Node architecture implications for in-memory data analytics on scale-in clusters

Conference Paper · December 2016

DOI: 10.1145/3006299.3006319

Ahsan Javed Awan
Ericsson
24 PUBLICATIONS **85** CITATIONS
SEE PROFILE

Vladimir Vlassov
KTH Royal Institute of Technology
122 PUBLICATIONS **772** CITATIONS
SEE PROFILE

Mats Brorsson
KTH Royal Institute of Technology
84 PUBLICATIONS **801** CITATIONS
SEE PROFILE

Node Architecture Implications for In-Memory Data Analytics on Scale-in Clusters

Ahsan Javed Awan
ajawan@kth.se

Mats Brorsson
matsbror@kth.se

Vladimir Vlassov
vladv@kth.se

Eduard Ayguade*
eduard.ayguade@bsc.es

KTH Royal Institute of Technology
Department of Software and Computer Systems

*Barcelona Super Computing Center(BSC)
Technical University of Catalunya(UPC)

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. Recent studies propose scale-in clusters with in-storage processing devices to process big data analytics with Spark However the proposal is based solely on the memory bandwidth characterization of in-memory data analytics and also does not shed light on the specification of host CPU and memory. Through empirical evaluation of in-memory data analytics with Apache Spark on an Ivy Bridge dual socket server, we have found that (i) simultaneous multi-threading is effective up to 6 cores (ii) data locality on NUMA nodes can improve the performance by 10% on average, (iii) disabling next-line L1-D prefetchers can reduce the execution time by up to 14%, (iv) DDR3 operating at 1333 MT/s is sufficient and (v) multiple small executors can provide up to 36% speedup over single large executor.

Keywords

NUMA, SMT, 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. 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 big data sets and modern scale-up servers are adequate to run analytics jobs [10]. Additionally, commonly used predictive analytics such as machine learning algorithms, work on filtered data sets that easily fit into the memory of modern scale-up servers. Moreover, the today's scale-up servers can have CPU, memory, and persistent storage resources in abundance at affordable prices. Thus, we envision a small cluster of scale-up servers to be the preferable choice for processing data analytics. Choi et al. [16, 17] define such clusters as scale-in clusters. They propose scale-in clusters with in-storage processing devices to reduce data movements towards CPUs. However, their proposal is based solely on the memory bandwidth characterization of in-memory data analytics with Spark and does not shed light on the specification of host CPU and memory.

While Phoenix [33], Ostrich [14] and Polymer [35] 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 [34], is getting popular in the industry because it enables in-memory processing, scales out to a large number of commodity machines and provides a unified framework for batch and stream processing of big data workloads. Like Choi et al. [16], we also favour Apache Spark to be the big data processing platform for scale-in clusters. By quantifying the architectural impact on the performance of in-memory data analytics with Spark on an Ivy Bridge server, we define the specifications of host CPU and memory and argue that a node with fixed function hardware accelerators near DRAM and NVRAM suits better for the processing of in-memory data analytics with Spark on scale-in clusters. Our contributions are:

- We evaluate the impact of NUMA locality on the performance of in-memory data analytics with Spark.
- We analyze the effectiveness of Hyper-threading and existing prefetchers in scale-up server to hide data access latencies for in-memory data analytics with Spark.
- We quantify the potential of high bandwidth memories to boost the performance of in-memory data analytics with Spark.
- We recommend how to configure scale-up server and Spark to accelerate in-memory data analytics with Spark

BDCAT'16, December 06-09, 2016, Shanghai, China
© 2016 ACM. ISBN 978-1-4503-4617-7/16/12...$15.00
DOI: http://dx.doi.org/10.1145/3006299.3006319

2. BACKGROUND

2.1 Spark

Spark is a cluster computing framework that uses Resilient Distributed Datasets (RDDs) [34] 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. Spark assigns tasks to the executor pool of threads and executes all tasks within a stage before moving on to the next stage. Finally, once all jobs are completed, it saves the results to file system.

2.2 Spark on Modern Scale-up Servers

Our recent efforts on identifying the bottlenecks in Spark [11, 21] on scale-up server shows (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 and file I/O

We reproduce the multi-core scalability experiments from our previous work [11, 21] to highlight the performance issues incurred by Spark workloads on scale-up servers. Each benchmark is run with 1, 6, 12, 18 and 24 executor pool threads. The size of input dataset is 6 GB. For each run, we set the CPU affinity of the Spark process to emulate hardware with the same number of cores as the worker threads. The cores are allocated from one socket first before switching to the second socket. Figure 1a plots speed-up as a function of the number of cores. It shows benchmarks scale linearly up to 4 cores within a socket. Beyond 4 cores, the workloads exhibit sub-linear speed-up, e.g., at 12 cores within a socket, average speed-up across workloads is 7.45. This average speed-up increases up to 8.74 when the Spark process is configured to use all 24 cores in the system. The performance gain of mere 17.3% over the 12 cores case suggest Spark applications gain less by using more than 12-core executors. Figure 1b shows pipeline-slots breakdown of Spark workloads.They are configured to run at 24 cores. The data show that most of the benchmarks are back-end bound because DRAM bound stalls are the primary bottleneck (see Figure 1c) and remote DRAM accesses incur additional latency (see Figure 1d).

Simultaneous multi-threading and hardware prefetching are effective ways to hide data access latencies and additional latency overhead 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 test the aforementioned techniques and study their implications on the architecture of node in scale-in cluster for in-memory data analytics with Spark.

3. METHODOLOGY

Our study of the architectural impact on in-memory data analytics is based on an empirical study of the 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 non uniform memory access (NUMA) nodes, hardware prefetchers, and hyper-threading, on in-memory data analytics with Spark

3.1 Workloads

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. Batch processing workloads from Spark-core, Spark MLlib, Graph-X and Spark SQL are subset of BigdataBench [30] and HiBench [19] 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 [29].

The source codes for Word Count, Grep, Sort, and NaiveBayes are taken from BigDataBench [30], whereas the source codes for K-Means, Gaussian, and Sparse NaiveBayes are taken from Spark MLlib (which is Spark's scalable machine learning library [27]) 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 [28].

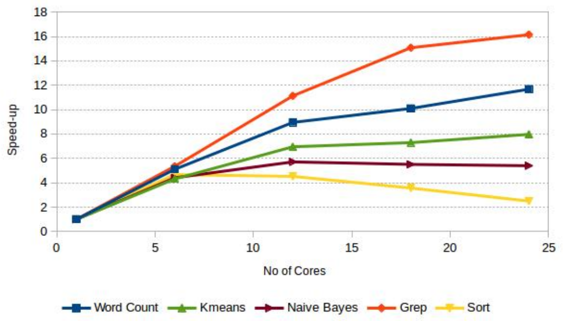
3.2 System Configuration

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 2 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 [9]. With Hyper-Threading and Turbo-boost disabled, there are 24 cores in the system operating at the frequency of 2.7 GHz.

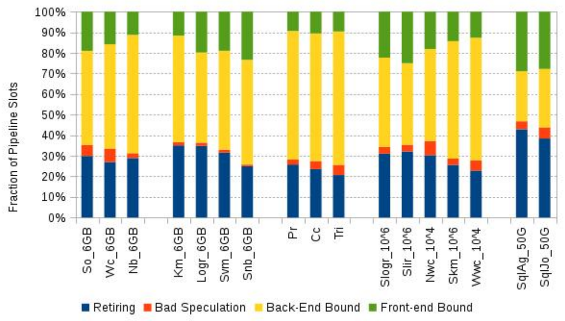
Table 3 also lists the parameters of JVM and Spark after tuning. For our experiments, we configure Spark in local mode in which driver and executor run inside a single JVM. We use HotSpot JDK version 7u71 configured in server mode (64 bit). The Hotspot JDK provides several parallel/concurrent GCs out of which we use Parallel Scavenge (PS) and Parallel Mark Sweep for young and old generations respectively as recommended in [11]. The heap size is chosen such that the memory consumed is within the system. The details on Spark internal parameters are available [7].

3.3 Measurement Tools and Techniques

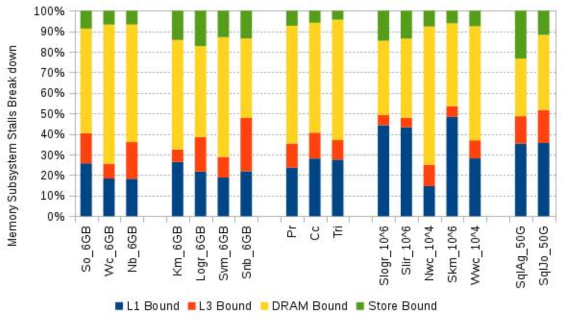
We configure Spark to collect GC logs which are then parsed to measure time (called real time in GC logs) spent in garbage collection. We rely on the log files generated by Spark to calculate the execution time of the benchmarks. We use Intel Vtune Amplifier [4] to perform general micro-



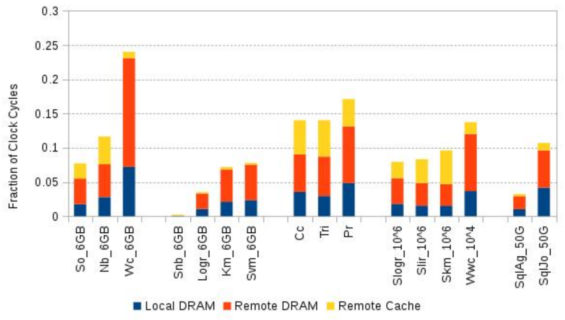
(a) Spark workloads don't benefit by adding more than 12 cores



(b) Spark workloads are back-end bound



(c) Spark workloads are DRAM bound



(d) Spark workloads have significant remote memory stalls

Figure 1: Top Down Analysis of Spark Workloads

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 (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 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
	Streaming Kmeans (Skm)	uses streaming version of K-Means clustering algorithm from Spark Mllib. The benchmark is run for 4 iterations with 8 desired clusters	Numerical Records
	Streaming Logistic Regression (Slogr)	uses streaming version of Logistic Regression algorithm from Spark Mllib. The benchmark is run for 4 iterations with 8 desired clusters	
	Streaming Linear Regression (Slir)	uses streaming version of Logistic Regression algorithm from Spark Mllib. The benchmark is run for 4 iterations with 8 desired clusters	
Spark SQL	Aggregation (SqlAg)	implements aggregation query from BigdataBench using DataFrame API	Tables
	Join (SqlJo)	implements join query from BigdataBench using DataFrame API	

Table 2: 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 3: 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. We use numactl [6] to control the process and memory allocation affinity to a particular socket. We use hwloc [13] to get the CPU ID of hardware threads. We use msr-tools [5] 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. We find variance in measurements to be negligible and hence do not use box plots. Through concurrency analysis in Intel Vtune, we find 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.

3.4 Top-Down Analysis Approach

We use top-down analysis method proposed by Yasin [31] to study the micro-architectural performance of the workloads because earlier studies on profiling on big data workloads shows the efficacy of this method in identifying the micro-architectural bottlenecks [21,25,32]. 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 for each CPU core, there are four pipeline slots available per clock cycle. At issue point, each pipeline slot is classified into one of four base categories: Front-end Bound, Back-end Bound, Bad Speculation and Retiring. If a micro-operation is issued in a given cycle, it would eventually either get retired or 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 4, whose definition are taken from Intel Vtune on-line help [4].

Table 4: 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.

4. EVALUATION

4.1 How much performance gain is achievable by co-locating the data and computations on NUMA nodes for 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 another 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.

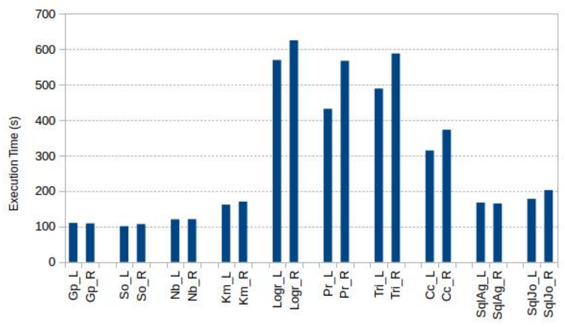
To evaluate the impact of NUMA on Spark workloads, we run the benchmarks in two configurations: a) Local DRAM, where Spark process is bound to socket 0 and memory node 0, i.e. computations and data accesses are co-located, and b) Remote DRAM, where spark process is bound to socket 0 and memory node 1, i.e. all data accesses incur the additional latency. The input data size for the workloads is chosen as 6GB to ensure memory working set sizes fit socket memory. Spark parameters for the two configurations are given in Table 5.

Table 5: Machine and Spark Configuration for NUMA Evaluation

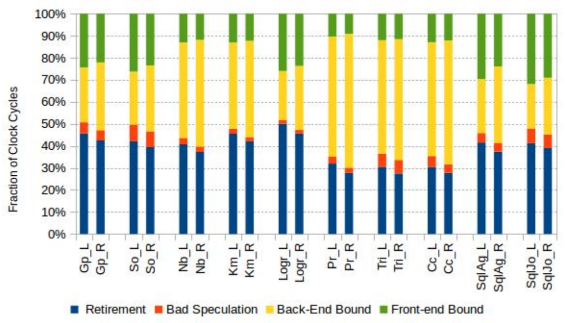
		Local DRAM (L)	Remote DRAM (R)
Hardware	Socket ID	0	0
	Memory Node ID	0	1
	No. of cores	12	12
	No. of threads	12	12
Spark	spark.driver.cores	12	12
	spark.default.parallelism	12	12
	spark.driver.memory (GB)	24	24

Figure 2a 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 2c), retiring category degrades by only 10.79%, Back-end bound stalls increases by 20.26%, bad speculation decreases by 13.08% and front-end bound stalls decreases by 12.66% on average as shown in Figure 2b. 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 2d).

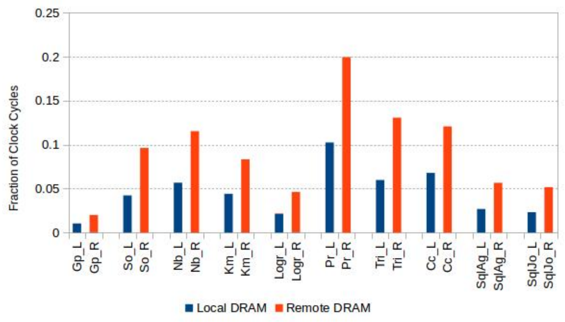
Implications: In-memory data analytics with Spark should use data from local memory on a multi-socket node of the scale-in cluster.



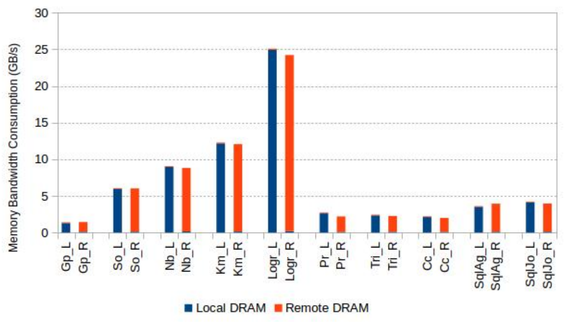
(a) Performance degradation due to NUMA is 10% on average across the workloads.



(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

Figure 2: NUMA Characterization of Spark Benchmarks

4.2 Is simultaneous multi-threading effective for in-memory data analytics with Spark?

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 baseline single threaded configuration where Spark process is bound to two physical cores b) 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 6. We also experiment with baseline 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 effects and the size of input data for the workloads is 6GB.

Table 6: 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

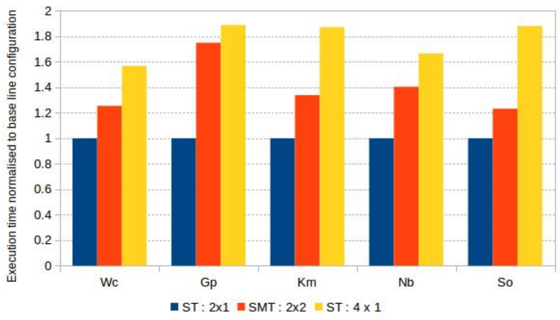
Figure 3a shows SMT provides 39.5% speedup on average across the workloads over baseline configuration, while the upper-bound configuration provided 77.45% on average across the workloads. The memory bandwidth in SMT case also keeps up with the multi-core case it is 20.54% less than that of the multi-core version on average across the workloads as shown in Figure 3c. Figure 3b 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 [2]. The data reveal 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 [21]

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 3d implying that Hyper-threading is effective in hiding the memory access latency for Spark workloads.

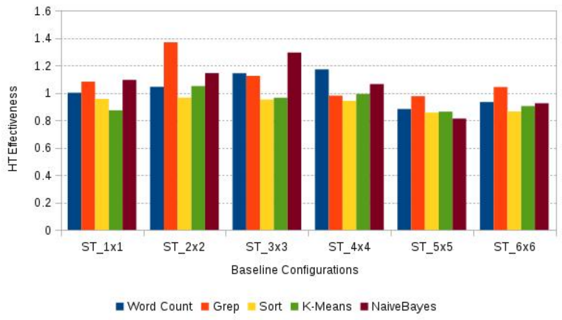
Implications: 6 HT cores per socket are sufficient for a node in scale-in clusters.

4.3 Are existing hardware prefetchers in modern scale-up servers effective for in-memory data analytics with Spark?

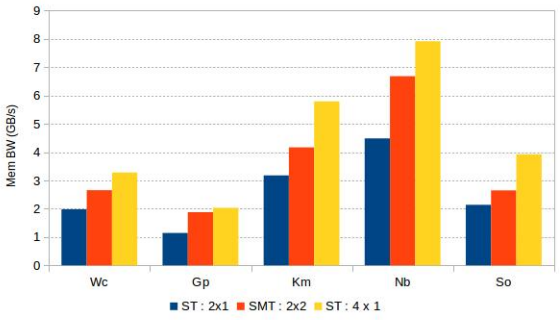
Prefetching is a promising approach to hide memory access latency by predicting the future memory accesses and fetching the corresponding memory blocks into the cache ahead of explicit accesses by the processor. Intel Ivy Bridge Server has two L1-D prefetchers and two L2 prefetchers.The



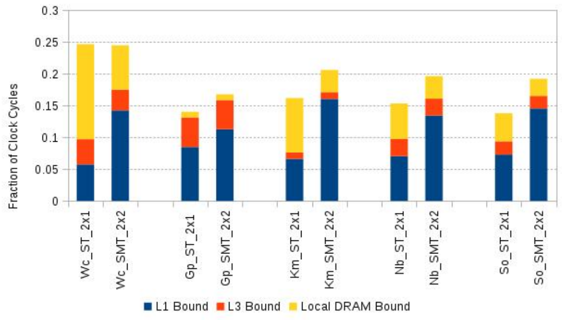
(a) Multi-core vs Hyper-Threading



(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

Figure 3: Hyper Threading is Effective

description about prefetchers is given in Table 7. This information is taken from Intel software forum [1].

Table 7: 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 that comprises 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.

Figure 4a shows 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 DCU prefetcher is ineffective.

Figure 4b shows L2 miss rate increases by at most 5% in Grep when L2 adjacent cache line prefetcher disabled. In some cases for example sort and naivebayes, disabling L2 adjacent line prefetcher reduces the L2 miss rate. This implies L2 adjacent cache line prefetcher is ineffective. It also shows L2 miss rate increases by 14.31% on average across the workloads when L2 hardware prefetcher is disabled.

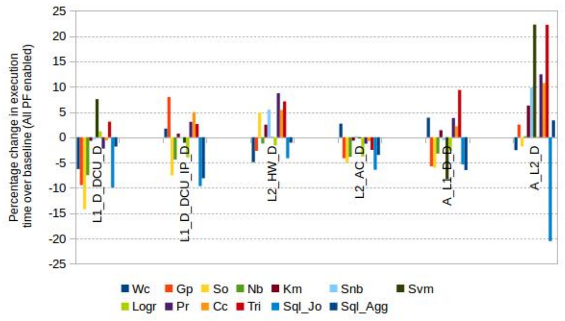
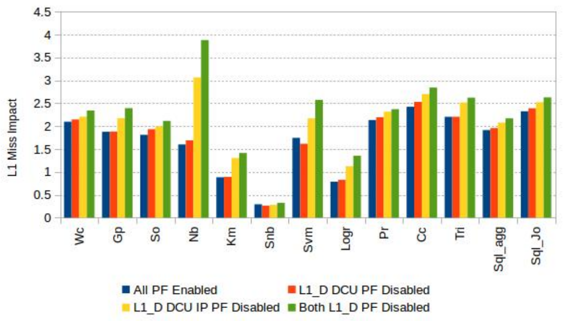
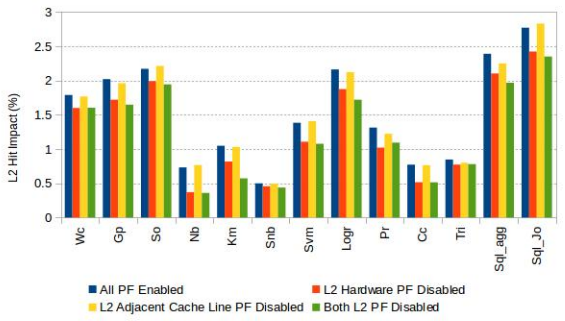


Figure 5: Disabling L1-D next-line and L2 Adjacent Cache Line Prefetchers can reduce the execution of Spark jobs up to 15% and 5% respectively

Figure 5 shows percentage change in execution time of Spark workloads over baseline configuration (all prefetchers are enabled). The data show L1-D next-line and adjacent cache line L2 prefetchers have a negative impact on Spark workloads and disabling them improves the performance of Spark workloads on average by 7.9% and 2.31% respectively.



(a) L1-D DCU Prefetcher is ineffective



(b) Adjacent Cache Line L2 Prefecher is ineffective

Figure 4: Evaluation of Hardware Prefetchers

This implies simple next-line hardware prefetchers in modern scale-up servers are ineffective for in-memory data analytics.

Implications: Cores without next-line hardware prefetchers are suitable for a node in scale-in clusters.

4.4 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. [20], 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. Note that maximum sustained bandwidth is 65% to 75% of the theoretical maximum for server workloads.

Using the formula taken from Intel's document [9], we calculate maximum theoretical bandwidth, per socket, for a 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 OpenMP version of STREAM [8] using Intel's ICC compiler. On running the benchmark, we find the maximum sustained bandwidth to be 92 GB/s.

Figure 6 shows the average bandwidth consumption as a fraction of sustained maximum bandwidth for different BIOS configurable data transfer rates of DDR3 memory. The data reveal Spark workloads consume less than 40% of sustained maximum bandwidth at 1866 data transfer rate and thus operate in the constant region. By lowering the data transfer rates to 1066, the majority of workloads from Spark core, all workloads from Spark SQL, Spark Streaming, and Graph-X still operate on the boundary of linear region whereas 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 the linear region. From the bandwidth consumption over time curves of the Km, Snb and Nb in Figure 7, it can be seen 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. As we enlarge the input data set, Figure 8a shows average memory bandwidth consumption decreases from 20.7 GB/s in the 6 GB case to 13.7 GB/s in the 24 GB case on average across the workloads. Moreover, wait time on file I/O becomes dominant at large input data sets as shown in Figure 8b.

Implications: High Bandwidth Memories like Hybrid Memory cubes [3] are inessential for in-memory data analytics with Spark and DDR3-1333 is sufficient for a node in scale-in clusters and the future single node should include faster persistent storage devices like SSD or NVRAM to re-

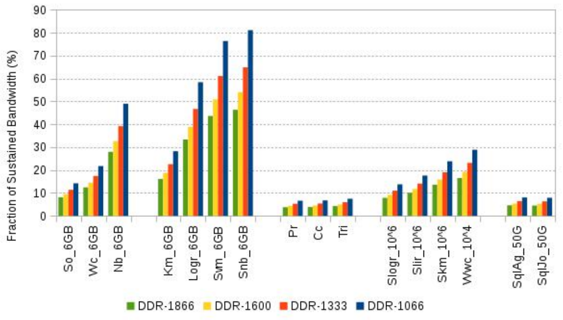


Figure 6: Spark workloads do not experience loaded latencies

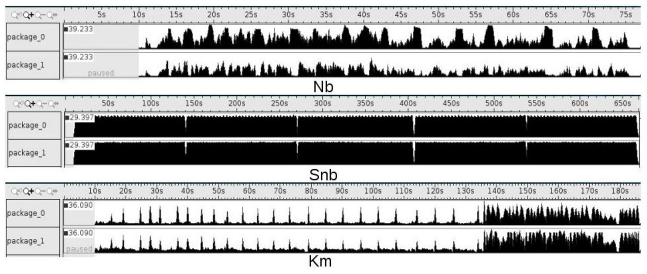
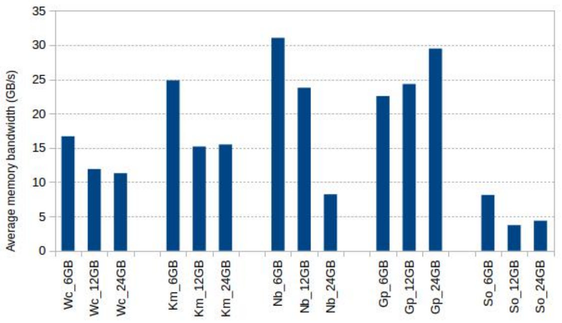
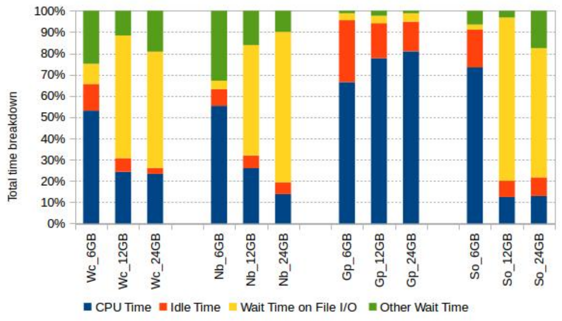


Figure 7: Bandwidth Consumption over time



(a) Memory traffic decreases with data size.



(b) Wait time becomes dominant at larger datasets due to significant increase in file I/O operations.

Figure 8: Effect of Data Volume on Spark workloads

duce the wait time on file I/O.

4.5 Are multiple small executors (which are java processes in Spark that run computations and store data for the application) better than single large executor?

With the 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 a maximum number of threads in the system. Table 8 lists down the configuration details. 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 8: 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 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 4.60x less than the 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.

Implications: In-memory data analytics with Spark should use multiple executors with heap size smaller than 32GB instead of single large executor on the node of the scale-in cluster.

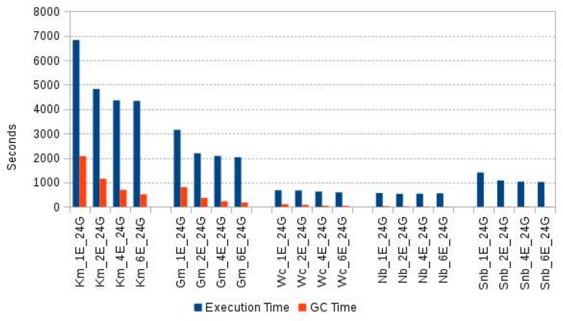


Figure 9: Multiple small executors are better than single large executor due to reduction in GC time

5. THE CASE OF NEAR DATA COMPUTING BOTH IN DRAM AND IN STORAGE

Since DRAM scaling is lagging behind the Moore's law, increasing DRAM capacity will be a challenge. NVRAM, on the other hand, shows a promising trend in terms of capacity scaling. Since Spark based workloads are I/O intensive when the input datasets don't fit in memory and are bound on latency when they do fit in-memory, In-Memory processing, and In-storage processing can be combined together into a hybrid architecture where the host is connected to DRAM with custom accelerators and flash based NVRAM with integrated hardware units to reduce the data movement. We envision a single node with fixed function hardware accelerators both in DRAM and also in-Storage. Figure 10 shows the architecture.

Let's consider an example. Many transformations in Spark such as groupByKey, reduceByKey, sortByKey, join etc involve shuffling of data between the tasks. To organize the data for shuffle, spark generates set of tasks; map tasks to organize the data and a set of reduce tasks to aggregate it. Map output records from each task are kept in memory until they can't fit. At that point records are sorted by reduce tasks for which they are destined and then spilled to a single file. Since the records are dispersed throughout the memory, they results in poor cache locality and sorting them on

CPU will experience a significant amount of cache misses and using near DRAM hardware accelerators for sort function, this phase can be accelerated. If this process occurs multiple times, the spilled segments are merged later. On the reduce side, tasks read the relevant sorted blocks. A single reduce task can receive blocks from thousands of map tasks. To make this many-way merge efficient, especially in the case where the data exceeds the memory size, It is better to use hardware accelerators for merge function near the faster persistent storage device like NVRAM.

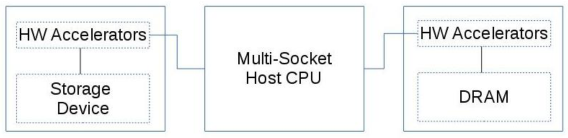


Figure 10: NDC Supported Single Node in Scale-in Clusters for in-Memory Data Analytics with Spark

6. RELATED WORK

Several studies characterize the behaviour of big data workloads and identify the mismatch between the processor and the big data applications [12, 18, 22–26, 30, 32]. None of the above-mentioned works analyze the impact of NUMA, SMT and hardware prefetchers on the performance of in-memory data analytics with Apache Spark.

Chiba et al. [15] also study the impact of NUMA, SMT and multiple executors on the performance of TPC-H queries with Apache Spark on IBM Power 8 server. However, their work is limited to Spark SQL library only. They only explore thread affinity, i.e. bind JVMs to sockets but allow the cross socket accesses. Our study covers the workloads not only from Spark SQL but also from Spark-core, Spark MLlib, Graph X and Spark Streaming. We use Intel Ivy bridge server. By using a diverse category of Spark workloads and a different hardware platform, our findings build upon Chiba's work. We give in-depth insights into the limited potential of NUMA affinity for Spark SQL workloads, e.g. Spark SQL queries exhibit 2 - 3% performance improvement by considering NUMA locality whereas Graph-X workloads show more than 20% speed-up because CPU stalled cycles on remote accesses are much less in Spark SQL queries compared to Graph-X workloads. We show the effectiveness of hyper-threading is due to the reduction in DRAM bound stalls and also show that HT is effective for Spark workloads only up to 6 cores. Besides that, we also quantify the impact of existing hardware prefetchers in scale-up servers on Spark workloads and quantify the DRAM speed sufficient for Spark workloads. Moreover, we derive insights about the architecture of a node in scale-in cluster for in-memory data analytics based on their performance characterization.

7. 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:

- 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 do 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.

We advise using executors with memory size less than or equal to 32GB and restrict each executor to use NUMA-local memory. We recommend enabling 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 [3] is not justified for in-memory data analytics with Spark.

Acknowledgments

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

References

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

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

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

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

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

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

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

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

[9] 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.

[10] Appuswamy, R., Gkantsidis, C., Narayanan, D., Hodson, O., and Rowstron, A. I. T. Scale-up vs scale-out for hadoop: time to rethink? In *ACM Symposium on Cloud Computing, SOCC* (2013), p. 20.

[11] 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.

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

[13] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. 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* (2010), IEEE, pp. 180–186.

[14] Chen, R., Chen, H., and Zang, B. 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* (2010), PACT '10, pp. 523–534.

[15] Chiba, T., and Onodera, T. Workload characterization and optimization of tpc-h queries on apache spark. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (April 2016), pp. 112–121.

[16] Choi, I. S., and Kee, Y.-S. Energy efficient scale-in clusters with in-storage processing for big-data analytics. In *Proceedings of the 2015 International Symposium on Memory Systems* (2015), ACM, pp. 265–273.

[17] Choi, I. S., Yang, W., and Kee, Y.-S. Early experience with optimizing i/o performance using high-performance ssds for in-memory cluster computing. In *Big Data (Big Data), 2015 IEEE International Conference on* (2015), IEEE, pp. 1073–1083.

[18] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B. 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* (2012), ASPLOS XVII, pp. 37–48.

[19] 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.

[20] 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.

[21] Javed Awan, A., 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.

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

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

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

[25] 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.

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

[27] 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).

[28] 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.

[29] 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.

[30] 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.

[31] 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).

[32] 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.

[33] Yoo, R. M., Romano, A., and Kozyrakis, C. 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.

[34] 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.

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