Identifying the potential of Near Data Processing for Apache Spark

Ahsan Javed Awan
ajawan@kth.se

Eduard Ayguade
eduard.ayguade@bsc.es

Mats Brorsson
matsbror@kth.se

Moriyoshi Ohara
ohara@jp.ibm.com

Kazuaki Ishizaki
ishizaki@jp.ibm.com

Vladimir Vlassov
vladv@kth.se

KTH Royal Institute of Technology, Sweden Barcelona Super Computing Center, Spain

IBM Research Tokyo, Japan

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. There is also a renewed interest in Near Data Processing (NDP) due to technological advancement in the last decade. However, it is not known if NDP architectures can improve the performance of big data processing frameworks such as Apache Spark. In this paper, we build the case of NDP architecture comprising programmable logic based hybrid 2D integrated processing-in-memory and in-storage processing for Apache Spark, by extensive profiling of Apache Spark based workloads on Ivy Bridge Server.

Keywords

Processing in Memory, In-Storage Processing, Apache Spark

1. 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. While cluster computing frameworks are continuously evolving to provide real-time data analysis capabilities, Apache Spark [37] has managed to be at the forefront of big data analytics for being a unified framework for SQL queries, machine learning algorithms, graph analysis and stream data processing. Recent studies on characterizing in-memory data analytics with Spark show that (i) in-memory data analytics are bound by the latency of frequent data accesses to DRAM [7] and (ii) their performance deteriorates severely as we enlarge the input data size due to significant wait time on I/O [8].

The concept of near-data processing (NDP) is regaining the attention of researchers partially because of technological advancement and partially because moving the compute closer to the data where it resides, can remove the performance bottlenecks due to data movement. The umbrella of NDP covers 2D-integrated Processing-In-Memory, 3D-stacked Processing-In-Memory (PIM) and In-Storage Processing (ISP). Existing studies show efficacy of processing-in-memory (PIM) approach for simple map-reduce applications [16, 28], graph analytics [6, 25], machine learning applications [11, 20] and SQL queries [24, 34]. Researchers also show the potential of processing in non-volatile memories for I/O bound big data applications [12, 30, 33]. However, it is not clear which aspect of NDP (high bandwidth, improved latency, reduction in data movement, etc..) will benefit state-of-art big data frameworks like Apache Spark. Before quantifying the performance gain achievable by NDP for Spark, it is pertinent to answer which form of NDP (PIM, ISP) would better suit Spark workloads?

To answer this, we characterize Apache Spark workloads into compute bound, memory bound and I/O bound. We use hardware performance counters to identify the memory bound applications and OS level metrics like CPU utilization, idle time and wait time on I/O to filter out the I/O bound applications in Apache Spark and position ourselves as under

- ISP matches well with the characteristics of non iterative batch processing workloads in Apache Spark.
- PIM suits stream processing and iterative batch processing workloads in Apache Spark.
- Machine Learning workloads in Apache Spark are phasic and require hybrid ISP and PIM.
- 3D-Stacked PIM is an overkill for Apache Spark and programmable logic based hybrid ISP and 2D integrated PIM can satisfy the varying compute demands of Apache Spark based workloads.

2. BACKGROUND AND RELATED WORK

2.1 Spark

Spark is a cluster computing framework that uses Resilient Distributed Datasets (RDDs), 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 pipe-lined transformations. 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.

Spark MLlib is a scalable machine learning library [22] on top of Spark Core. GraphX enables graph-parallel computation in Spark. Spark SQL is a Spark module for structured data processing with data schema information. This schema information is used to perform extra optimization. Spark Streaming 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 Apache Kafka [19]. It then divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

2.2 Near Data Processing

The umbrella of near-data processing covers both processing in memory and in-storage processing. A survey [31] highlights historical achievements in technology that enables Processing-In-Memory (PIM) and various PIM architectures. It depicts PIM's advantages and challenges. Challenges of PIM architecture design are the cost-effective integration of logic and memory, unconventional programming models and lack of inter-operability with caches and virtual memory.

PIM approach can reduce the latency and energy consumption associated with moving data back-and-forth through the cache and memory hierarchy, as well as greatly increase memory bandwidth by sidestepping the conventional memory-package pin-count limitations. There exists a continuum of processing that can be embedded "in memory" [21]. This includes i) software transparent applications of logic in memory, ii) fixed function accelerators, iii) bounded operand PIM operations, which can be specified in a manner that is consistent with existing instruction-level memory operand formats, directly encoded in the opcode in the instruction set architecture, iv) compound PIM operations, which may access an arbitrary number of memory locations and perform number of different operations and v) fully programmable logic in memory, either a processor or re-configurable logic device.

2.3 Related work for NDP

2.3.1 Applications of PIM

PIM for Map-Reduce: For Map-Reduce applications, prior studies [16, 27] propose simple processing cores in the logic layer of 3D-stacked memory devices to perform Map operations with efficient data access and without hitting the memory bandwidth wall. The reduce operations despite having random memory access patterns are performed on the central host processor.

PIM for Graph Analytics: The performance of graph analytics is bound by the inability of conventional processing systems to fully utilize the memory bandwidth and Ahn et al. [6] propose in-order cores with graph processing specific prefetchers in the logic layer of 3D-stacked DRAM to fully utilize the memory bandwidth. Graph traversals are bounded by irregular memory access patterns of graph property and a study [25] proposes to offload the graph property to hybrid memory cube [1] (HMC) by utilizing the atomic requests described in HMC 2.0 specification (that is limited to only integer operations and one memory operand).

PIM for Machine Learning: Lee et al. [20] use State Synchronous Parallel (SSP) model to evaluate asynchronous parallel machine learning workloads and observe that atomic operations are the hotspots and propose to offload them onto logic layers in 3D stacked memories. These atomic operations are overlapped with main computation to increase the execution efficiency. K-means, a popular machine learning algorithm, is shown to benefit from higher bandwidth achieved by physically bonding the memory to the package containing processing elements [11]. Another proposal [13] is to use content addressable memories with hamming distance units in the logic layer to minimize the impact of significant data movement in k-nearest neighbours.

PIM for SQL queries: Researchers also exploit PIM for SQL queries. The motivation for pushing select query down to memory is reduce data movement by pushing only relevant data up the memory hierarchy [34]. Join query can exploit 3D stacked PIM as it is characterized by irregular access patterns, but near-memory algorithms are required that consider data placement and communication cost and exploit locality with in one stack as much as possible [24]

PIM for Data Re-organization operations: Another application of PIM is to accelerate data access and to help CPU cores to compute on complex linked data structures by efficiently packing them into the cache. Using strided DMA units, gather/scatter hardware and in-memory scratchpad buffers, the programmable near memory data rearrangement engines proposed in [14] perform fill and drain operations to gather the blocks of application data structures.

2.3.2 In-Storage Processing

Ranganathan et al. [30] propose nano-stores that co-locates processors and non-volatile memory on the same chip and connect to one another to form a large cluster for data-centric workloads that operate on more diverse data with I/O intensive, often random data access patterns and limited locality. Chang et al. [12] examine the potential and limit of designs that move compute in close proximity of NVM based data stores. The limit study demonstrates significant potential of this approach (3-162x improvement in energy-delay product) particularly for I/O intensive workloads. Wang et al. [33] observe that NVM is often naturally incorporated with basic logic like data comparison write or flip-n-write module and exploit the existing resources inside memory chips to accelerate the key non-compute intensive functions of emerging big data applications.

3. BIG DATA FRAMEWORKS AND NDP

3.1 Motivation

Even though NDP seems promising for applications like map-reduce, machine learning algorithms, SQL queries and graph analytics, but the existing literature lacks a study that identifies the potential of NDP for big data processing frameworks like Apache Spark, which run on top of Java Virtual Machine and use map-reduce programming model to enable machine learning, graph analysis and SQL processing on batched and streaming data. One can argue that previous NDP proposals made only by studying the algorithms can be extrapolated to the big data frameworks but we refute the argument by stating that earlier proposal of using 3D-Stacked PIM for map reduce applications [16, 27] was

motivated by the fact that the performance of map phase is limited by the memory bandwidth. Our experiments show that Apache Spark based map-reduce workloads don't fully utilize the available memory bandwidth. Prior work [10] also shows that high bandwidth memories are not needed for Apache Spark based workloads.

3.2 Methodology

Our study of identifying the potential of NDP to boost the performance of Spark workloads is based on matching the characteristics of Apache Spark based workloads to different forms of NDP (2D integrated PIM, 3D Stacked PIM, ISP)

3.2.1 Workloads

Our selection of benchmarks is inspired by [10]. We select the benchmarks based on following criteria;(a) workloads should cover a diverse set of Spark lazy transformations and actions, (b) workloads should be common among different big data benchmark suites available in the literature and (c) workloads have been used in the experimental evaluation of Map-Reduce frameworks. Table 1 shows the description of benchmarks and the breakdown of each benchmark into transformations and actions are given in Table 2. Batch processing workloads from Spark-core, Spark MLlib, Graph-X and Spark SQL are subset of BigdataBench [32] and HiBench [15] which are highly referenced benchmark suites in the big data domain. Stream processing workloads used in the paper also partially cover the solution patterns for real-time streaming analytics [26].

The source codes for Word Count, Grep, Sort, and NaiveBayes are taken from BigDataBench [32], whereas the source codes for K-Means, Gaussian, and Sparse NaiveBayes are taken from Spark MLlib (which is Spark's scalable machine learning library [22]) examples available along with Spark distribution. Likewise, the source codes for stream processing workloads and graph analytics are also available from Spark Streaming and GraphX examples respectively. Spark SQL queries from BigDataBench have been reprogrammed to use DataFrame API. Big Data Generator Suite (BDGS), an open source tool is used to generate synthetic data sets based on raw data sets [23].

3.2.2 System Configuration

To perform our measurements, we use a current dual-socket Intel Ivy Bridge server (IVB) with E5-2697 v2 processors, similar to what one would find in a datacenter. Table 3 shows details about our test machine. Hyper-Threading and Turbo-boost are disabled through BIOS during the experiments as per Intel Vtune guidelines to tune software on the Intel Xeon processor E5/E7 v2 family [5]. With Hyper-Threading and Turbo-boost disabled, there are 24 cores in the system operating at the frequency of 2.7 GHz.

Table 4 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) and use Parallel Scavenge (PS) and Parallel Mark Sweep for young and old generations respectively as recommended in [8]. The heap size is chosen such that the memory consumed is within the system.

3.2.3 Measurement Tools and Techniques

We use linux iotop command to measure the total disk bandwidth. To find sustained maximum bandwidth, we compile the OpenMP version of STREAM [3] using Intel's ICC compiler. We use linux top command in batch mode and monitor only java process of Spark to measure %usr (percentage CPU used by user process) and %io (percentage CPU waiting for I/O)

We use Intel Vtune Amplifier [2] to perform general micro-

Table 1: Spark 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 (Sql_Agg)	implements aggregation query from BigdataBench using DataFrame API	Tables
	Join (Sql_Jo)	implements join query from BigdataBench using DataFrame API	
	Difference (Sql_Diff)	implements difference query from BigdataBench using DataFrame API	
	Cross Product (Sql_Cro)	implements cross product query from BigdataBench using DataFrame API	
	Order By (Sql_Ord)	implements order by query from BigdataBench using DataFrame API	
Spark Streaming	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.	Wikipedia Entries
	Stateful Word Count (StWc)	counts words cumulatively in text received from the network every sec starting with initial value of word count.	
	Network Word Count (NWc)	counts the number of words in the 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)	

Table 2: Converted Spark Operations in Workloads

Workload	Converted Spark Operation
Wc	Map, ReduceByKey, SaveAsTextFile
Gp	Filter, SaveAsTextFile
So	Map, SortByKey, SaveAsTextFile
Nb	Map, Collect, SaveAsTextFile
Km	Map, MapPartitions, MapPartitionsWithIndex, FlatMap,Zip, Sample, ReduceByKey,
Snb	Map, RandomSplit, Filter, CombineByKey
Svm	Map, MapPartitions, MapPartionswithIndex, Zip, Sample,
Logr	RandomSplit,Filter,MakeRDD,Union, TreeAggregate, CombineByKey, SortByKey
Pr	Coalesce, MapPartitionswithIndex, MapPartitions, Map, PartitionBy, ZipPartitions
Cc	
Tr	
Sql_Jo	Map, MapPartitions, SortMergeJoin, TungstenProject, TungstenExchange, TungstenSort, ConverToSafe
Sql_Diff	Map, MapPartitions, SortMergeOuterJoin, TungstenProject, TungstenExchange, TungstenSort, ConverToSafe, ConverToUnsafe
Sql_Cro	Map, MapPartitions, SortMergeJoin, TungstenProject, TungstenExchange, TungstenSort, ConverToSafe, ConverToUnsafe
Sql_Agg	Map, MapPartitions, TungstenProject, TungstenExchange, TungstenAggregate, ConvertToSafe
Sql_Ord	Map, MapPartitions, TakeOrdered
WWc	FlatMap, Map, ReduceByKeyAndWindow
StWc	FlatMap, Map, UpdateStateByKey
NWc	FlatMap, Map, ReduceByKey

Table 3: Machine Details.

Component	Details	
Processor	Intel Xeon E5-2697 V2, Ivy Bridge micro-architecture	
	Cores	12 @ 2.7GHz
	Threads	1 per Core
	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 4: 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		

architecture exploration and to collect hardware performance counters. All measurement data are the average of three measure runs; Before each run, the file 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 [35] to study the micro-architectural performance of the workloads. Earlier studies on profiling of big data workloads shows the efficacy of this method in identifying the micro-architectural bottlenecks [7, 18, 36]. The top-down method requires following metrics described in Table 5, whose definition are taken from Intel Vtune on-line help [2].

4. EVALUATION

4.1 The case of ISP for Spark

Figure 1b shows the average amount of data read from and written to the disk per second for different Spark workloads. The data reveal that on average across the workloads,

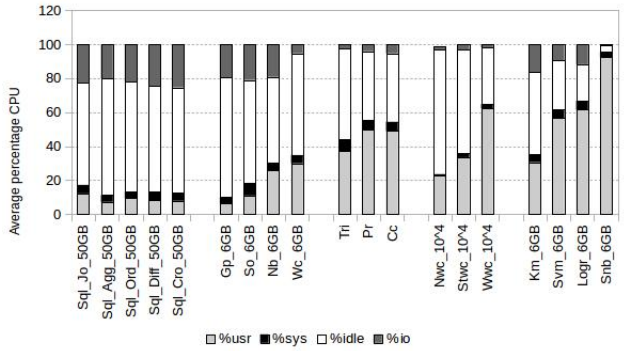
Table 5: Metrics for Top-Down Analysis of Workloads

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.

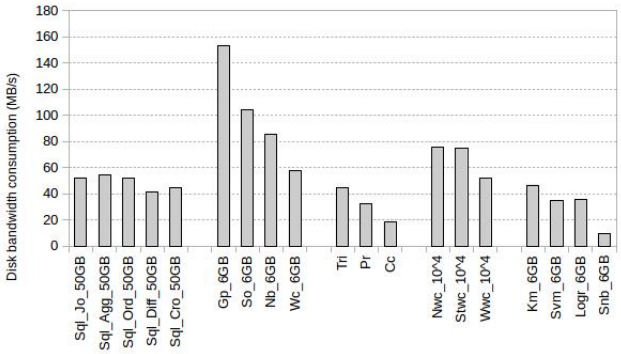
total disk bandwidth consumption is 56 MB/s. The SATA HDD installed in the machine under test can support up to 164.5 MB/s of 128 KB sequential reads and writes. However, the average response time for 4 KB reads and writes are 1803.41ms and 1305.66ms respectively [4]. This implies that Spark workloads do not saturate the bandwidth of SATA HDD, Earlier work [8] shows severe degradation in the performance of Spark workloads using large datasets due to significant wait time on I/O. Hence, it is the latency of I/O operations that are detrimental to the performance of Spark workloads.

Figure 1a shows average percentage CPU, a) used by Spark java process, b) in system mode c) waiting for I/O and d) in idle state during the execution of different Spark workloads. Even though the number of Spark worker threads are equal to the number of CPUs available in the system, during the execution of Spark SQL queries, only 8.97% CPUs are in user mode, 22.93% CPUs are waiting for I/O and 63.52% CPUs are in idle state. We see similar characteristics for Grep and Sort.

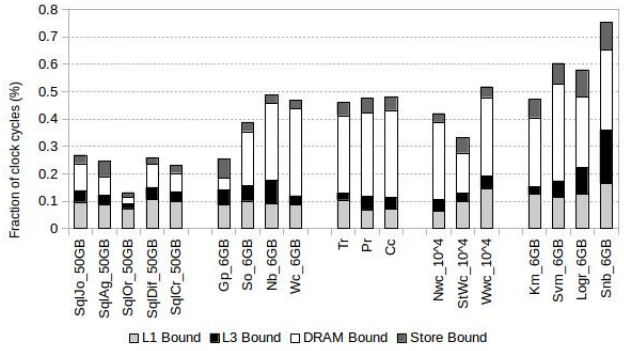
Grep, WordCount, Sort, NaiveBayes, Join, Aggregation, Cross Product, Difference and Orderby queries are all non iterative workloads, the data is read from and written to disk through out the execution period of workloads (see Figure 2a, 2b, 2c, 2d, 2e, 2f) and compute intensity varies from low to medium and the amount of data written to the disk also varies. For all these disk based workloads, we recommend in-storage processing. Since these workloads differ in the compute intensity, putting simple in-order cores would be less effective as compared to programmable logic, which can be programmed with workload specific hardware accelerators. Moreover, using hardware accelerators inside the NAND flash can free up the resources at the host CPU, which in turn can be used for other compute-intensive tasks.



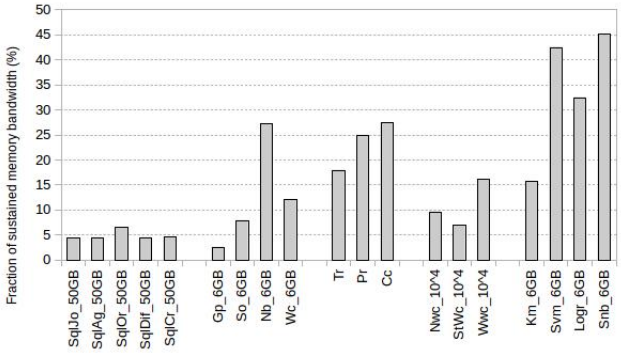
(a) Average percentage CPU in user mode, wait on I/O and in idle state during the execution of Spark workloads



(b) Spark workloads do not saturate the disk bandwidth



(c) Spark workloads are DRAM bound



(d) Spark workloads do not experience loaded latencies

Figure 1: Characterization of Spark workloads from NDP perspective

4.2 The case of PIM for Apache Spark

When Graph-X workloads are run, 45.15% CPUs are in the user mode, 3.98% CPUs wait for I/O and 44.63% CPUs are in the idle state. Pagerank, Connected Components and Triangle counting are iterative applications on graph data. All these workloads have a phase of heavy I/O with moderate CPU utilization followed by the phase of high CPU utilization and negligible I/O (see Figure 2j, 2k, 2l). These workloads are dominant by the second phase.

During the execution of stream processing workloads, 39.52% CPUs are in the user mode, 2.29% CPUs wait for I/O and 55.78% CPUs are in the idle state. The wait time on I/O for stream processing workloads is negligible (see Figure 2g, 2h, 2i) due to the streaming nature of the workloads but the CPU utilization also varies from low to high.

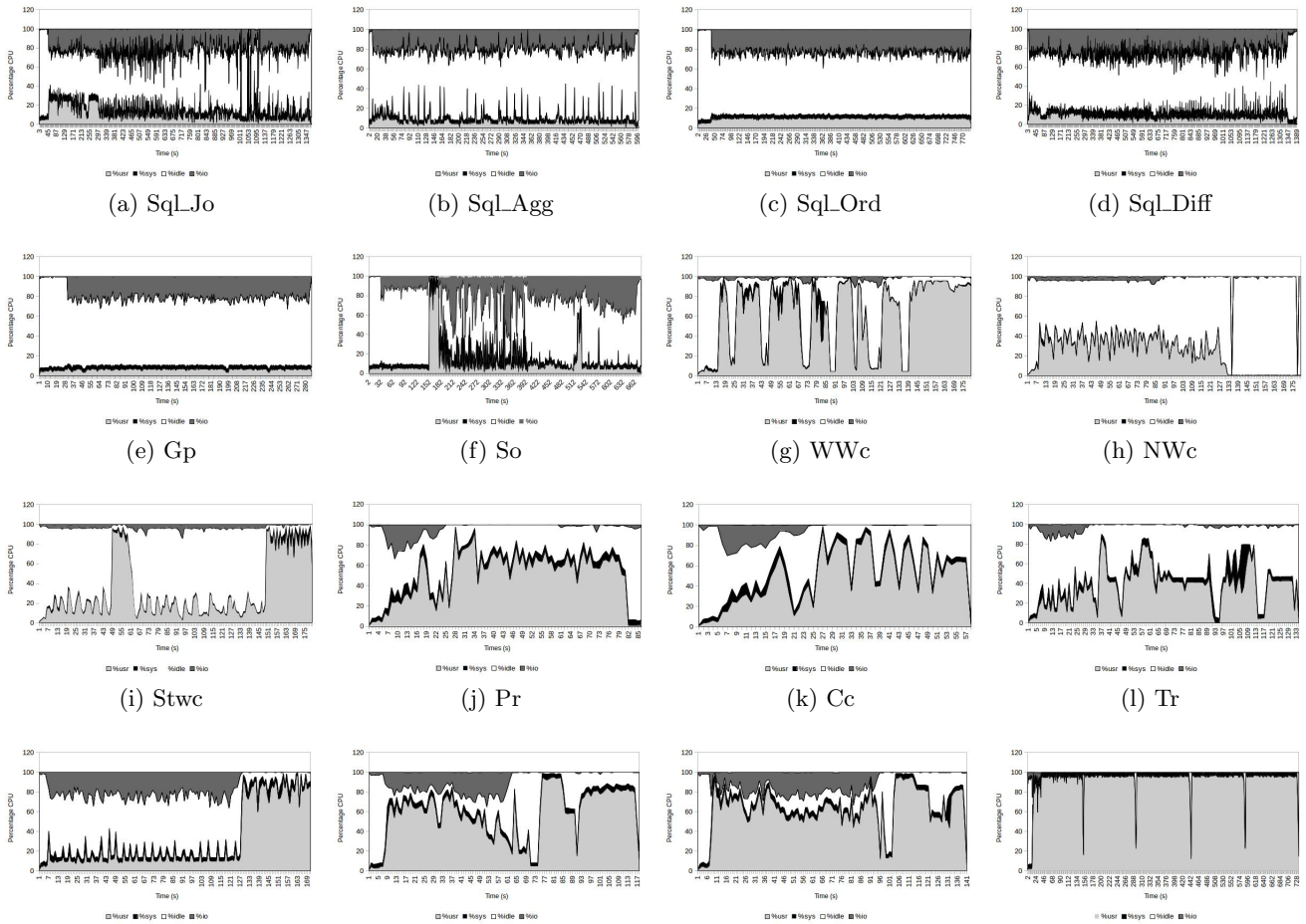
For Spark MLlib workloads, the percentage of CPUs in user mode, waiting for I/O and in idle state are 60.27%, 9.56% and 25.48%. SVM and Logistic Regression are phasic in terms of I/O (see Figure 2n, 2o). The training phase has significant I/O and also high CPU utilization, whereas the testing phase has negligible I/O and high CPU utilization because before the training starts, the input data is split into training and testing data and are cached in the memory.

Since DRAM bound stalls are higher than L3 bound stalls and L1 bound stalls for most of the Graph-X, Spark Spark Streaming and Spark MLlib workloads (see Figure 1c), it means that CPUs are stalled waiting for the data to be fetched from the main memory and not by the caches (for detailed analysis see [7–9]). So, instead of moving the data back and forth through the cache hierarchy in between the iterations, it would be beneficial to use programmable logic based processing-in-memory. As a result, application specific hardware accelerators are brought closer to the data, which will reduce the data movement and improve the performance of Spark workloads.

4.3 The case of 2D integrated PIM instead of 3D stacked PIM for Apache Spark

According to Jacob et al. [17], 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 unbounded 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.

3D-Stacked PIM based on Hybrid Memory Cube (HMC) enables significantly more bandwidth between the memory banks and the compute units as compared to 2D integrated



(a) Sql_Jo (b) Sql_Agg (c) Sql_Ord (d) Sql_Diff

(e) Gp (f) So (g) WWc (h) NWc

(i) Stwc (j) Pr (k) Cc (l) Tr

(m) Km (n) Svm (o) Logr (p) Snb

Figure 2: Execution time breakdown for Spark workloads

PIM, e.g. maximum theoretical bandwidth of 4 DDR3-1066 is 68.2 GB/s where as 4 HMC links provide 480 GB/s [29]. If the workload is operating in the exponential region on bandwidth vs latency curve of DDR3 based system, using HMC will move the workload to operate again in the constant region and average memory latency equals idle latency of the system. On the other hand, if the workloads are not bounded by the memory bandwidth, NDP architecture based on 3D-stacked PIM would not be able to fully utilize the excessive bandwidth and goal of reducing the data movement can be achieved instead by 2D integrated PIM.

Figure 1d shows the average bandwidth consumption as a fraction of sustained maximum bandwidth. The data reveal Spark workloads consume less than 40% of sustained maximum bandwidth at 1866 MT/s data transfer rate and thus operate in the constant region. Awan et al. [10] study the bandwidth consumption of Spark workloads during the whole execution time of the workloads and show that even when the peak bandwidth utilization goes into the exponential region, it lasts only for a short period of time and thus, have a negligible impact on the performance. Thus we envision 2D integrated PIM instead of 3D stacked PIM for Apache Spark.

4.4 The case of Hybrid 2D integrated PIM and ISP for Spark

K-means is also an iterative algorithm. It has two distinct phases (see Figure 2m), heavy I/O phase followed by negligible I/O phase. The heavy IO phase has low cpu utilization. This phase implements kmeans|| initialization method to assign initial values to the clusters. This phase can be mapped to hardware accelerators in the programmable logic inside the storage, where as the main clustering algorithm can be mapped to 2D integrated PIM.

5. CONCLUSION

We study the characteristics of Apache Spark workloads from the NDP perspective and and position ourselves as follows;

- Spark workloads, which are not iterative and have high ratio of % cpu waiting for I/O to % cpu in user mode like SQL queries, filter, word count and sort are ideal candidates for ISP.
- Spark workloads, which have low ratio of % cpu waiting for I/O to % cpu in user mode like stream processing and iterative graph processing workloads are

bound by latency of frequent accesses to DRAM and are ideal candidates for 2D integrated PIM.

- Spark workloads, which are iterative and have moderate ratio of % cpu waiting for I/O to %cpu in user mode like K-means, have both I/O bound and memory bound phases and hence will benefit from the combination of 2D integrated PIM and ISP.

- To satisfy the varying compute demands of Spark workloads, we envision an NDP architecture with programmable logic based hybrid ISP and 2D integrated PIM.

Future work involves quantifying the performance gain for Spark workloads achievable through programmable logic based ISP and 2D integrated PIM.

Acknowledgments

This work is supported by Erasmus Mundus Joint Doctorate in Distributed processing (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).

6. REFERENCES

[1] Hybrid memory cube consortium. hybrid memory cube specification 2.0. http://www.hybridmemorycube.org/specification-v2-download-form/,Nov.2014.

[2] Intel Vtune Amplifier XE 2013.

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

[4] Toshiba SATA HDD Enterprise, Performance Review.

[5] 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.

[6] Ahn, J., Hong, S., Yoo, S., Mutlu, O., and Choi, K. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), ACM, pp. 105–117.

[7] Awan, A. J., Brorsson, M., Vlassov, V., and Ayguade, E. 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* (2015), IEEE, pp. 1–8.

[8] Awan, A. J., Brorsson, M., Vlassov, V., and Ayguade, E. *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.

[9] Awan, A. J., Brorsson, M., Vlassov, V., and Ayguade, E. Micro-architectural characterization of apache spark on batch and stream processing workloads. In *Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom), 2016 IEEE International Conferences on* (2016), IEEE, pp. 59–66.

[10] Awan, A. J., Brorsson, M., Vlassov, V., and Ayguade, E. Node architecture implications for in-memory data analytics on scale-in clusters. In *Big Data Computing Applications and Technologies (BDCAT), 2016 IEEE/ACM 3rd International Conference on* (2016), IEEE, pp. 237–246.

[11] Bender, M. A., Berry, J., Hammond, S. D., Moore, B., Moseley, B., and Phillips, C. A. k-means clustering on two-level memory systems. In *Proceedings of the 2015 International Symposium on Memory Systems* (2015), ACM, pp. 197–205.

[12] Chang, J., Ranganathan, P., Mudge, T., Roberts, D., Shah, M. A., and Lim, K. T. A limits study of benefits from nanostore-based future data-centric system architectures. In *Proceedings of the 9th conference on Computing Frontiers* (2012), ACM, pp. 33–42.

[13] del Mundo, C. C., Lee, V. T., Ceze, L., and Oskin, M. Ncam: Near-data processing for nearest neighbor search. In *Proceedings of the 2015 International Symposium on Memory Systems* (2015), ACM, pp. 274–275.

[14] Gokhale, M., Lloyd, S., and Hajas, C. Near memory data structure rearrangement. In *Proceedings of the 2015 International Symposium on Memory Systems* (2015), ACM, pp. 283–290.

[15] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. 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.

[16] Islam, M., Scrbak, M., Kavi, K. M., Ignatowski, M., and Jayasena, N. Improving node-level mapreduce performance using processing-in-memory technologies. In *Euro-Par 2014: Parallel Processing Workshops* (2014), Springer, pp. 425–437.

[17] Jacob, B. The memory system: you can't avoid it, you can't ignore it, you can't fake it. *Synthesis Lectures on Computer Architecture 4*, 1 (2009), 1–77.

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

[19] Kreps, J., Narkhede, N., Rao, J., et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB* (2011), pp. 1–7.

[20] Lee, J. H., Sim, J., and Kim, H. Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models.

[21] Loh, G., Jayasena, N., Oskin, M., Nutter, M., Roberts, D., Meswani, M., Zhang, D., and Ignatowski, M. A processing in memory taxonomy

and a case for studying fixed-function pim. In *Workshop on Near-Data Processing (WoNDP)* (2013).

[22] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. Mllib: Machine learning in apache spark. *arXiv preprint arXiv:1505.06807* (2015).

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

[24] Mirzadeh, N., Koçberber, Y. O., Falsafi, B., and Grot, B. Sort vs. hash join revisited for near-memory execution. In *5th Workshop on Architectures and Systems for Big Data (ASBD 2015)* (2015), no. EPFL-CONF-209121.

[25] Nai, L., and Kim, H. Instruction offloading with hmc 2.0 standard: A case study for graph traversals. In *Proceedings of the 2015 International Symposium on Memory Systems* (2015), ACM, pp. 258–261.

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

[27] Pugsley, S. H. *Opportunities for near data computing in MapReduce workloads*. PhD thesis, The University of Utah, 2015.

[28] Pugsley, S. H., Jestes, J., Zhang, H., Balasubramonian, R., Srinivasan, V., Buyuktosunoglu, A., Li, F., et al. Ndc: Analyzing the impact of 3d-stacked memory+ logic devices on mapreduce workloads. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on* (2014), IEEE, pp. 190–200.

[29] Radulovic, M., Zivanovic, D., Ruiz, D., de Supinski, B. R., McKee, S. A., Radojković, P., and Ayguadé, E. Another trip to the wall: How much will stacked dram benefit hpc? In *Proceedings of the 2015 International Symposium on Memory Systems* (2015), ACM, pp. 31–36.

[30] Ranganathan, P. From microprocessors to nanostores: Rethinking data-centric systems (vol 44, pg 39, 2010). *COMPUTER 44*, 3 (2011), 6–6.

[31] Siegl, P., Buchty, R., and Berekovic, M. Data-centric computing frontiers: A survey on processing-in-memory. In *Proceedings of the Second International Symposium on Memory Systems* (2016), ACM, pp. 295–308.

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

[33] Wang, Y., Han, Y., Zhang, L., Li, H., and Li, X. Propram: exploiting the transparent logic resources in non-volatile memory for near data computing. In *Proceedings of the 52nd Annual Design Automation Conference* (2015), ACM, p. 47.

[34] Xi, S. L., Babarinsa, O., Athanassoulis, M., and Idreos, S. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware (DaMoN)* (2015).

[35] Yasin, A. 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.

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

[37] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. 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.