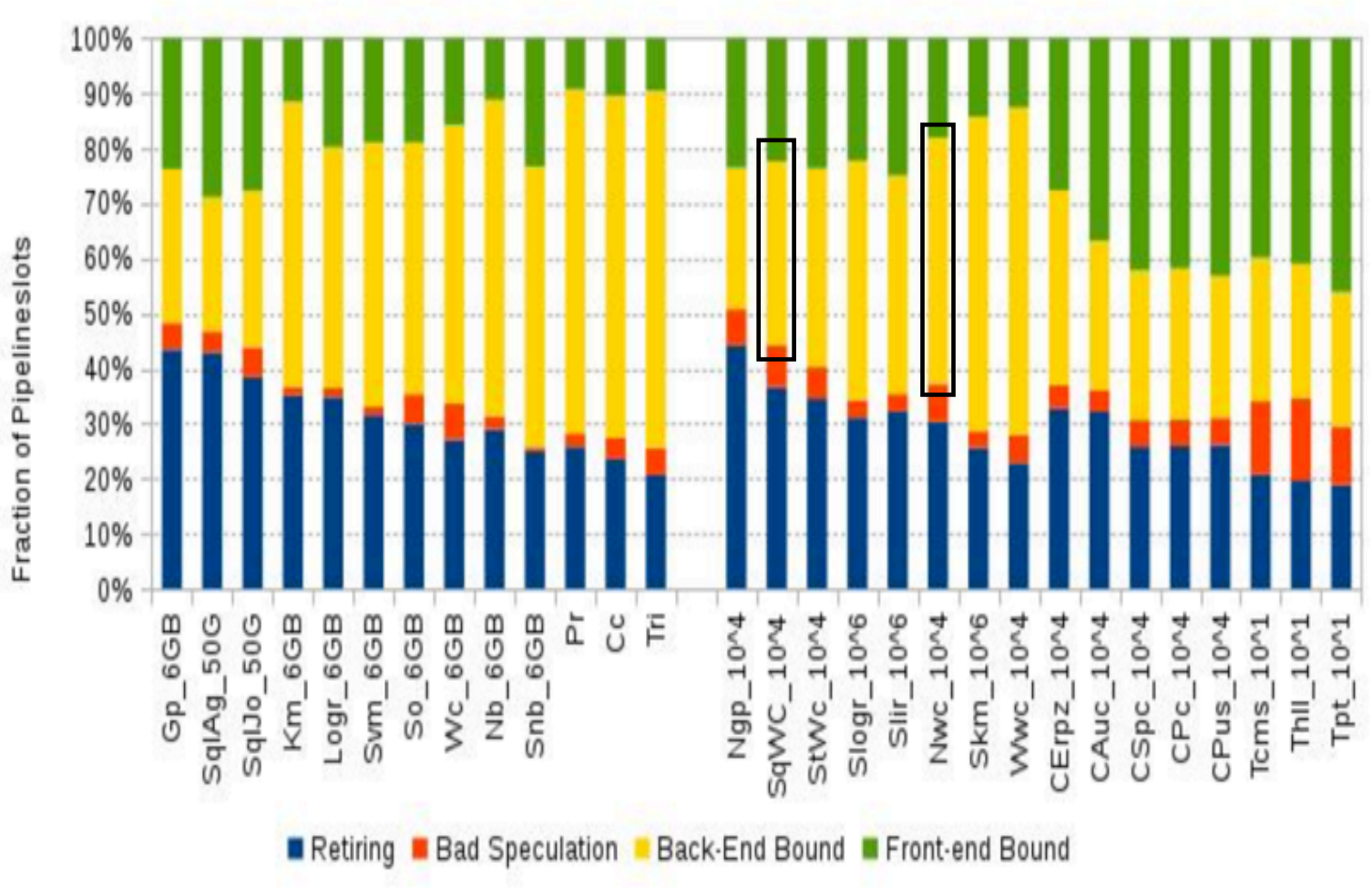
the pipeline slots breakdown in Figure b for the same workloads are quite similar

Conclusion of two graphs
implies that batch processing and stream processing will have same micro-architectural behavior 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

Sql Word Count(SqWc), which uses the Dataframes has better IPC than both word count (Wc) and network word count (NWc), which use RDDs



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

Sql Word Count (SqWc) exhibits 25.56% less back-end bound slots than streaming network word count (NWc)

Conclusion of two graphs
Dataframes have the potential to improve the micro-architectural performance of Spark workloads

Reason:
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

Evaluation

How does data velocity affect micro-architectural perfor- mance of in-memory data analytics with Spark?

Input data rate per second

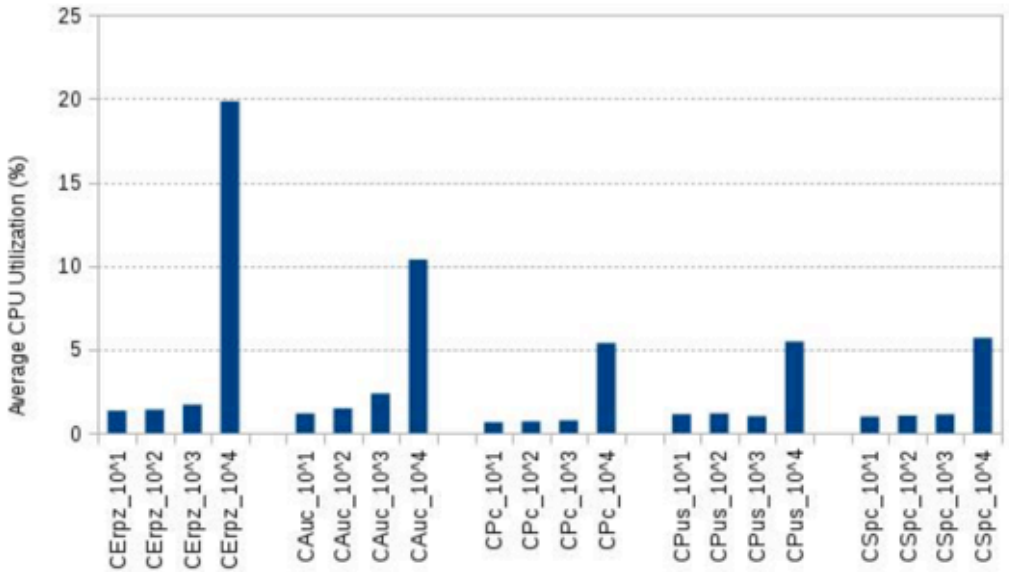
---10

---100

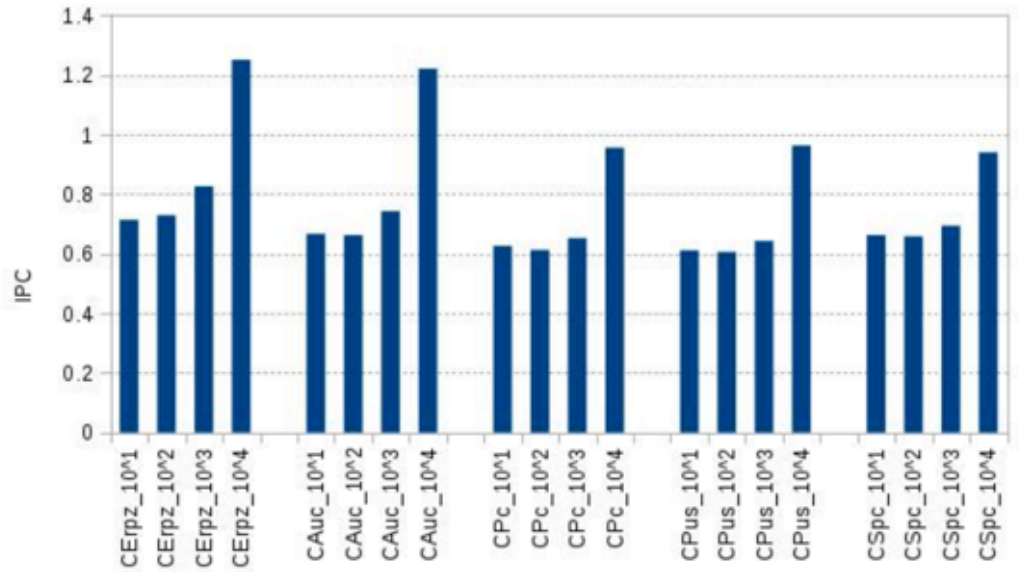
---1000

---10000

Evaluation



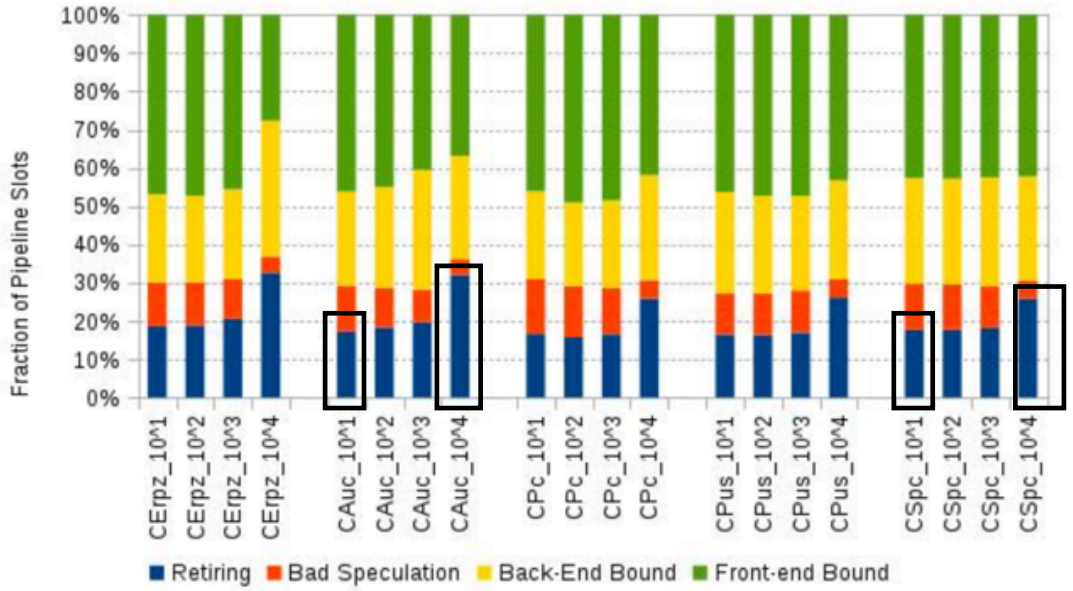
(a) CPU utilization increases with data velocity



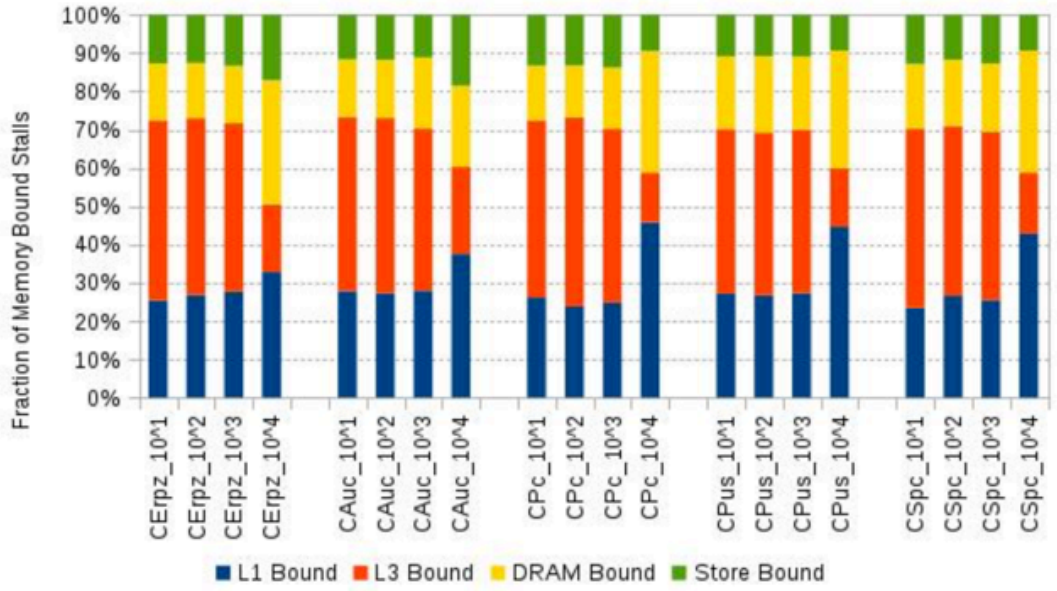
(b) Better IPC at higher data velocity

---CPU utilization increases only modestly up to 1000 events/s after which it increases up to 20%.

---IPC increases by 42% in CSpc and 83% in CAuc when input rate is increased from 10 to 10,000 events per second



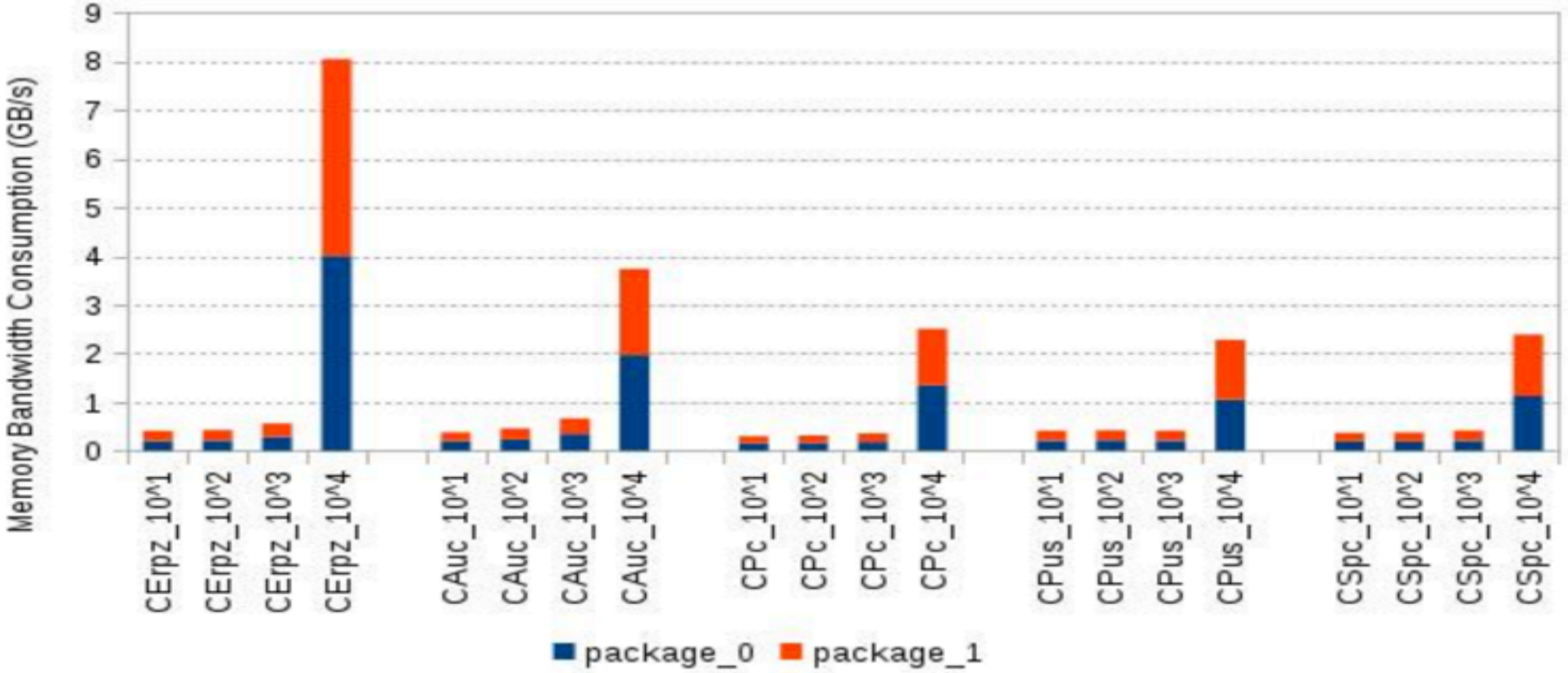
(c) Front-end bound stalls decrease and fraction of retiring slots increases with data velocity



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

---fraction of pipeline slots being retired increases by 14.9% in CAuc and 8.1% in CSpc

---L1 bound stalls increase, L3 bound stalls decrease and DRAM bound stalls increase at high data input rate



(f) Memory bandwidth consumption increases with data velocity

the memory bandwidth consumption is constant at 10, 100 and 1000 events/s and then increases significantly at 10,000 events/s.

Conclusion:

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

input data rates should be high enough to provide working sets large enough to keep the execution units busy

Conclusion

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

2. Spark workloads using DataFrames have improved instruction retirement over workloads using RDDs.

3. 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.