Architectural Impact on Performance of In-memory Data Analytics: Apache Spark Case Study

Ahsan Javed Awan*, Mats Brorsson*, Vladimir Vlassov* and Eduard Ayguade†
*KTH Royal Institute of Technology,
Software and Computer Systems Department(SCS),
{ajawan, matsbror, vladv}@kth.se
†Technical University of Catalunya (UPC),
Computer Architecture Department,
eduard@ac.upc.edu

***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 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 are stream processing workloads have similar micro-architectural characteristics are bounded by the latency of frequent data access to DRAM. For data accesses we have found that simultaneous multi-threading is effective in hiding the data latencies. We have also observed that (i) data locality on NUMA nodes can improve the performance by 10% on average and(ii) disabling next-line L1-D prefetchers can reduce the execution time by up-to 14% and (iii) multiple small executors can provide up-to 36% speedup over single large executor**

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 a majority of analytics jobs do not process really data sets and that 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 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 small cluster of scale-up servers to 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 large number 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 performance of in-memory data analytics with Spark on a scale-up server but they are limited in two ways. Firstly, they are limited only to batch processing workloads and secondly, they do not quantify the impact of NUMA, Hyper-Threading and hardware prefetchers on the performance of Spark workloads. Knowing the limitations of modern scale-up servers for in-memory data analytics with Spark will help in achieving the future goal of improving the performance of in-memory data analytics with Spark on small clusters of scale-up servers.

Our contributions are:

- We characterize the micro-architectural performance of Spak-core, Spark Mllib, Spark SQL, GraphX and Spark Streaming.
- We quantify the impact of data velocity on micro-architectural performance of Spark Streaming.
- We analyze the impact of data locality on NUMA nodes for Spark.
- We analyze the effectiveness of Hyper-threading and existing prefetchers in Ivy Bridge server to hide data access latencies for in-memory data analytics with Spark.
- We quantify the potential for high bandwidth memories to improve the the performance of in-memory data analytics with Spark.
- We make recommendations on the configuration of Ivy Bridge server and Spark to improve the performance of in-memory data analytics with Spark

The rest of this paper is organised as follows. Firstly, we provide background and formulate 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 the 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 is a machine learning library on top of Spark-core. It contains commonly used algorithms related to collaborative filtering, clustering, regression, classification and dimensionality reduction.

C. Graph X

GraphX 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 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 [8] 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 like 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

Dual socket servers are becoming commodity in the data centers, e.g Google warehouse scale computers have been optimized for NUMA machines [9]. Moder scale-up servers like Ivy Bridge machines are also part of Google data centers as recent study [10] presents a detailed micro-architectural analysis of Google data center jobs running on the Ivy Bridge servers. Realizing the need of the hour, the inventors of Spark are also developing cache friendly data structures and algorithms under the project name Tungsten [11] to improve the memory and CPU performance of Spark applications on modern 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 accesses 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.

Simulatenous multi-threading and hardware prefectching are effective ways to hide data access latencies and additional latency over-head due to accesses to remote memory can be removed by co-locating the computations with data they access on the same socket.

One reason for severe impact of garbage collection is that full generation garbage collections are triggered frequently at large volumes of input data and the size of JVM is directly related to Full GC time. Multiple smaller JVMs could be better than a single large JVM.

In this paper, we answer the following questions concerning in-memory data analytics running on modern scale-up servers using the 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 micro-architectural performance of in-memory data analytics with Spark?
- How much performance gain is achievable by co-locating the data and computations on NUMA nodes for in-memory data analytics with Spark?
- Is simultaneous multi-threading effective for in-memory data analytics with Spark?
- Are existing hardware prefetchers in modern scale-up servers effective for in-memory data analytics with Spark?

- Does in-memory data analytics with Spark experience loaded latencies (happens if bandwidth consumption is more than 80% of sustained bandwidth)
- Are multiple small executors (which are java processes in Spark that run computations and store data for the application) better than single large executor?

III. Methodology

Our study of architectural impact on in-memory data analytics is based on an imperial study of performance of batch and stream processing with Spark using representative benchmark workloads. We have performed several series of experiments, in which we have evaluated impact of each of the architectural features, such as data locality in NUMA, HW prefetchers, and hyper-threading, on in-memory data analytics with Spark.

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 chose the workloads carefully. Batch processing workloads are a subset BigdataBench [12] and HiBench [13] which are highly referenced benchmark suites in Big data domain. Stream processing workloads used in the paper are super set 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 (which is Spark's scalable machine learning library [16]) 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 was used to generate synthetic data sets based on raw data sets [17].

B. *System Configuration*

Table III shows details about our test machine. Hyper-threading is only enabled during the evaluation of simultaneous multi-threading for Spark workloads. Otherwise 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 [18]. With Hyper-Threading and Turbo-boost disabled, there are 24 cores in the system operating at the frequency of 2.7 GHz.

Table IV also 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 [19].

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 (Structured)
	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 (Structured)
	Gaussian (Gu)	uses Gaussian clustering algorithm from Spark Mllib. The benchmark is run for 10 iterations with 2 desired clusters	
	Sparse NaiveBayes (SNb)	uses NaiveBayes classification alogrithm from Spark Mllib	
	Support Vector Machines (Svm)	uses SVM classification alogrithm from Spark Mllib	
	Logistic Regression(Logr)	uses Logistic Regression alogrithm 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 aggregation query from BigdataBench using DataFrame API	

C. *Measurement Tools and Techniques*

We configure Spark to collect GC logs which are then parsed to measure time (called real time in GC logs) spent in garbage collection. We rely on the log files generated by Spark to calculate the execution time of the benchmarks. We use Intel Vtune Amplifier [20] to perform general micro-architecture exploration and to collect hardware performance counters. We use numactl [21] to control the process and memory allocation affinity to a particular socket. We use hwloc [22] to get the CPU ID of hardware threads. We use msr-tools [23] to read and write model specific registers (MSRs).

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 found 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 [24] to study the micro-architectural performance of the workloads. Earlier studies on profiling on big data workloads shows the efficacy of this method in identifying the micro-architectural bottlenecks [6], [10], [25]. 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

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)	Use 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.	

Bound, Bad Speculation and Retiring. If a micro-operation is issued in a given cycle, it would eventually either get retired or cancelled. 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 following the metrics described in Table V, whose definition are taken from Intel Vtune on-line help [20].

TABLE III: 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	

TABLE IV: 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		

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 behaviour. Figure 1a shows the IPC values of batch processing workloads range between 1.78 to 0.76, where as IPC values of stream processing workloads also range between 1.85 to 0.71. 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). 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 behaviour 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. Aggregration (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.

TABLE V: 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
DTLB Overhead	fraction of cycles spent on handling first-level data TLB load misses
Cycles of 0 ports Utilized	the number of cycles during which no port was utilized.

Sql Word Count (SqWc) exhibits 25.56% less back-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 fraction of clock cycles during which no ports are utilized, is 5.23% less than in network wordcount. 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 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). Fraction of cycles during which 0 port is utilized, however differ only by 2.94%. Icahce 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 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).

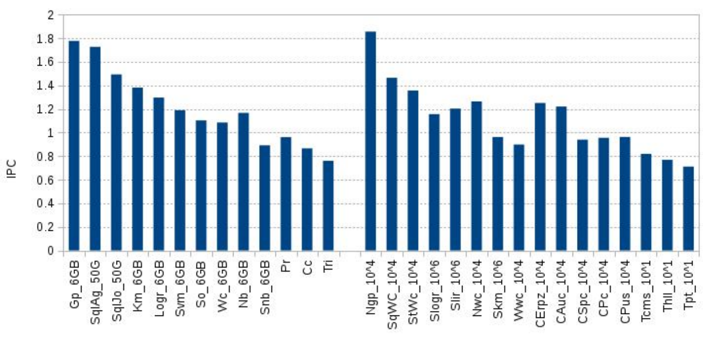
CErpz and CAuc both use "window", "map" and "groupbyKey" transformations but the front-end bound fraction and icache miss imapct in CAuc is larger than in CErpz. However, back-end bound fraction, DRAM bound stalled cycles, memory bandwidth consumption are larger in CErpz than in CAuC. The retiring fraction is almost same in both workloads. The difference is again the working set. The working set in CErpz with window length of 30 seconds is 15 x 10,000 events which is 3x larger than in CAuc with 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?

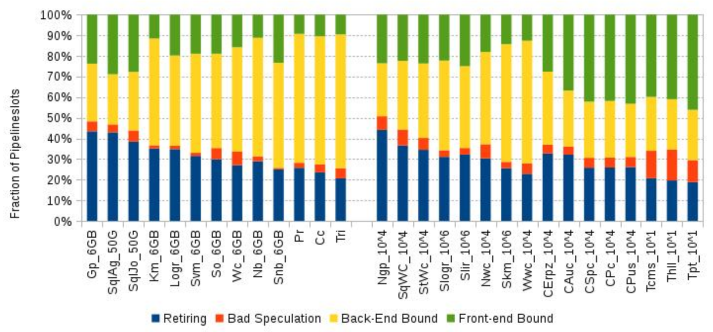
In order to answer the question, we compare the micro-architectural characterization of stream processing workloads at input data rates of 10, 100, 1000 and 10,000 events per second. Figure 2a shows that CPU utilization increases only modestly up-to 1000 events/s after which it increases up-to 20%. Like wise 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%, where as 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%.

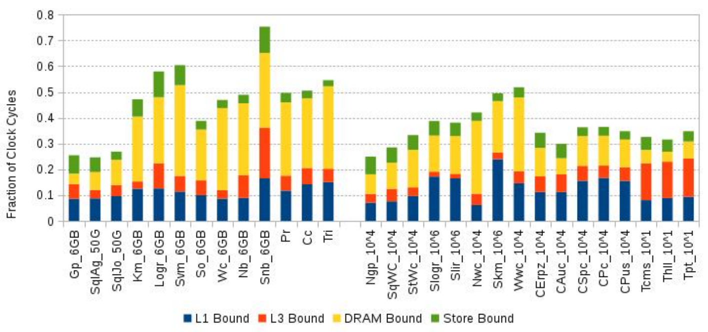
The memory subsystem stalls break down in Figure 2d show that L1 bound stalls increase, L3 bound stalls decrease and DRAM bound stalls increase at high data input rate, e.g in CErpz, L3 bound stall and DRAM bound stalls remain roughly constant at 10, 100 and 1000 events/s because the working sets are 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



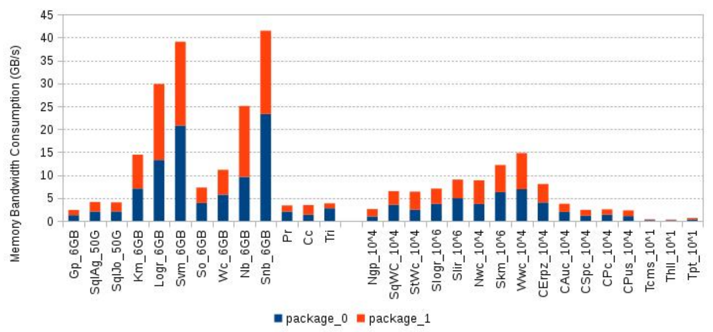
(a) IPC values of stream processing workloads lie in the same range as of batch processing workloads



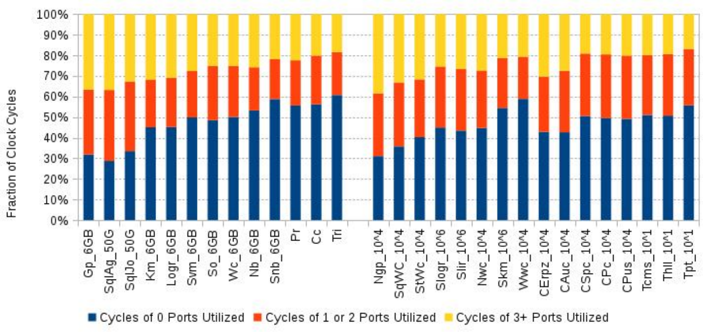
(b) Majority of stream processing workloads are back-end bound as that of batch processing workloads



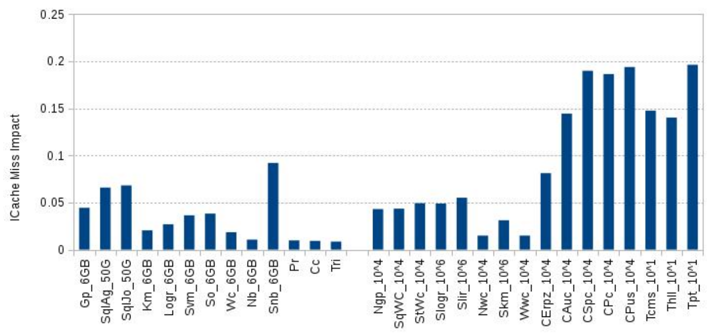
(c) Stream processing workloads are also DRAM bound but their fraction of DRAM bound stalled cycles is lower than that of batch processing workloads



(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

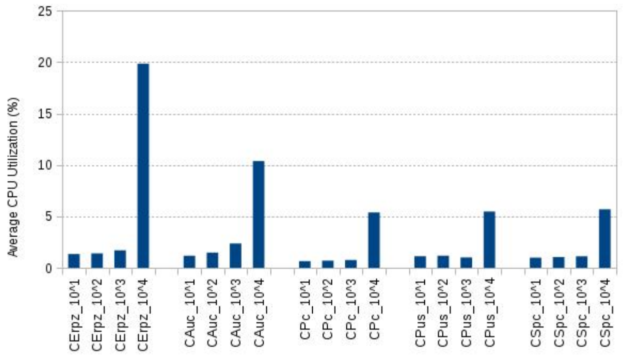
Fig. 1: Comparison of micro-architectural characteristics of batch and stream processing workloads

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.

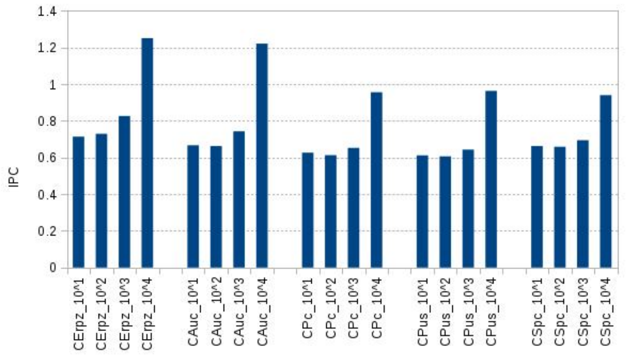
C. Does data locality on NUMA nodes improve the performance of in-memory data analytics with Spark?

Ivy Bridge Server is a NUMA multi-socket system. Each socket has 2 on-chip memory controllers and a part of the main memory is directly connected to each socket. This layout offers high bandwidth and low access latency to the directly connected part of the main memory. The sockets are connected by two QPI (Quick Path Interconnect) links, thus a socket can access the main memory of other socket. However, a memory access from one socket to memory from another socket (remote memory access) incurs additional latency overhead due to transferring the data by cross-chip interconnect. By co-locating the computations with the data they access, the NUMA overhead can be avoided.

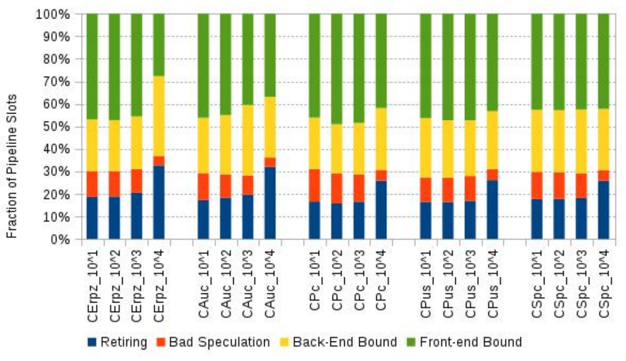
To evaluate the impact of NUMA on Spark workloads, we run the benchmarks in two configurations: a) Local DRAM, where Spark process is bound to socket 0 and memory node 0, i.e. computations and data accesses are co-located, and b) Remote DRAM, where spark process is bound to socket 0 and memory node 1, i.e. all data accesses incur the additional



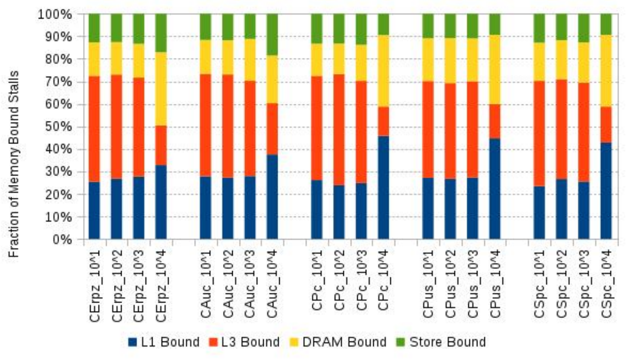
(a) CPU utilization increases with data velocity



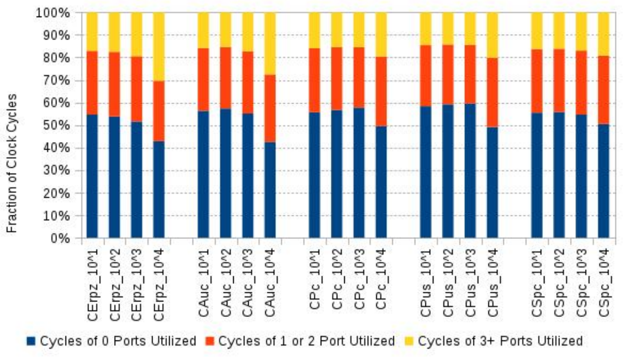
(b) Better IPC at higher data velocity



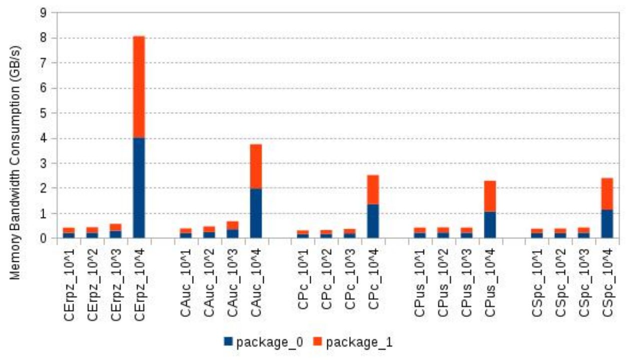
(c) Front-end bound stalls decrease and fraction of retiring slots increases with data velocity



(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

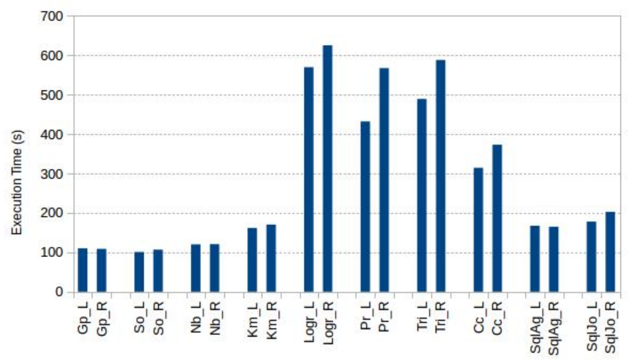
Fig. 2: Impact of Data Velocity on Micro-architectural Performance of Spark Streaming Workloads

latency. The input data size for the workloads is chosen as 6GB to ensure that memory working set sizes fit socket memory. Spark parameters for the two configurations are given in Table VI. Equation 1 and 2 in Appendix give the formulae for fraction of clock cycles, CPU stalled on local DRAM and remote DRAM respectively.

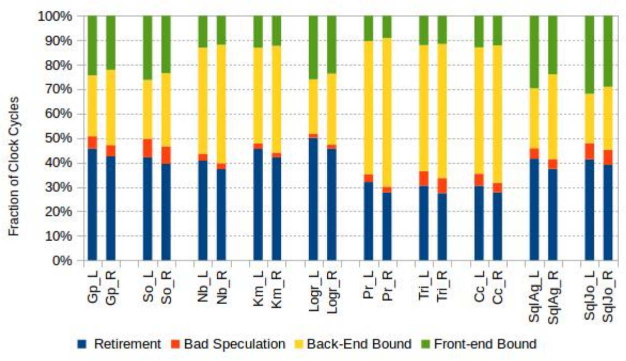
Figure 3a shows remote memory accesses can degrade the performance of Spark workloads by 10% on average. This is because despite the stalled cycles on remote memory accesses double (see Figure 3c), retiring category degrades by only 8.7%, Back-end bound stalls increases by 19.45%, bad speculation decreases by 9.1% and front-end bound stalls decreases by 9.58% on average as shown in Figure 3b. Furthermore the total cross-chip bandwidth of 32 GB/sec (peak bandwidth of 16 GB/s per QPI link) satisfies the memory bandwidth requirements of Spark workloads (see Figure 3d).

D. Is simultaneous multi-threading effective for in-memory data analytics?

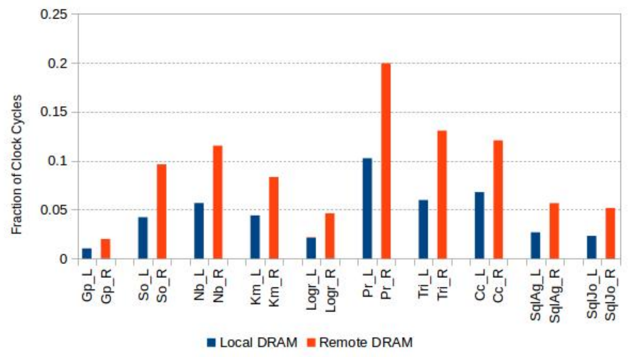
Ivy Bridge Machine uses Simultaneous Multi-threading(SMT), which enables one processor core to run two software threads simultaneously to hide data access latencies. To evaluate the effectiveness of Hyper-Threading, we run Spark process in the three different configurations a) ST:2x1, the base-line single threaded configuration where Spark process is bound to two physical cores b)



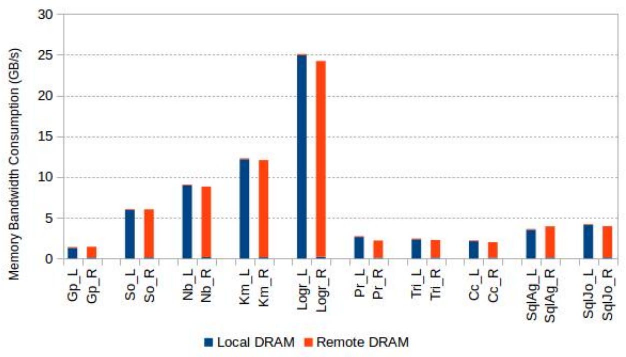
(a) Performance degradation due to NUMA is 10% on average across the workloads.



(b) Retiring decreases due to increased back-end bound in remote only mode.



(c) Stalled Cycles double in remote memory case



(d) Memory Bandwidth consumption is well under the limits of QPI bandwidth

Fig. 3: NUMA Characterization of Spark Benchmarks

TABLE VI: Machine and Spark Configuration for NUMA Evaluation

		Local DRAM	**Remote DRAM**
Hardware	Socket ID	0	0
	Memory Node ID	0	1
	No. of cores	12	12
	No. of threads	12	12
Spark	spark.driver.cores	12	12
	spark.default.parallelism	12	12
	spark.driver.memory (GB)	24	24

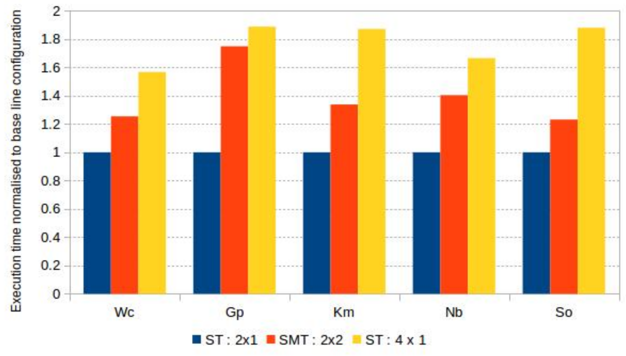
TABLE VII: Machine and Spark Configurations to evaluate Hyper Threading

		ST:2x1	**SMT:2x2**	**ST:4x1**
Hardware	No of sockets	1	1	1
	No of memory nodes	1	1	1
	No. of cores	2	2	4
	No. of threads	1	2	1
Spark	spark.driver.cores	2	4	4
	spark.default.parallelism	2	4	4
	spark.driver.memory (GB)	24	24	24

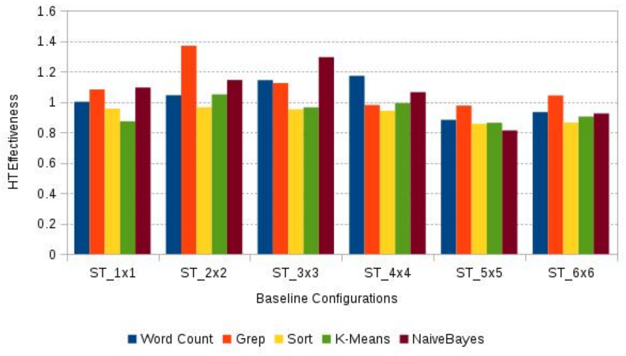
SMT:2x2, a simultaneous multi-threaded configuration where Spark process is allowed to use 2 physical cores and their corresponding hyper threads and c) ST:4x1, the upper-bound single threaded configuration where Spark process is allowed to use 4 physical cores. Spark parameters for the aforementioned configurations are given in Table VII. We also experimented with base-line configurations, ST:1x1, ST:3x3, ST:4x4, ST:5x5 and ST:6x6. In all experiments socket 0 and memory node 0 is used to avoid NUMA affects and the size of input data for the workloads is 6GB

Figure 4a shows that SMT provides 39.5% speedup on average across the workloads over baseline configuration, while the upper-bound configuration provided 77.45% on average across the workloads. The memory bandwidth in SMT case also keeps up with multi-core case it is 20.54% less than that of multi-core version on average across the workloads 4c. Figure 4b presents HT Effectiveness at different baseline configurations. HT Effectiveness of 1 is desirable as it implies 30% performance improvement in Hyper-Threading case over the baseline single threaded configuration [26]. Equation 3 in Appendix gives the formula for HT effectiveness. One can see HT effectiveness remains close to 1 on average across the workloads till 4 cores after that it drops. This is because of poor multi-core scalability of Spark workloads as shown in [6]

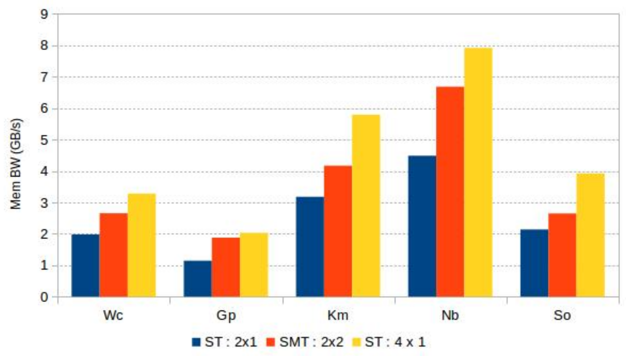
For most of the workloads, DRAM bound is reduced to half whereas L1 Bound doubles when comparing the SMT case over baseline ST case in Figure 4d implying that Hyper-threading is effective in hiding the memory access latency for Spark workloads



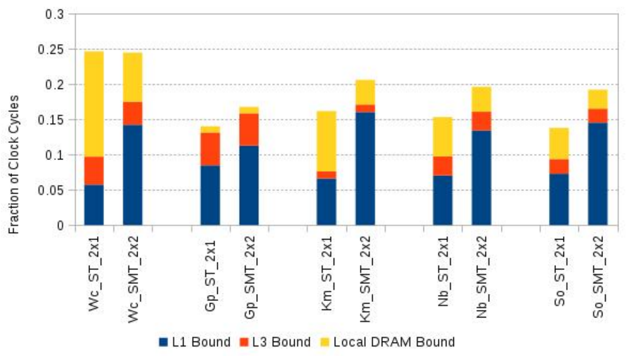
(a) Multi-core vs Hyper-Threading



(b) HT Effectiveness is around 1



(c) Memory Bandwidth in multi-threaded case keeps up with that in multi-core case.



(d) DRAM Bound decreases and L1 Bound increases

Fig. 4: Hyper Threading is Effective

E. Are existing hardware prefetchers in modern scale-up servers effective for in-memory data analytics?

Prefetching is a promising approach to hide memory access latency by predicting the future memory accesses and fetching the corresponding memory blocks into the cache ahead of explicit accesses by the processor. Intel Ivy Bridge Server has two L1-D prefetchers and two L2 prefetchers.The description about prefetchers is given in Table VIII. This information is taken from Intel software forum [27].

TABLE VIII: Hardware Prefetchers Description

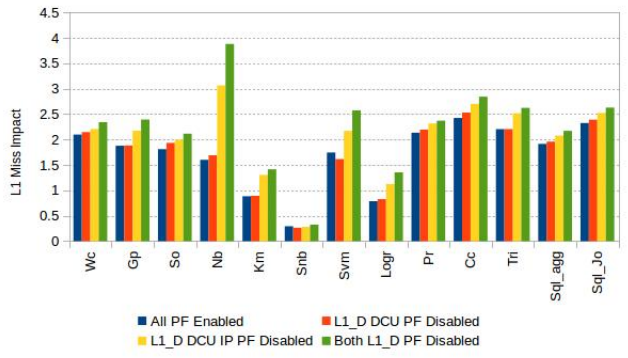
Prefetcher	**Bit No. in MSR (0x1A4)**	**Description**
L2 hardware prefetcher	0	Fetches additional lines of code or data into the L2 cache
L2 adjacent cache line prefetcher	1	Fetches the cache line thatcomprises a cache line pair(128 bytes)
DCU prefetcher	2	Fetches the next cache line into L1-D cache
DCU IP prefetcher	3	Uses sequential load history (based on Instruction Pointer of previous loads) to determine whether to prefetch additional lines

To evaluate the effectiveness of L1-D prefetchers, we measure L1-D miss impact for the benchmarks at four configurations: a) all processor prefetchers are enabled, b) DCU prefetcher is disabled only, c) DCU IP prefetcher is disabled only and d) both L1-D prefetchers are disabled. To assess the effectiveness of L2 prefetchers, we measure L2 miss rate for the benchmarks at four configurations: a) all processor prefetchers are enabled, b) L2 hardware prefetcher is disabled only, c) L2 adjacent cache line prefetcher is disabled only and d) both L2 prefetchers are disabled. Equations 4 and 5 in the Appendix give formulae for L1-D miss impact and L2 hit impact.

Figure 5a shows that L1-D miss impact increases by only 3.17% on average across the workloads when DCU prefetcher disabled, whereas the same metric increases by 34.13% when DCU IP prefetcher is disabled in comparison with the case when all processor prefetchers are enabled. It implies that DCU prefetcher is ineffective.

Figure 5b shows that L2 hit impact decreases by 18% on average across the workloads, when L2 adjacent cache line prefetcher disabled, whereas disabling L2 adjacent line prefetcher decreases the L2 hit imapct by only 1.36% on average across the workloads. This implies that L2 adjacent cache line prefetcher is ineffective. .

By looking at the percentage change in execution time of Spark workloads over baseline configuration (all prefetchers are enabled), One can see that L1-D next-line and adjacent


(a) L1-D DCU Prefetcher is ineffective

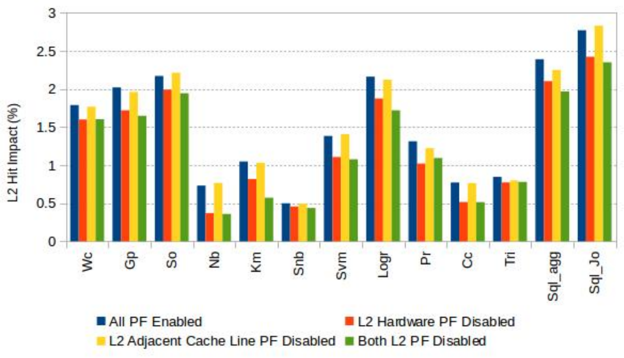

(b) Adjacent Cache Line L2 Prefecher is ineffective

Fig. 5: Evaluation of Hardware Prefetchers

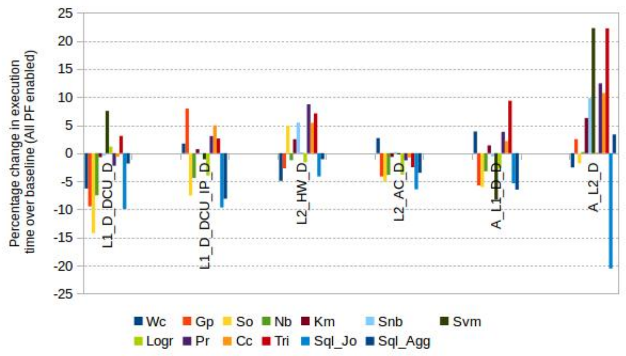


Fig. 6: Disabling L1-D next-line and L2 Adjacent Cache Line Prefetchers can reduce the execution of Spark jobs up-to 15% and 5% respectively

cache line L2 prefetchers have a negative impact on Spark workloads and disabling them improves the performance of Spark workloads up to 14.2% and 4.13%. This shows that simple next-line hardware prefetchers in modern scale-up servers are ineffective for in-memory data analytics.

F. Does in-memory data analytics with Spark experience loaded latencies (happens if bandwidth consumption is more than 80% of sustained bandwidth)

According to Jacob et al. in [28], the bandwidth vs latency response curve for a system has three regions. For the first 40% of the sustained bandwidth, the latency response is nearly constant. The average memory latency equals idle latency in the system and the system performance is not limited by the memory bandwidth in the constant region. In between 40% to 80% of the sustained bandwidth, the average memory latency increases almost linearly due to contention overhead by numerous memory requests. The performance degradation of the system starts in this linear region. Between 80% to 100% of the sustained bandwidth, the memory latency can increase exponentially over the idle latency of DRAM system and the applications performance is limited by available memory bandwidth in this exponential region. Note that maximum sustained bandwidth is 65% to 75% of the theoretical maximum for server workloads.

Using the formula 6, taken from Intel's document [18], we calculate that maximum theoretical bandwidth, per socket, for processor with DDR3-1866 and 4 channels is 59.7GB/s and the total system bandwidth is 119.4 GB/s. To find sustained maximum bandwidth, we compile the ompenmp version of STREAM [29] using Intel's icc compiler. The compiler flags used are given in the Appendix. On running the benchmark, we find maximum sustained bandwidth to be 92 GB/s.

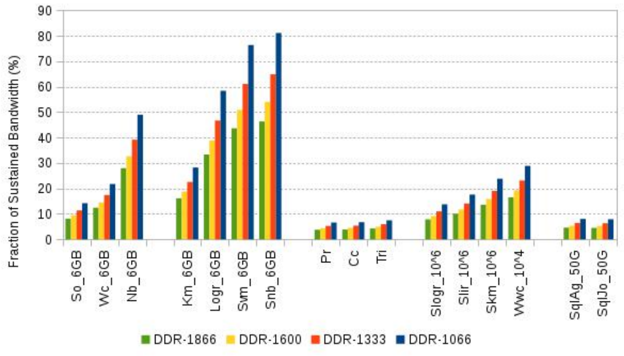


Fig. 7: Spark workloads dont experience loaded latencies

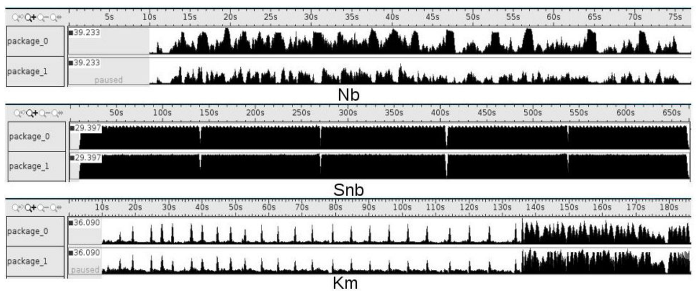


Fig. 8: Bandwidth Consumption over time

Figure 7 shows the average bandwidth consumption as a fraction of sustained maximum bandwidth for different BIOS configurable data transfer rates of DDR3 memory. It can be seen that Spark workloads consume less than 40% of sustained maximum bandwidth at 1866 data transfer rate and thus operate in the constant region. By lowering down the data transfer rates to 1066, majority of workloads from Spark core, all workloads from Spark SQL, Spark Streaming and Graph X still operate on the boundary of linear region where as workloads from Spark MLlib shift to the linear region and mostly operate at the boundary of linear and exponential region. However at 1333, Spark MLlib workloads operate roughly in the middle of linear region. From the bandwidth consumption over time curves of the Km, Snb and Nb in Figure 8,it can be seen that even when the peak bandwidth utilization goes into the exponential region, it lasts only for a short period of time and thus have negligible impact on the performance.

It implies that Spark workloads do not experience loaded latencies and by lowering down the data transfer rate to 1333, performance is not affected. However, DRAM power consumption will be reduced as it is proportional to the frequency of DRAM.

G. Are multiple small executors better than single large executor?

With increase in the number of executors, the heap size of each executor's JVM is decreased. Heap size smaller than 32 GB enables "CompressedOops", that results in fewer garbage collection pauses. On the other-hand, multiple executors may need to communicate with each other and also with the driver. This leads to increase in the communication overhead. We study the trade-off between GC time and communication overhead for Spark applications.

We deploy Spark in standalone mode on a single machine, i.e. master and worker daemons run on the same machine. We run applications with 1, 2, 4 and 6 executors. Beyond 6, we hit the operating system limit of maximum number of threads in the system. Table 1 lists downs the configuration details, e.g in 1E case, one Java Virtual Machine of 50 GB Heap size is launched and executor pool uses 24 threads, where as in 2E case 2 Java Virtual machines are launched, each with 25 GB of Heap space and 12 threads in the executor pool. In all configurations, the total number of cores and the total memory used by the applications are constant at 24 cores and 50GB respectively.

TABLE IX: Multiple Executors Configuration

Configuration	**1E**	**2E**	**4E**	**6E**
spark.executor.instances	1	2	4	6
spark.executor.memory (GB)	50	25	12.5	8.33
spark.executor.cores	24	12	6	4
spark.driver.cores	1	1	1	1
spark.driver.memory (GB)	5	5	5	5

Figure 9 data shows that 2 executors configuration are better than 1 executor configuration, e.g. for K-Means and Gaussian, 2E configuration provides 29.31% and 30.43% performance improvement over the baseline 1E configuration, however 6E configuration only increases the performance gain to 36.48% and 35.47% respectively. For the same workloads, GC time in 6E case is 4.08x and and 4.60x less than 1E case. A small performance gain from 2E to 6E despite the reduction in GC time can be attributed to increased communication overhead among the executors and master

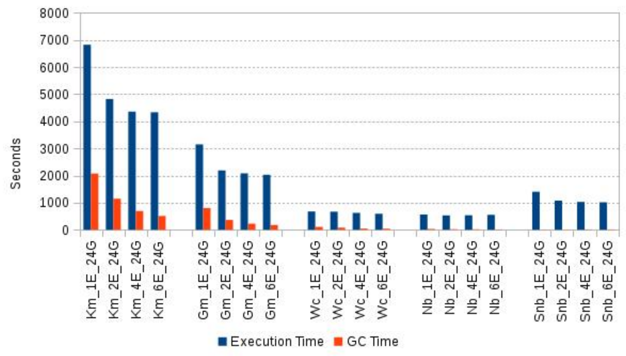


Fig. 9: Multiple small executors are better than single large executor due to reduction in GC time

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], [25], [30]–[34]. However these studies lack in identifying the limitations of modern scale-up servers for Spark based data analytics. Ferdman et al. [30] 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. [35] 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 [31]. They also suffer from notable from 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 [25] reveal that big data analysis workload is bound on memory latency but the conclusion can not be generalised. None of the above mentioned works consider frameworks that enable in-memory computing of data analysis workloads.

Tang et al. [9] have shown that NUMA has significant impact on Gmail backend and web.search frontend. Researchers at IBM's Spark technology center [36] has only explored the thread affinity, bind only JVMs to sockets but does not limit the cross socket accesses. Beamer et al. [37] have shown NUMA has moderate performance penalty and SMT has limited potential for graph analytics running on Ivy bridge server. We show that exploiting the data locality on the modern servers will not yield significant performance gain for Spark and give micro-architectural reasons why this is so.

Kanev et al. [10] have argued in favour of SMT after profiling live data center jobs on 20,000 google machines. While SMT has been shown to be effective for Hadoop workloads [25], the same conclusion could not be translated about Spark workloads as previous work shows that as memory access characteristics of Spark and Hadoop differ [33] and software stacks have significant impact on the micro-architecture behaviour of big data workloads [33]. By reaching the same conclusion for Spark, we consolidate the general understanding of effectiveness of SMT for Big Data workloads

The general understanding about current Intel prefetchers is that they have either neutral or positive impact on SPEC benchmarks and Cloudsuite [30]. We show for the first time the they have negative impact on the performance of in-memory data analytics with Spark.

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 behaviour in Spark if the difference between two implementations is of micro-batching only.
- 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.
- Exploiting data locality on NUMA nodes can only reduce the job completion time by 10% on average as it reduces the back-end bound stalls by 19%, which improves the instruction retirement only by 9%.
- Hyper-Threading is effective to reduce DRAM bound stalls by 50%, HT effectiveness is 1.
- Disabling next-line L1-D and Adjacent Cache line L2 prefetchers can improve the performance by up-to 14% and 4% respectively.
- Spark workloads does not experience loaded latencies and it is better to lower down the DDR3 speed from 1866 to 1333.
- Multiple small executors can provide up-to 36% speedup over single large executor.

Firstly, we recommend Spark users to prefer DataFrames over RDDs while developing Spark applications and input data rates should be large enough for real time streaming analytics to improve the instruction retirement. Secondly, We advise to use executors with memory size less than or equal to 32GB and restrict each executor to use NUMA local memory. Thirdly we recommend to enable hyper-threading, disable next-line L1-D and adjacent cache line L2 prefetchers and lower the DDR3 speed to 1333.

We also envision processors with 6 hyper-threaded cores without L1-D next line and adjacent cache line L2 prefetchers. The die area saved can be used to increase the LLC capacity. and the use of high bandwidth memories like Hybrid memory cubes is not justified for in-memory data analytics with Spark.

VII. Appendix

Here we give the formulas for metrics used in the evaluation of NUMA, SMT and hardware prefetchers in our study.

Equation 1 gives the formula for $LocalDRAMBound$, which tells how often the CPU was stalled on local memory node. It is calculated by multiplying the number of retired load micro-operations, which data sources missed LLC but serviced from local dram with the corresponding latency cycles and then dividing by the total number of clock cycles when the cores are not in halted state.

$$Local\ DRAM\ Bound = (130 * MEM\_LOAD\_UOPS\_LLC\_MISS\_RETIRED.LOCAL\_DRAM) / CPU\_CLK\_UNHALTED.THREAD \tag{1}$$

Equation 2 gives the formula for $RemoteDRAMBound$, which tells how often the CPU was stalled on remote memory node. It is calculated by multiplying the number of retired load micro-operations, which data sources missed LLC but serviced from remote dram with the corresponding latency cycles and then dividing by the total number of clock cycles when the cores are not in halted state.

$$Remote\ DRAM\ Bound = (310 * MEM\_LOAD\_UOPS\_LLC\_MISS\_RETIRED.REMOTE\_DRAM) / CPU\_CLK\_UNHALTED.THREAD \tag{2}$$

Equation 3 gives the formula for $HT\ Effectiveness$, which is taken from Intel's on-line forum [26]. $HT\ Scaling_{obs}$ is the speedup observed in simultaneous multi-threaded case over the baseline single-threaded case, whereas $DP\ Scaling_{obs}$ speedup observed in the upper-bound single-threaded case.

$$HT\ Effectiveness = HT\ Scaling_{obs} * (0.538 + 0.462 / DP\ Scaling_{obs})) \tag{3}$$

Equation 4 gives the formula for L1 miss impact, which is obtained by multiplying the number of retired load micro-operations which data sources following L1 data-cache miss with the corresponding latency cycles and dividing product by the total number of clock cycles when the cores are not in halted state.

$$L1\ Miss\ Impact = (6 * MEM\_LOAD\_UOPS\_RETIRED.L1\_MISS\_PS) / CPU\_CLK\_UNHALTED.THREAD \tag{4}$$

Equation 5 gives the formula for L2 hit impact, which is obtained by multiplying the number of retired load micro-operations with L2 cache hits as data sources, with the corresponding latency cycles and dividing product by the total number of clock cycles when the cores are not in halted state.

$$L2\ Hit\ Impact = (12 * MEM\_LOAD\_UOPS\_RETIRED.L2\_HIT\_PS) / CPU\_CLK\_UNHALTED.THREAD \tag{5}$$

$$Maximum\ Theoretical\ Bandwidth\ per\ socket\ (GB/s) = (< MT/s > *8\ Bytes/clock* < num\ channels >)/1000 \tag{6}$$

$$icc\ -O3\ -openmp\ -DSTREAM\_ARRAY\_SIZE = 64000000\ -opt-prefetch-distance = 64, 8\ -opt-streaming-cache-evict = 0\ -opt-streaming-stores\ always\ stream.c\ -ostream\_omp.64M\_icc \tag{7}$$

Acknowledgments

This work was 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 was also supported by projects, Severo Ochoa SEV2015-0493 and TIN2015-65316 from the Spanish Ministry of Science and Innovation. We thank Ananya Muddukrishna for their 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 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] 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.

[9] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune, "Optimizing google's warehouse scale computers: The numa experience," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 188–197.

[10] 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.

[11] Project tungsten. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[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] 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.

[17] 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.

[18] 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.

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

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

[21] Numactl. http://linux.die.net/man/8/numactl.

[22] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in hpc applications," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE, 2010, pp. 180–186.

[23] msr-tools. https://01.org/msr-tools.

[24] 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.

[25] 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.

[26] HT Effectiveness. https://software.intel.com/en-us/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-with-an-application.

[27] Hardware Prefetcher Control on Intel Processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.

[28] B. Jacob, "The memory system: you can't avoid it, you can't ignore it, you can't fake it," *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, pp. 1–77, 2009.

[29] STREAM. https://www.cs.virginia.edu/stream/.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] Spark executors love numa process affinity. http://www.spark.tc/spark-executors-love-numa-process-affinity/.

[37] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 56–65.