Performance Characterization of In-Memory Data Analytics on a Scale-up Server

AHSAN JAVED AWAN

Licentiate Thesis in Information and Communication Technology
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden 2016

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

TRITA-ICT 2016:07
ISBN 978-91-7595-926-9

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatexamen i informations och kommunikationsteknik måndagen den 23 maj 2016 klockan 09.15 i Ka-210, Electrum, Kungl Tekniska högskolan, Isafjordsgatan, 29, Kista.

Tryck: Universitetsservice US AB

Abstract

The sheer increase in volume of data over the last decade has triggered research in cluster computing frameworks that enable web enterprises to extract big insights from big data. While Apache Spark defines the state of the art in big data analytics platforms for (i) exploiting data-flow and in-memory computing and (ii) for exhibiting superior scale-out performance on the commodity machines, little effort has been devoted at understanding the performance of in-memory data analytics with Spark on modern scale-up servers. This thesis characterizes the performance of in-memory data analytics with Spark on scale-up servers.

Through empirical evaluation of representative benchmark workloads on a dual socket server, we have found that in-memory data analytics with Spark exhibit poor multi-core scalability beyond 12 cores due to thread level load imbalance and work-time inflation. We have also found that workloads are bound by the latency of frequent data accesses to DRAM. By enlarging input data size, application performance degrades significantly due to substantial increase in wait time during I/O operations and garbage collection, despite 10% better instruction retirement rate (due to lower L1 cache misses and higher core utilization).

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, (ii) disabling next-line L1-D prefetchers can reduce the execution time by up-to 14%. For GC impact, we match memory behaviour with the garbage collector to improve performance of applications between 1.6x to 3x. and recommend to use multiple small executors that can provide up-to 36% speedup over single large executor.

Sammanfattning

Det senaste årtiondets ökning av datavolym har uppmuntrat forskning kring ”cluster computing” och hur man möjliggör extrahering av insikter från stora datamängder. Trots att kända ramverk som till exempel Apache Spark definerar hur man utnyttjar beräkningar på strömmande data och på data som ligger resident i minnet, samt hur man uppnår skalbarhet med lätt tillängliga komponenter, så förstår man ännu inte fullt ut prestanda eller de analytiska-aspekterna av beräkning på data resident i minnet hos moderna flerkärniga servrar.

Denna avhandling behandlar karakterisering av minnesresident data analys i Apache Spark. Vi har genomfört empiriska undersökningar på välkända test-program som pekar på att skalbarheten hos Apache Spark är begränsad till 12 processorkärnor. De testprogram lider huvudsakligen av begränsningar i accesser till huvudminnet.

Vi fann att en ökning av problemets storlek (öningen av behandlad datamängd) medförde en 10% snabbare instruktionsexekvering, men även tillförde en ökad last på in- och utmatningsenheter och skräpsamling som gör att skalbarheten försämras ytterligare vid ökning av data-mängd.

We fann att moderna, flertrådade processorer döljer minnes-latenser väldigt bra. Vi har också observerat att det finns möjlighet för upp till 10% förbättrad prestanda om man tar hänsyn till var data är placerat i minnet. Denna förbättrade kan uppnås i såkallade NUMA system. Vi föreslår även förbättringar i hårdvaran som hämtar data på förhand, där vi har kvantifierat förbättringarna till upp till 14% för första-nivå cachen.

Genom att anpassa skräpsamlingen till användningen av data i programmet har vi har visat på upp till tre gångers förbättrad prestande. Vi föreslår användningen av flera små exekveringsprocesser i Javamotorn som kör programmet, istället för en stor då vi visat att det kan ge upp till 36% uppsnabbning.

To My Family

Acknowledgements

I am grateful to Allah Almighty for His countless blessings. I would like to thank my supervisors Mats Brorsson, Vladimir Vlassov at KTH, Sweden and Eduard Ayguade at UPC/BSC, Spain for their guidance, feedback and encouragement. I am indebted to my fellow PhD students; Roberto Castañeda Lozano for providing the thesis template, Georgios Varisteas, Artur Podobas and Muhammad Anis-uddin Nasir for discussions on my work and to Ananya Muddukrishna for improving the first draft of my papers. With-out Thomas Sjöland taking care of the financial aspects and Sandra Gustavsson Nylén looking after the administration stuff, it would not have been a smooth journey. I also thank Jim Dowling for reviewing the thesis and acting as the internal quality controller. I am also obliged to David Broman, Seif Haridi and Sverker Jansson for their feedback on my progress.

Contents

Part I

Overview

Chapter 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 that modern scale-up servers are adequate to run analytics jobs [7]. 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 [97], Ostrich [15] and Polymer [101] 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 [98] 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. 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 perform an in-depth evaluation of Spark based data analysis workloads on a scale-up server. We discover that work time inflation (the additional CPU time spent by threads in a multi-threaded computation beyond the CPU time required to perform the same work in a sequential computation) and load imbalance on the threads are the scalability bottlenecks. We quantify the impact of micro-architecture on the performance, and observe that DRAM latency is the major bottleneck.

- We evaluate the impact of data volume on the performance of Spark based data analytics running on a scale-up server. We find the limitations of using Spark on a scale-up server with large volumes of data. We quantify the variations in micro-architectural performance of applications across different data volumes.

- 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 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 remainder of thesis is organized as follows. Chapter 2 discusses background information and related work. Chapter 3 provides a summary of publications. Conclusions and future work are presented in Chapter 4 and Papers A, B and C are attached as supplements in Part II.

Chapter 2

Background and Related Work

Scaling is the ability of the system to adapt to increased demands in terms of data processing. To support big data processing, different platforms incorporate scaling in different forms. From a broader perspective, the big data platforms can be categorized into the two types of scaling: 1) Horizontal scaling or Scale-out means distributing the data and workload across many commodity machines in order to improve the processing capability and 2) Vertical scaling or Scale-up includes assembling machines with more processors, more memory and specialized hardware like GPUs as co-processors [75].

2.1 Horizontally Scaled Systems

MapReduce [22] has become a popular programming framework for big data analytics. It was originally proposed by Google for simplified parallel programming on a large number of machines. A plethora of research exists on improving the performance of big data analytics using MapReduce [25, 50, 76]. Sakr et al. [76] provide a comprehensive survey for a family of approaches and mechanisms of large-scale data processing mechanisms that have been implemented based on the original idea of the MapReduce framework and are currently gaining a lot of momentum in both research and industrial communities. Doulkeridis et al. [25] review a set of the most significant weaknesses and limitations of MapReduce at a high level, along with solving techniques. A taxonomy is presented for categorizing existing research on MapReduce improvements according to the specific problem they target. Based on the proposed taxonomy, a classification of existing research is provided focusing on the optimization objective. The state-of-art on stream and large scale graph processing can be found in [34] and [9] respectively. Spark [98] provides a unified framework for batch and stream processing [99]. Graph processing [94], predictive analtyics using machine learning approaches [59] and SQL query analysis [95] is also supported in Spark.

Spark

Spark is a cluster computing framework that uses Resilient Distributed Datasets (RDDs) [98] 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.

Spark Streaming

Spark Streaming [99] 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.

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.

2.2 Vertically Scaled Systems

MapReduce has been extended to different architectures to facilitate parallel programming, such as multi-core CPUs [7,15,47,48,65,73,74,81,82,84,97,101], GPUs [27, 28, 35, 38, 71], the coupled CPU-GPU architecture [14, 46], FPGA [24, 42, 78], Xeon Phi co-processor [56, 57] and Cell processors [21].

2.3 GPU based Heterogeneous Clusters

Shirahata et al. [80] propose a hybrid scheduling technique for GPU-based computer clusters, which minimizes the execution time of a submitted job using dynamic profiles of Map tasks running on CPU cores and GPU devices. They extend Hadoop to invoke CUDA codes in order to run map tasks on GPU devices. Herrero [33] addresses the problem of integrating GPUs into existing MapReduce framework (Hadoop). OpenCL with Hadoop has been proposed in [66, 93] for the same problem. Zhai et al. [100] provide an annotation based approach to automatically generate CUDA codes from Hadoop codes to hide the complexity of programming on CPU/GPU cluster. To achieve Hadoop and GPU integration, four approaches including Jcuda, JNI, Hadoop Streaming, and Hadoop Pipes, have been accomplished in [103].

El-Helw et al. [26] present Glasswing, a MapReduce framework that uses OpenCL to exploit multi-core CPUs and accelerators. The core of Glasswing is a 5-stage pipeline that overlaps computation, communication between cluster nodes, memory transfers to compute devices, and disk access in a coarse grained manner. Glasswing uses fine-grained parallelism within each node to target modern multi-core and many-core processors. It exploits OpenCL to execute tasks on different types of compute devices without sacrificing the MapReduce abstraction. Additionally, it is capable of controlling task granularity to adapt to the diverse needs of each distinct compute device.

Stuart et al. [83] propose standalone MapReduce library written in C++ and CUDA for GPU clusters. Xie et al. propose Moim [92] which 1) effectively utilizes both CPUs and GPUs, 2) overlaps CPU and GPU computations, 3) enhances load balancing in the map and reduce phases, and 4) efficiently handles not only fixed but also variable size data. Guo et al. [31] present a new approach to design the MapReduce framework on GPU clusters for handling large-scale data processing. They use CUDA and MPI parallel programming models to implement this framework. To derive an efficient mapping onto GPU clusters, they introduce a two-level parallelization approach: the inter node level and intra node level parallelization. Furthermore in order to improve the overall MapReduce efficiency, a multi-threading scheme is used to overlap the communication and computation on a multi-GPU node. An optimized MapReduce framework has been presented for CPU-MIC heterogeneous cluster [87, 88].

Shirahata et al. [79] argue that the capacity of device memory on GPUs limits the size of graph to process and they propose a MapReduce-based out-of-core GPU memory management technique for processing large-scale graph applications on heterogeneous GPU-based supercomputers. The proposed technique automatically handles memory overflows from GPUs by dynamically dividing graph data into multiple chunks and overlaps CPU-GPU data transfer and computation on GPUs as much as possible.

Choi et al. [18] presents Vispark, an extension of Spark for GPU-accelerated MapReduce processing on array-based scientific computing and image processing tasks. Vispark provides an easy-to-use, Python-like high-level language syntax and a novel data abstraction for MapReduce programming on a GPU cluster system. Vispark introduces a programming abstraction for accessing neighbor data in the mapper function, which greatly simplifies many image processing tasks using MapReduce by reducing memory footprints and bypassing the reduce stage.

2.4 FPGA based Heterogeneous Clusters

Choi et al. [19] present MapReduce style implementation of the k-means clustering algorithm on an FPGA-accelerated computer cluster and study system-level tradeoff between computation and I/O performance in the target multi-FPGA execution environment. Neshatpour et al. [62–64] analyze how offloading computationally intensive kernels of machine learning algorithms to a heterogeneous CPU+FPGA platform enhances the performance. They use the latest Xilinx Zynq boards for implementation and result analysis and perform a comprehensive analysis of communication and computation overheads such as data I/O movements, and calling several standard libraries that can not be offloaded to the accelerator to understand how the speedup of each will contribute to an application's overall execution in an end-to-end Hadoop MapReduce environment.

2.5 In-Storage Processing

Choi et al. [16] propose scale-in clusters with in-storage processing devices to reduce data movements towards CPUs. Scale-in clusters with ISP can improve the overall energy efficiency of similarly performing scale-out clusters upto 5.5x according to model based evaluation. They show that memory and storage bandwidths are the main bottlenecks in clusters with commodity servers. By replacing SATA-HD with PCIe-SSD, 23x performance improvement can be achieved. Scale-out clusters introduce high data-movement energy consumption as cluster size increases. Further energy ratio (data movement energy / consumption energy per byte) increases in scale-out clusters comprising thousands of nodes with process technology scaling. At 7nm, data movement energy consumption takes around 85% of total energy consumption while computation energy account for only 15%. They also present a short survey on In-Storage processing. Moreover they evaluate performance improvements of different configurations of storing persisted RDDs and shuffling data between memory and high performance SSDs and find that performance can be improved 23% on average by utilizing high performance SSDs to store persisted RDDs along with shuffle data compared to memory only approach [17].

2.6 Processing in DRAM 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. Gabriel et al. [54] in their position paper presented an initial taxonomy for in-memory computing. There exists a continuum of compute capabilities that can be embedded "in memory". This includes:

- Software transparent applications of logic in memory.
- Pre-defined or fixed functions accelerators.
- Bounded-operand PIM operations (BPO), which can be specified in a manner that is consistent with existing instruction-level memory operand formats. Simple extensions to this format could encode the PIM operation directly in the opcode, or perhaps as a special prefix in the case of the x86-64 ISA, but no additional fields are required to specify the memory operands
- Compound PIM operations (CPOs), which may access an arbitrary number of memory locations (not-specifically pre-defined) and perform a number of different operations. Some examples include data movement operations such as scatter/gather, list reversal, matrix transpose, and in-memory sorting.
- Fully-programmable logic in memory, which provide the expressiveness and flexibility of a conventional processor (or configurable logic device), along with all of the associated overheads except off-chip data migration.

Kersy et al. [45] present FPGA-based prototype in order to evaluate the impact of SIMT (single instruction multiple thread) based logic layers in 3D stacked DRAM architecture, due to their ability to take advantage of high memory bandwidth and memory level parallelism. In SIMT, multiple threads are in flight simultaneously, threads in the same wrap execute at the same program counter. Since there are many warps and many threads per warp, the demand for memory bandwidth is quite large, they have high tolerance to memory system latency, reducing their dependence on caches and allowing them in case of stacked DRAM systems to be connected directly to DRAM interface. These processors are well suited to intrinsically parallel tasks like traversing data structures, e.g. in data analytics applications in which large irregular data structures must be traversed many times, with little reuse during each traversal, limiting the effectiveness of caches.

PIM for Simple MapReduce Applications

Pugsley et al. [69] propose near data computing (NDC) architecture in which a central host processor with many energy efficient cores is connected to many daisy chained 3D-stacked memory devices with simple cores in their logic layer; these cores can perform Map operations with efficient data access and without hitting the memory bandwidth wall. Reduce operations, however are executed on the central host processor because it requires random access to data. For random access, average hop count is minimized if requests originate in the central location i.e. host processor. They also show that their proposed design can reduce power by disabling expensive SerDes circuits on the memory device and by powering down the cores that are inactive in each phase. Compared to a baseline that is heavily optimized for MapReduce execution, the NDC yields upto 15x reduction in execution time and 18x reduction in system energy. Islam et al. [36] propose a similar PIM architecture with a difference that they do not assume the entire input for computation to reside in memory and consider conventional storage systems as the source of input. Their calculations show logic layer can accommodate 26 ARM like cores without crossing the power budget of 10W [77].

PIM for Graph Analytics

Ahn et al. [3] find that high memory bandwidth is the key to the scalability of graph processing and conventional systems do not fully utilize high memory bandwidth. They propose PIM architecture based on 3D-stacked DRAM, where specialized in-order cores with graph processing specific prefetchers are used. Moreover the programming model employed is also latency tolerant. Nai et al. [61] show that graph traversals, bounded by irregular memory access patterns of graph property, can be accelerated by offloading the graph property to hybrid memory cube (HMC) by utilizing the atomic requests described in HMC 2.0 specification (that is limited to only integer operations and one-memory operand). Atomic requests (arithmetic, bitwise, boolean, comparison) include three steps, reading 16 bytes of data from DRAM, performing one operation on the data and then writing back the result to the same DRAM location. Their calculations based on analytical model for off-chip bandwidth show instruction offloading method can save the memory bandwidth by 67% and can also remove the latency of redundant cache look-ups

PIM for Machine Learning Workloads

Lee [49] use State Synchronous Parallel (SSP) model to evaluate asynchronous parallel machine learning workloads and observe that atomic operations occupy a large portion of overall execution time. Their proposal called BSSync is based on two ideas regarding the iterative convergent algorithms, 1) atomic update stage is separate from the main computation and it can be overlapped with the main computation 2) atomic operations are a limited, predefined set of operations that do not require the flexibility of general purpose core. They propose to offload atomic operations onto logic layers in 3D stacked memories. Atomic operations are overlapped with main computation that increases the execution efficiency. Through cycle accurate simulations on Zsim of iterative convergent ML workloads, their proposal outperforms the asynchronous parallel implementation by 1.33x.

Bender [11] use a variant of k-means algorithm in which traditional DRAM is analogous to disk and near-memory is analogous to traditional DRAM. Near-memory is physically bonded to a package containing processing elements rather remotely available via bus. The benefit is much higher bandwidth compared to traditional DRAM, with similar latency. Such architecture is available in Knight's Landing processor from Intel. Using theoretical analysis, they predict 30% speedup. Mudo et al. [23] propose content addressable memories (address the data based on the query vector content) with hamming distance computing units (XOR operators) in the logic layer to minimize the impact of significant data movement in k-nearest neighbours and estimate an order of magnitude performance improvement over the best off-the-shelf software libraries, however the study lacks experimentation results and presents only the architecture.

PIM for SQL Query Analysis Workloads

Mirzadeh et al. [60] studies Join workload, which is characterized by irregular access pattern, on multiple HMC like 3D stacked DRAM devices connected together via SerDes links. The architecture is chosen because CPU-HMC interface consumes twice as much energy as accessing the DRAM itself and also due to capacity to each HMC constrained to 8GB. They argue that the design of near memory processing (NMP) algorithms should consider data placement and communication cost and should exploit locality with in one stack as much as possible because a memory access may require traversing multiple SerDes links to reach the appropriate HMC target and because SerDEs link traversal is more expensive that the actual DRAM access. Moreover they suggest that the design should minimize the number of fine-grain(single word) accesses to stacked DRAM since the DRAM access has a wide interface in comparison to a cache access and the access is destructive i.e. even when the single word of a DRAM row is accessed, the whole row must be pre-charged in row buffer and then written back to DRAM. In NMP architecture, join algorithms execute on the logic layer of HMC. The logic layer of HMC is modelled as a simple micro-controller that supports 256B SIMD, bitonic merge sort and 2D mesh NoC to support data movement within a chip. Evaluation is based on first order analytical model. Xi et al. [91] present JAFAR, a Near-Data Processing (NDP) accelerator for pushing selects down to memory in modern column-stores.Thus only relevant data will still be pushed up the memory hierarchy, causing a significant reduction in data movement.

PIM for Data Reorganization Operations

Akin et al. [5] focus on common data reorganization operations such as shuffle, pack/unpack, swap, transpose, and layout transformations. Although these operations simply relocate the data in the memory, they are costly on conventional systems mainly due to inefficient access patterns, limited data reuse and round-trip data traversal throughout the memory hierarchy. They have proposed DRAMaware reshape accelerator integrated within 3D-stacked DRAM, and a mathematical framework that is used to represent and optimize the reorganization operations.

Gokhale et al. [30] argue that applications that manipulate complex, linked data structures benefit much less from the deep cache hierarchy and experience high latency due to random access and cache pollution when only a small portion of a cache line is used. They design a system to benefit data intensive applications with access patterns that have little spatial or temporal locality. Examples include switching between row-wise and column-wise access to arrays, sparse matrix operations and pointer traversal. Using strided DMA units, gather/scatter hardware and in-memory scratchpad buffers, the programmable near memory data rearrangement engines perform fill and drain operations to gather the blocks of application data structures. The goal is to accelerate data access, making it possible for many CPU cores to compute on complex data structures efficiently packed into the cache. Using custom FPGA emulator, they evaluate the performance of near-memory hardware structures that dynamically restructure in-memory data to cache friendly layout.

2.7 Processing in Nonvolatile Memory

Ranganathan et al. [72] propose nano-stores that co-locates processors and NVM 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. They also develop and validate a new methodology to evaluate such system architectures for large scale data centric workloads. 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. [89] 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.

2.8 Processing in Hybrid 3D-Stacked DRAM and NVRAM

Huang et al. [70] propose a 3D hybrid storage structure that tightly integrates CPU, DRAM and Flash based NVRAM to meet the memory needs of big data applications with larger capacity, smaller delay and wider bandwidth. Similar to scale-out processors' pod [55], DRAM and NVM layers are divided into multiple zones, corresponding to their core sets. Through multiple high speed TSV's connecting compute and storage resources, the localization of computing and storage resources are achieved which results in performance improvement.

2.9 Interoperability of PIM with Cache and Virtual Memory

Challenges of PIM architecture design are cost-effective integration of logic and memory, unconventional programming models and lack of interoperability with caches and virtual memory. Ahn et al. [4] propose PIM-enabled instruction, a low-cost PIM abstraction HW. It interfaces PIM operations as ISA extension which simplifies cache coherence and virtual memory support for PIM. Another advantage is locality-aware execution of PIM operations. Evaluations show good adaptivity across randomly generated workloads

2.10 Profiling Bigdata Platforms

Oliver et al. [67] have shown that task parallel applications can exhibit poor performance due to work time inflation. We see similar phenomena in Spark based workloads. Ousterhout et al. [68] have developed blocked time analysis to quantify performance bottlenecks in the Spark framework and found out that CPU (and not I/O) is often the bottleneck. Our thread level analysis of executor pool threads also reveal that CPU time (and not wait time) is the dominant performance bottleneck in Spark based workloads.

Several studies characterize the behaviour of big data workloads and identify the mismatch between the processor and the big data applications [29, 39–41, 44, 86, 96]. However these studies lack in identifying the limitations of modern scale-up servers for Spark based data analytics. Ferdman et al. [29] 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. [102] 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 [39]. They also suffer from notable from end stalls but L2 and L3 caches are effective for them. Wang et al. [86] conclude the same about L3 caches and L1 I Cache miss rates despite using larger data sets. Deep dive analysis [96] 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.

Jiang et al. [41] observe that memory access characteristics of the Spark and Hadoop workloads differ. At the micro-architecture level, they have roughly same behaviour and point current micro-architecture works for Spark workloads. Contrary to that, Jia et al. [40] conclude that Software stacks have significant impact on the micro-architecture behaviour of big data workloads.

Tang et al. [85] have shown that NUMA has significant impact on Gmail backend and web.search frontend. Beamer et al. [10] have shown NUMA has moderate performance penalty and SMT has limited potential for graph analytics running on Ivy bridge server. Kanev et al. [43] have argued in favour of SMT after profiling live data center jobs on 20,000 google machines. Our work extends extends extends the literature by profiling Spark jobs. Researchers at IBM's Spark technology center [2] have also shown moderate performance gain from NUMA process affinity. Our work gives micro-architectural reasons for this moderate performance gain.

Ruirui et al. [58] have compared throughput, latency, data reception capability and performance penalty under a node failure of Apache Spark with Apache Storm. Miyuru et al. [20] have compared the performance of five streaming applications on System S and S4. Jagmon et al. [13] have analyzed performance of S4 in terms of scalability, lost events, resource usage and fault tolerance. Our work analyzes the micro-architectural performance of Spark Streaming.

2.11 Project Tungsten

The inventors of Spark have a road map for optimizing the single node performance of Spark under the project name Tungsten [1]. Its goal is to improve the memory and CPU efficiency of the Spark applications by a) memory management and binary processing; leverage application semantics to manage memory explicitly and eliminate the overhead of JVM object model garbage collection, b) Cache-aware computation: algorithms and data structures to exploit memory hierarchy, c) exploit modern compilers and CPUs; allow efficient operation directly on binary data.

Java object-based row representation has high space overhead. Tungsten gives new Unsafe Row format where rows are always 8-byte word aligned (size is multiple of 8 bytes). Equality comparison and hashing can be performed on raw bytes without additional interpretation. Sun.misc.Unsafe exposes C style memory access e.g. explicit allocation, deallocation and pointer arithmetic. Furthermore Unsafe methods are intrinsic, meaning each method call is compiled by JIT into a single machine instruction. Most distributed data processing can be boiled down to a small list of operations, such as aggregations, sorting, and join. By improving the efficiency of these operations, the efficiency of Spark applications can be improved as a whole.

2.12 New Server Architectures

Recent research shows that the architectures of current servers do not comply well the computational requirements of big data processing applications. Therefore, it is required to look for a new architecture for servers as a replacement for currently used machines for both performance and energy enhancement. Using low-power processors (microservers), more system- level integration and a new architecture for server processors are some of the solutions that have been discussed recently as performance/energy- efficient replacement for current machines.

Microservers for Big Data Analytics

Prior research shows that the processors based on simple in-order cores are well suited for certain scale-out workloads [52]. A 3000-node cluster simulation driven by a real-world trace from Facebook shows that on average a cluster comprising ARM-based microservers which support the Hadoop platform reaches the same performance of standard servers while saving energy up to 31% at only 60% of the acquisition cost. Recently, ARM big.LITTLE boards (as small nodes) have been introduced as a platform for big data processing [53]. In comparison with Intel Xeon server systems (as traditional big nodes), the I/O-intensive MapReduce workloads are more energy-efficient to run on Xeon nodes. In contrast, database query processing is always more energy-efficient on ARM servers, at the cost of slightly lower throughput. With minor software modifications, CPU-intensive MapReduce workloads are almost four times cheaper to execute on ARM servers. Unfortunately, small memory size, low memory, and I/O bandwidths, and software immaturity ruins the lower power advantages obtained by ARM servers.

Novel Server Processors

Due to the large mismatch between the demands of the scale-out workloads and today's processor micro-architecture, scale-out processors have been recently introduced that can result in more area- and energy-efficient servers in future [29,32,55]. The building block of a scale-out processor is the pod. A pod is a complete server that runs its copy of the OS. A pod acts as the tiling unit in a scale-out processor, and multiple pods can be placed on a die. A scale-out chip is a simple composition of one or more pods and a set of memory and I/O interfaces. Each pod couples a small last-level cache to a number of cores using a low-latency interconnect. Having a higher per-core performance and lower energy per operation leads to better energy efficiency in scale-out processors. Due to smaller caches and smaller communication distances, scale-out processors dissipate less energy in the memory hierarchy [55]. FAWN architecture [6] is another solution for building cluster systems for energy-efficient serving massive-scale I/O and data-intensive workloads. FAWN couples low-power and efficient embedded processors with flash storage to provide fast and energy-efficient processing of random read-intensive workloads.

System-Level Integration (Server-on-Chip)

System-level integration is an alternative approach that has been proposed for improving the efficiency of the warehouse-scale data-center server market. System-level integration discusses placing CPUs and components on the same die for servers, as done for embedded systems. Integration reduces the (1) latency: by placing cores and components closer to one another, (2) cost: by reducing parts in the bill of material, and (3) power: by decreasing the number of chip-to-chip pin-crossings.

Initial results show a reduction of more than 23% of capital cost and 35% of power costs at 16 nm [51].

Chapter 3

Summary of Publications

We present a summary of publications part of the thesis in the chapter.

3.1 List of Publications

- **Paper A:** [**37**] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Performance Characterization of In-Memory Data Analytics on a Modern Cloud server", in 5th IEEE International Conference on Big Data and Cloud Computing (BDCloud), Dalian, China, 2015. (Best Paper Award)
- **Paper B:** [**8**] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "How Data Volume Affects Spark Based Data Analytics on a Scale-up Server" in 6th International Workshop on Big data Benchmarks, Performance Optimization and Emerging Hardware (BpoE) held in conjunction with 41st International Conference on Very Large Data Bases (VLDB), Hawaii, USA, 2015.
- **Paper C:** A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Architectural Impact on Performance of In-memory Data Analytics: Apache Spark Case Study" (submitted)

3.2 Individual Contribution of Authors

- **Ahsan Javed Awan:** contributes to, literature review, problems identification, hypothesis formulation, experiment design, data analysis and paper writing.
- **Mats Brorsson:** contributes to, problem selection and feed back on experiment design, results, conclusions and draft of paper.
- **Vladimir Vlassov:** contributes to, problem selection and feed back on draft of paper.

- **Eduard Ayguade:** contributes to, problem selection and feed back on experiment design, results, conclusions and draft of paper.

3.3 Summary of Paper A

In order to ensure effective utilization of scale-up servers, it is imperative to make a workload-driven study on the requirements that big data analytics put on processor and memory architectures. Existing studies lack in quantifying the impact of processor inefficiencies on the performance of in memory data analytics, which is impediment to propose novel hardware designs to increase the efficiency of modern servers for in-memory data analytics. To fill in this, we characterize the performance of in-memory data analytics using Apache Spark framework. We use a single node NUMA machine and identify the bottlenecks hampering the multi-core scalability of workloads. We also quantify the inefficiencies at micro-architecture level for various data analysis workloads.

The key insights are:

- More than 12 threads in an executor pool does not yield significant performance.
- Work time inflation and load imbalance on the threads are the scalability bottlenecks.
- Removing the bottlenecks in the front-end of the processor would not remove more than 20% of stalls.
- Effort should be focused on removing the memory bound stalls since they account for up to 72% of stalls in the pipeline slots.
- Memory bandwidth of current processors is sufficient for in- memory data analytics

3.4 Summary of Paper B

This paper augments Paper A by quantifying the impact of data volume on the performance of in-memory data analytics with Spark on scale-up servers. In this paper, we answer the following questions concerning Spark based data analytics running on modern scale-up servers:

- Do Spark based data analytics benefit from using larger scale-up servers?
- How severe is the impact of garbage collection on performance of Spark based data analytics?
- Removing the bottlenecks in the front-end of the processor would not remove more than 20% of stalls.

- Is file I/O detrimental to Spark based data analytics performance?
- How does data size affect the micro-architecture performance of Spark based data analytics?

The key insights are:

- Spark workloads do not benefit significantly from executors with more than 12 cores.
- The performance of Spark workloads degrades with large volumes of data due to substantial increase in garbage collection and file I/O time.
- With out any tuning, Parallel Scavenge garbage collection scheme outperforms Concurrent Mark Sweep and G1 garbage collectors for Spark workloads.
- Spark workloads exhibit improved instruction retirement due to lower L1 cache misses and better utilization of functional units inside cores at large volumes of data.
- Memory bandwidth utilization of Spark benchmarks decreases with large volumes of data and is 3x lower than the available off-chip bandwidth on our test machine.

3.5 Summary of Paper C

The scope of previous two papers 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 inmemory 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?

The key insights are:

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

Chapter 4

Conclusion and Future Work

Firstly we find that performance bottlenecks in Spark workloads on a scale-up server are frequent data accesses to DRAM, thread level load imbalance, garbage collection overhead and wait time on file I/O. To improve the performance of Spark workloads on a scale-up server, we make following recommendation: (i) Spark users should prefer DataFrames over RDDs while developing Spark applications and input data rates should be large enough for real time streaming analytics to exhibit better instruction retirement, (ii) Spark should be configured to use executors with memory size less than or equal to 32GB and restrict each executor to use NUMA local memory, (iii) GC scheme should be matched to the workload, (iv) Hyper-threading should be turned on, next line L1-D and adjacent cache line L2 prefetchers should be turned off and DDR3 speed should be configured to 1333.

Secondly, we 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. Since DRAM scaling is not picking up with the moore's law, increasing DRAM capacity will be a challenge. NVRAM on the other hand shows 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 into a hybrid architecture where the host is connected to DRAM with custom accelerators and flash based NVM with integrated hardware units to reduce the data movement. Figure 4.1 shows the architecture.

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 organise the data and a set of reduce tasks to aggregate it. Internally results are kept in memory until they can't fit. Then these are sorted based on the target partition and written to a single file. On the reduce side, tasks read the relevant sorted blocks. It is worth while to investigate

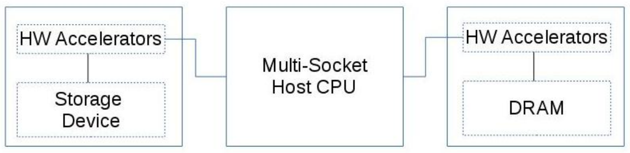


Figure 4.1: NDC Supported Single Node in Scale-in Clusters for in-Memory Data Analytics with Spark

the hardware software co-design of shuffle for near data computing architectures.

Real time analytics are enabled through large scale distributed stream processing frameworks like D-streams in Apache Spark. Existing literature lacks the understanding of Distributed streaming applications from the architectural perspective. PIM architecture for such applications is worth looking at. PIM accelerators for data base operations like Aggregations, Projections, Joins, Sorting, Indexing and Compression can be researched further. Q100 [90] like data processing units in DRAM can be used to accelerate SQL queries.

In a conventional MapReduce system, it is possible to carefully data across vaults in an NDC system to ensure good map phase locality and high performance but with iterative MapReduce, it is impossible to predict how RDDs will be produced and how well behaved they will be. It might be beneficial to migrate data between nodes between one Reduce and the next Map Phase and to even use a hardware accelerator to decide which data should end up where.

Bibliography

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

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

[3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 105–117. ACM, 2015.

[4] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 336–348. ACM, 2015.

[5] Berkin Akin, Franz Franchetti, and James C Hoe. Data reorganization in memory using 3d-stacked dram. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 131–143. ACM, 2015.

[6] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14. ACM, 2009.

[7] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony I. T. Rowstron. Scale-up vs scale-out for hadoop: time to rethink? In *ACM Symposium on Cloud Computing, SOCC*, page 20, 2013.

[8] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. *Big Data Benchmarks, Performance Optimization, and Emerging Hardware: 6th Workshop, BPOE 2015, Kohala, HI, USA, August 31 - September 4, 2015. Revised Selected Papers*, chapter How Data Volume Affects Spark Based Data Analytics on a Scale-up Server, pages 81–92. Springer International Publishing, 2016.

[9] Omar Batarfi, Radwa El Shawi, Ayman G Fayoumi, Reza Nouri, Ahmed Barnawi, Sherif Sakr, *et al.* Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.

[10] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 56–65. IEEE, 2015.

[11] Michael A Bender, Jonathan Berry, Simon D Hammond, Branden Moore, Benjamin Moseley, and Cynthia A Phillips. k-means clustering on two-level memory systems. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 197–205. ACM, 2015.

[12] Jichuan Chang, Parthasarathy Ranganathan, Trevor Mudge, David Roberts, Mehul A Shah, and Kevin T Lim. A limits study of benefits from nanostore-based future data-centric system architectures. In *Proceedings of the 9th conference on Computing Frontiers*, pages 33–42. ACM, 2012.

[13] Jagmohan Chauhan, Shaiful Alam Chowdhury, and Dwight Makaroff. Performance evaluation of yahoo! s4: A first look. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012 Seventh International Conference on*, pages 58–65. IEEE, 2012.

[14] Linchuan Chen, Xin Huo, and Gagan Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 25. IEEE Computer Society Press, 2012.

[15] Rong Chen, Haibo Chen, and Binyu 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*, PACT '10, pages 523–534, 2010.

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

[17] I Stephen Choi, Weiqing Yang, and Yang-Suk Kee. 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*, pages 1073–1083. IEEE, 2015.

[18] Woohyuk Choi and Won-Ki Jeong. Vispark: Gpu-accelerated distributed visual computing using spark. In *Large Data Analysis and Visualization (LDAV), 2015 IEEE 5th Symposium on*, pages 125–126. IEEE, 2015.

[19] Yuk-Ming Choi and Hayden Kwok-Hay So. Map-reduce processing of k-means algorithm with fpga-accelerated computer cluster. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pages 9–16. IEEE, 2014.

[20] Miyuru Dayarathna and Toyotaro Suzumura. A performance analysis of system s, s4, and esper via two level benchmarking. In *Quantitative Evaluation of Systems*, pages 225–240. Springer, 2013.

[21] Marc De Kruijf and Karthikeyan Sankaralingam. Mapreduce for the cell broadband engine architecture. *IBM Journal of Research and Development*, 53(5):10–1, 2009.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] Carlo C del Mundo, Vincent T Lee, Luis Ceze, and Mark Oskin. Ncam: Near-data processing for nearest neighbor search. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 274–275. ACM, 2015.

[24] Dionysios Diamantopoulos and Christoforos Kachris. High-level synthesizable dataflow mapreduce accelerator for fpga-coupled data centers. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pages 26–33. IEEE, 2015.

[25] Christos Doulkeridis and Kjetil Nørvåg. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, 23(3):355–380, 2014.

[26] Ismail El-Helw, Rutger Hofman, and Henri E Bal. Scaling mapreduce vertically and horizontally. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 525–535. IEEE, 2014.

[27] Marwa Elteir, Heshan Lin, Wu-chun Feng, and Tom Scogland. Streammr: an optimized mapreduce framework for amd gpus. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 364–371. IEEE, 2011.

[28] Wenbin Fang, Bingsheng He, Qiong Luo, and Naga K Govindaraju. Mars: Accelerating mapreduce with graphics processors. *Parallel and Distributed Systems, IEEE Transactions on*, 22(4):608–620, 2011.

[29] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak 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*, ASPLOS XVII, pages 37–48, 2012.

[30] Maya Gokhale, Scott Lloyd, and Chris Hajas. Near memory data structure rearrangement. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 283–290. ACM, 2015.

[31] Yiru Guo, Weiguo Liu, Bo Gong, Gerrit Voss, and Wolfgang Muller-Wittig. Gcmr: A gpu cluster-based mapreduce framework for large-scale data processing. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 580–586. IEEE, 2013.

[32] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*, number DIAS-CONF-2007-008, 2007.

[33] Sergio Herrero-Lopez. Accelerating svms by integrating gpus into mapreduce clusters. In *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*, pages 1298–1305. IEEE, 2011.

[34] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[35] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. Mapcg: writing parallel program portable between cpu and gpu. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 217–226. ACM, 2010.

[36] Mahzabeen Islam, Marko Scrbak, Krishna M Kavi, Mike Ignatowski, and Nuwan Jayasena. Improving node-level mapreduce performance using processing-in-memory technologies. In *Euro-Par 2014: Parallel Processing Workshops*, pages 425–437. Springer, 2014.

[37] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard 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*, pages 1–8. IEEE, 2015.

[38] Feng Ji and Xiaosong Ma. Using shared memory to accelerate mapreduce on graphics processing units. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 805–816. IEEE, 2011.

[39] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. Characterizing data analysis workloads in data centers. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 66–76, 2013.

[40] Zhen Jia, Jianfeng Zhan, Lei Wang, Rui Han, Sally A. McKee, Qiang Yang, Chunjie Luo, and Jingwei Li. Characterizing and subsetting big data workloads. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 191–201, 2014.

[41] Tao Jiang, Qianlong Zhang, Rui Hou, Lin Chai, Sally A. McKee, Zhen Jia, and Ninghui Sun. Understanding the behavior of in-memory computing workloads. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 22–30, 2014.

[42] Christoforos Kachris, Georgios Ch Sirakoulis, and Dimitrios Soudris. A reconfigurable mapreduce accelerator for multi-core all-programmable socs. In *ISSoC*, pages 1–6, 2014.

[43] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, David Brooks, Simone Campanoni, Kevin Brownell, Timothy M Jones, *et al.* Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 158–169. ACM, 2015.

[44] Vasileios Karakostas, Osman S. Unsal, Mario Nemirovsky, Adrian Cristal, and Michael Swift. Performance analysis of the memory management unit under scale-out workloads. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 1–12, Oct 2014.

[45] Chad D Kersey, Sudhakar Yalamanchili, and Hyesoon Kim. Simt-based logic layers for stacked dram architectures: A prototype. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 29–30. ACM, 2015.

[46] SungYe Kim, Jeremy Bottleson, Jingyi Jin, Preeti Bindu, Snehal C Sakhare, and Joseph S Spisak. Power efficient mapreduce workload acceleration using integrated-gpu. In *Big Data Computing Service and Applications (BigDataService), 2015 IEEE First International Conference on*, pages 162–169. IEEE, 2015.

[47] K. Ashwin Kumar, Jonathan Gluck, Amol Deshpande, and Jimmy Lin. Hone: "scaling down" hadoop on shared-memory systems. *Proc. VLDB Endow.*, 6 (12):1354–1357, August 2013.

[48] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[49] Joo Hwan Lee, Jaewoong Sim, and Hyesoon Kim. Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models.

[50] Ren Li, Haibo Hu, Heng Li, Yunsong Wu, and Jianxi Yang. Mapreduce parallel programming model: A state-of-the-art survey. *International Journal of Parallel Programming*, pages 1–35, 2015.

[51] Sheng Li, Kevin Lim, Paolo Faraboschi, Jichuan Chang, Parthasarathy Ranganathan, and Norman P Jouppi. System-level integrated server architectures for scale-out datacenters. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 260–271. ACM, 2011.

[52] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 315–326. IEEE, 2008.

[53] Dumitrel Loghin, Bogdan Marius Tudor, Hao Zhang, Beng Chin Ooi, and Yong Meng Teo. A performance study of big data on small nodes. *Proceedings of the VLDB Endowment*, 8(7):762–773, 2015.

[54] GH Loh, N Jayasena, M Oskin, M Nutter, D Roberts, M Meswani, DP Zhang, and M Ignatowski. A processing in memory taxonomy and a case for studying fixed-function pim. In *Workshop on Near-Data Processing (WoNDP)*, 2013.

[55] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, *et al.* Scale-out processors. *ACM SIGARCH Computer Architecture News*, 40(3):500–511, 2012.

[56] Mian Lu, Yun Liang, Huynh Phung Huynh, Zhongliang Ong, Bingsheng He, and Rick Siow Mong Goh. Mrphi: An optimized mapreduce framework on intel xeon phi coprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 26(11):3066–3078, 2015.

[57] Mian Lu, Lei Zhang, Huynh Phung Huynh, Zhongliang Ong, Yun Liang, Bingsheng He, Rick Siow Mong Goh, and Richard Huynh. Optimizing the mapreduce framework on intel xeon phi coprocessor. In *Big Data, 2013 IEEE International Conference on*, pages 125–130. IEEE, 2013.

[58] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 69–78. IEEE, 2014.

[59] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, *et al.* Mllib: Machine learning in apache spark. *arXiv preprint arXiv:1505.06807*, 2015.

[60] Nooshin Mirzadeh, Yusuf Onur Koçberber, Babak Falsafi, and Boris Grot. Sort vs. hash join revisited for near-memory execution. In *5th Workshop on Architectures and Systems for Big Data (ASBD 2015)*, number EPFL-CONF-209121, 2015.

[61] Lifeng Nai and Hyesoon Kim. Instruction offloading with hmc 2.0 standard: A case study for graph traversals. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 258–261. ACM, 2015.

[62] Katayoun Neshatpour, Maria Malik, Mohammad Ali Ghodrat, and Houman Homayoun. Accelerating big data analytics using fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 164–164. IEEE, 2015.

[63] Katayoun Neshatpour, Maria Malik, Mohammad Ali Ghodrat, Avesta Sasan, and Houman Homayoun. Energy-efficient acceleration of big data analytics applications using fpgas. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 115–123. IEEE, 2015.

[64] Katayoun Neshatpour, Maria Malik, and Houman Homayoun. Accelerating machine learning kernel in hadoop using fpgas. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 1151–1154. IEEE, 2015.

[65] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[66] Razvan Nitu, Elena Apostol, and Valentin Cristea. An improved gpu mapreduce framework for data intensive applications. In *Intelligent Computer Communication and Processing (ICCP), 2014 IEEE International Conference on*, pages 355–362. IEEE, 2014.

[67] Stephen L. Olivier, Bronis R. de Supinski, Martin Schulz, and Jan F. Prins. Characterizing and mitigating work time inflation in task parallel programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 65:1–65:12, 2012.

[68] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, 2015.

[69] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Feifei Li, *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*, pages 190–200. IEEE, 2014.

[70] Cheng Qian, Libo Huang, Peng Xie, Nong Xiao, and Zhiying Wang. A study on non-volatile 3d stacked memory for big data applications. In *Algorithms and Architectures for Parallel Processing*, pages 103–118. Springer, 2015.

[71] Zhi Qiao, Shuwen Liang, Hai Jiang, and Song Fu. Mr-graph: a customizable gpu mapreduce. In *Cyber Security and Cloud Computing (CSCloud), 2015 IEEE 2nd International Conference on*, pages 417–422. IEEE, 2015.

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

[73] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24, Feb 2007.

[74] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[75] Michael Saecker and Volker Markl. Big data analytics on modern hardware architectures: A technology survey. In *Business Intelligence*, pages 125–149. Springer, 2013.

[76] Sherif Sakr, Anna Liu, and Ayman G Fayoumi. The family of mapreduce and large-scale data processing systems. *ACM Computing Surveys (CSUR)*, 46 (1):11, 2013.

[77] Marko Scrbak, Mahzabeen Islam, Krishna M Kavi, Mike Ignatowski, and Nuwan Jayasena. Processing-in-memory: Exploring the design space. In *Architecture of Computing Systems–ARCS 2015*, pages 43–54. Springer, 2015.

[78] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. Fpmr: Mapreduce framework on fpga. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 93–102. ACM, 2010.

[79] Koichi Shirahata, Hikaru Sato, and Shingo Matsuoka. Out-of-core gpu memory management for mapreduce-based large-scale graph processing. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 221–229. IEEE, 2014.

[80] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Hybrid map task scheduling for gpu-based heterogeneous clusters. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 733–740. IEEE, 2010.

[81] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, volume 48, pages 135–146. ACM, 2013.

[82] Jeremy Singer, George Kovoor, Gavin Brown, and Mikel Luján. Garbage collection auto-tuning for java mapreduce on multi-cores. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 109–118, 2011. ISBN 978-1-4503-0263-0.

[83] Jeff A Stuart and John D Owens. Multi-gpu mapreduce on gpu clusters. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1068–1079. IEEE, 2011.

[84] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce, pages 9–16, 2011.

[85] Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. Optimizing google's warehouse scale computers: The numa experience. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 188–197. IEEE, 2013.

[86] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. Bigdatabench: A big data benchmark suite from internet services. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA*, pages 488–499, 2014.

[87] Wenzhu Wang, Qingbo Wu, Yusong Tan, and Yaoxue Zhang. An efficient mapreduce framework for intel mic cluster. In *Intelligence Science and Big Data Engineering. Big Data and Machine Learning Techniques*, pages 129–139. Springer, 2015.

[88] Wenzhu Wang, Qingbo Wu, Yusong Tan, and Yaoxue Zhang. Optimizing the mapreduce framework for cpu-mic heterogeneous cluster. In *Advanced Parallel Processing Technologies*, pages 33–44. Springer, 2015.

[89] Ying Wang, Yinhe Han, Lei Zhang, Huawei Li, and Xiaowei Li. Propram: exploiting the transparent logic resources in non-volatile memory for near data computing. In *Proceedings of the 52nd Annual Design Automation Conference*, page 47. ACM, 2015.

[90] Lisa Wu, Andrea Lottarini, Timothy K Paine, Martha A Kim, and Kenneth A Ross. Q100: the architecture and design of a database processing unit. In *ACM SIGPLAN Notices*, volume 49, pages 255–268. ACM, 2014.

[91] Sam Likun Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware (DaMoN)*, 2015.

[92] Mengjun Xie, Kyoung-Don Kang, and Can Basaran. Moim: A multi-gpu mapreduce framework. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pages 1279–1286. IEEE, 2013.

[93] Miao Xin and Hao Li. An implementation of gpu accelerated mapreduce: Using hadoop with opencl for data-and compute-intensive jobs. In *Service Sciences (IJCSS), 2012 International Joint Conference on*, pages 6–11. IEEE, 2012.

[94] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[95] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, pages 13–24. ACM, 2013.

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

[97] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, pages 198–207, 2009.

[98] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion 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)*, pages 15–28, San Jose, CA, 2012. ISBN 978-931971-92-8.

[99] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*, 2012.

[100] Yanlong Zhai, Emmanuel Mbarushimana, Wei Li, Jing Zhang, and Ying Guo. Lit: A high performance massive data computing framework based on cpu/gpu cluster. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[101] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 183–193. ACM, 2015.

[102] Chen Zheng, Jianfeng Zhan, Zhen Jia, and Lixin Zhang. Characterizing os behavior of scale-out data center workloads. In *The SeventhAnnual Workshop on the Interaction amongst Virtualization,Operating Systems and Computer Architecture(WIVOSCA2013) held in conjunction with The 40th International Symposium on Computer Architecture*, 2013.

[103] Jie Zhu, Juanjuan Li, Erikson Hardesty, Hai Jiang, and Kuan-Ching Li. Gpu-in-hadoop: Enabling mapreduce across distributed heterogeneous platforms. In *Computer and Information Science (ICIS), 2014 IEEE/ACIS 13th International Conference on*, pages 321–326. IEEE, 2014.

Part II

Publications

Publication A

Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server (Best Paper Award).
Ahsan Javed Awan, Mats Brorsson, Vladmir Vlassov and Eduard Ayguade.
5th IEEE International Conference on Big Data and Cloud Computing (BDCloud), Dalian, China, 2015.

Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server

Ahsan Javed Awan[1], Mats Brorsson[1], Vladimir Vlassov[1] and Eduard Ayguade[2]

[1]KTH Royal Institute of Technology,
Software and Computer Systems Department(SCS),
{ajawan,matsbror,vladv}@kth.se
[2]Technical University of Catalunya (UPC),
Computer Architecture Department,
eduard@ac.upc.edu

Abstract

In last decade, data analytics have rapidly progressed from traditional disk-based processing to modern in-memory processing. However, little effort has been devoted at enhancing performance at micro-architecture level. This paper characterizes the performance of in-memory data analytics using Apache Spark framework. We use a single node NUMA machine and identify the bottlenecks hampering the scalability of workloads. We also quantify the inefficiencies at micro-architecture level for various data analysis workloads. Through empirical evaluation, we show that spark workloads do not scale linearly beyond twelve threads, due to work time inflation and thread level load imbalance. Further, at the micro-architecture level, we observe memory bound latency to be the major cause of work time inflation.

1 Introduction

With a deluge in the volume and variety of data being collected at enormous rates, various enterprises, like Yahoo, Facebook and Google, are deploying clusters to run data analytics that extract valuable information from petabytes of data. For this reason various frameworks have been developed to target applications in the domain of batch processing [17], graph processing [13] and stream processing [20]. Clearly large clusters of commodity servers are the most cost-effective way to process exabytes but first, majority of analytic jobs do not process huge data sets [2]. Second, machine learning algorithms are becoming increasingly common, which work on filtered datasets that can easily fit into memory of modern scale-up servers. Third, today's servers can have substantial CPU, memory, and storage I/O resources. Therefore it is worthwhile to consider data analytics on modern scale-up servers.

In order to ensure effective utilization of scale-up servers, it is imperative to make a workload-driven study on the requirements that big data analytics put on processor and memory architectures. There have been several studies focusing on characterizing the behaviour of big data workloads and identifying the mismatch between the processor and the big data applications [6,9–12,21,23]. However, these studies lack in quantifying the impact of processor inefficiencies on the performance of in memory data analytics, which is impediment to propose novel hardware designs to increase the efficiency of modern servers for in-memory data analytics. To fill in this gap, we perform an extensive performance characterization of these workloads on a scale-up server using Spark framework.

In summary, we make the following contributions:

- We perform an in-depth evaluation of Spark based data analysis workloads on a scale-up server.
- We discover that work time inflation (the additional CPU time spent by threads in a multi-threaded computation beyond the CPU time required to perform the same work in a sequential computation) and load imbalance on the threads are the scalability bottlenecks.
- We quantify the impact of micro-architecture on the performance, and observe that DRAM latency is the major bottleneck.

2 Background

2.1 Spark

Spark is a cluster computing framework that uses Resilient Distributed Datasets (RDDs) [25], 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. 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. A task is a combination of data and computation. Tasks are assigned to executor pool threads. Spark executes all tasks within a stage before moving on to the next stage. Table 1 describe the parameters necessary to configure Spark properly in local mode on a scale-up server.

Table 1: Spark Configuration Parameters

Parameter	**Description**
spark.storage.memoryFraction	fraction of Java heap to use for Spark's memory cache
spark.shuffle.compress	whether to compress map output files
spark.shuffle.consolidateFiles	whether to consolidates intermediate files created during a shuffle
spark.broadcast.compress	whether to compress broadcast variables before sending them
spark.rdd.compress	whether to compress serialized RDD partitions
spark.default.parallelism	default number of tasks to use for shuffle operations (reduceByKey,groupByKey, etc) when not set by user

2.2 Top-Down Method for Hardware Performance Counters

Super-scalar processors can be conceptually divided into the "front-end" where instructions are fetched and decoded into constituent operations, and the "back-end" where the required computation is performed. A pipeline slot represents the hardware resources needed to process one micro-operation. The top-down method assumes that for each CPU core, there are four pipeline slots available per clock cycle. At issue point each pipeline slot is classified into one of four base categories: Front-end Bound, Back-end Bound, Bad Speculation and Retiring. If a micro-operation is issued in a given cycle, it would eventually either get retired or 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 [22].

3 Methodology

3.1 Benchmarks

We select the benchmarks based on following criteria; (a) Workloads should cover a diverse set of Spark lazy transformations and actions, (b) Same transformations with different compute complexity functions should be included, (c) Workloads should be common among different Big Data Benchmark suites available in the literature.(d) Workloads have been used in the experimental evaluation of Map-Reduce frameworks for Shared-Memory Systems.

Table 2 shows the list of benchmarks along with transformations and actions involved. Most of the workloads have been used in popular data analysis workload suites such as BigDataBench [21], DCBench [9], HiBench [8] and Cloudsuite [6]. Phoenix++ [19], Phoenix rebirth [24] and Java MapReduce [18] tests the performance of devised shared-memory frameworks based on Word Count, Grep and K-Means. We use Spark version of the selected benchmarks from BigDataBench and employ Big Data Generator Suite (BDGS), an open source tool, to generate synthetic datasets for every benchmark based on raw data sets [14]. We work with smaller datasets deliberately to fully exploit the potential of in-memory data processing.

- **Word Count (Wc)** counts the number of occurrences of each word in a text file. The input is unstructured 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. It works on unstructured Wikipedia Entries.
- **Sort (So)** ranks records by their key. Its input is a set of samples. Each sample is represented as a numerical d-dimensional vector.
- **Naive Bayes (Nb)** uses semi-structured Amazon Movie Reviews data-sets for sentiment classification. We use only the classification part of the benchmark in our experiments.
- **K-Means (Km)** clusters data points into a predefined number of clusters. We run the benchmark for 4 iterations with 8 desired clusters. Its input is structured records, each represented as a numerical d-dimensional vector.

3.2 System Configuration

Table 3 shows details about our test machine. Hyper-Threading and Turbo-boost are disabled through BIOS because it is difficult to interpret the micro-architectural data with these features enabled [4]. With Hyper-Threading and Turbo-boost disabled, there are 24 cores in the system operating at the frequency of 2.7 GHz.

Table 4 also lists the parameters of JVM and Spark. For our experiments, we use HotSpot JDK version 7u71 configured in server mode (64 bit). The heap size is chosen to avoid getting "Out of memory" errors while running the benchmarks. The open file limit in Linux is increased to avoid getting "Too many files open in the system" error. The young generation space is tuned for every benchmark to minimize the time spent both on young generation and old generation garbage

Table 2: Benchmarks

Benchmarks		Transformations	Actions
Micro-benchmarks	Word count	map reduceByKey	saveAsTextFile
	Grep	filter	saveAsTextFile
	Sort	map sortByKey	saveAsTextFile
Classification	Naive Bayes	map	collect saveAsTextFile
Clustering	K-Means	map mapPartitions reduceByKey filter	takeSample collectAsMap collect

Table 3: 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 0.8.0	

collection, which in turn reduces the execution time of the workload. The size of young generation space and the values of Spark internal parameters after tuning are available in Table 4.

Table 4: JVM and Spark Parameters for Different Workloads.

Parameters		**Wc**	**Gp**	**So**	**Km**	**Nb**
JVM	Heap Size (GB)	50				
	Young Generation Space (GB)	45	25	45	15	45
	MaxPermSize (MB)	512				
	Old Generation Garbage Collector	ConcMarkSweepGC				
	Young Generation Garbage Collector	ParNewGC				
Spark	spark.storage.memoryFraction	0.2	0.2	0.2	0.6	0.2
	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				

3.3 Measurement Tools and Techniques

We use jconsole to measure time 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 [1] to perform concurrency analysis and general micro-architecture exploration. For scalability study, each benchmark is run 10 times within a single JVM invocation and the median values of last 5 iterations are reported. For concurrency analysis, each benchmark is run 3 times within a single JVM invocation and Vtune measurements are recorded for the last iteration. This experiment is repeated 3 times and the best case in terms of execution time of the application is chosen. The same measurement technique is also applied in general architectural exploration, however the difference is best case is chosen on basis of IPC. Additionally, executor pool threads are bound to the cores before collecting hardware performance counter values. Although this measurement method is not the most optimal for Java experiments as suggested by Georges et al [7], we believe, it is enough for Big Data applications. We use a top-down analysis method proposed by Yasin [22] to identify the micro-architectural inefficiencies.

3.4 Metrics

The definition of metrics used in this paper, are taken from Intel Vtune online help [1].

- **CPU Time:** is time during which the CPU is actively executing your application on all cores.
- **Wait Time:** occurs when software threads are waiting on I/O or due to synchronization.
- **Spin Time:** is wait time during which the CPU is busy. This often occurs when a synchronization API causes the CPU to poll while the software thread is waiting.
- **Core Bound:** shows how core non-memory issues limit the performance when you run out of out-of-order execution resources or are saturating certain execution units.
- **Memory Bound:** measures a fraction of cycles where pipeline could be stalled due to demand load or store instructions.
- **DRAM Bound:** shows how often CPU was stalled on the main memory.
- **L1 Bound:** shows how often machine was stalled without missing the L1 data cache.
- **L2 Bound:** shows how often machine was stalled on L2 cache.
- **L3 Bound:** shows how often CPU was stalled on L3 cache, or contended with a sibling Core.
- **Store Bound:** This metric shows how often CPU was stalled on store operations.
- **Front-End Bandwidth:** represents a fraction of slots during which CPU was stalled due to front-end bandwidth issues.
- **Front-End Latency:** represents a fraction of slots during which CPU was stalled due to front-end latency issues.

4 Scalability Analysis

In this section, we evaluate the scalability of benchmarks. Speed-up is calculated as T_1/T_n, where T_1 is the execution time with a single executor pool thread, and T_n is the execution time using n threads in the executor pool.

4.1 Application Level

Figure 1 shows the speed-up of workloads for increasing number of executor pool threads. All workloads scale perfectly up to 4 threads. From 4 to 12 threads, they show linear speed-up. Beyond 12 threads, Word Count and Grep scale linearly but the speed-up for Sort, K-Means and Naive Bayes tend to saturate.

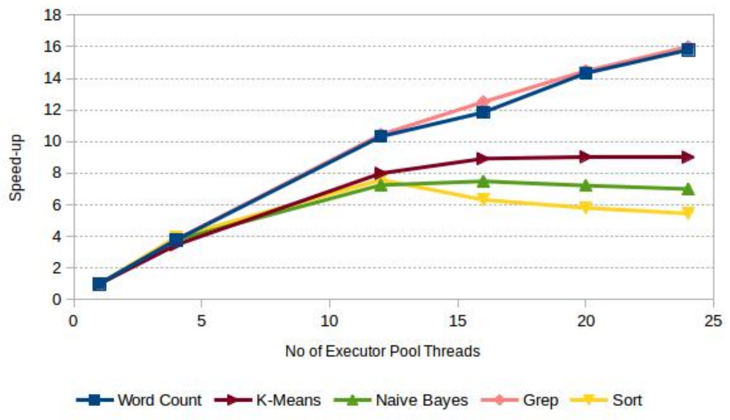
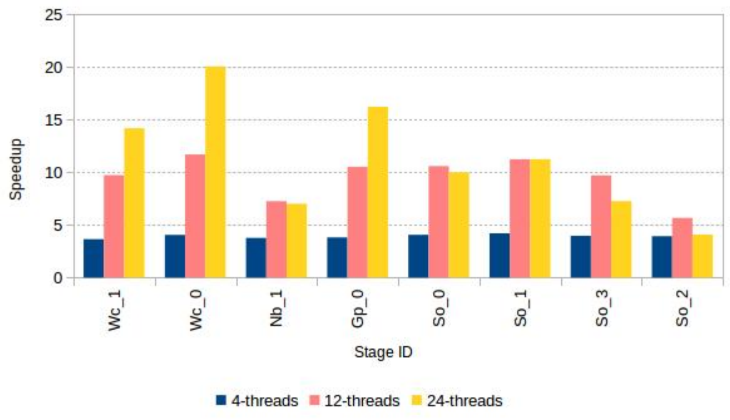


Figure 1: Scalability of Spark Workloads in Scale up Configuration

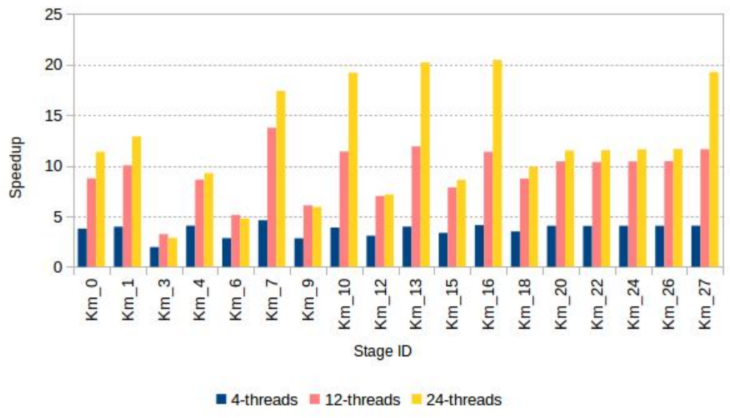
4.2 Stage Level

Next we drill down to stage level and observe how different stages scale with the number of executor pool threads. We only study those stages whose execution time contributes to 5% of total execution time of workload, e.g Naive Bayes has 2 stages but only the stage Nb_1 contributes significantly to the total execution time. Grep has only a filter stage, Word Count has a map stage (Wc_1) and a reduce stage (Wc_2). In Sort, So_0 and So_3 are map stages, So_1 is SortByKey stage and sorted data is written to local file in So_2 stage. In K-Means, map stages are Km_0, Km_18, Km_20, Km_22, Km_22, Km_24 and Km_26. Km_1 and Km_27 are takeSample and sum stages. Stages Km_3, Km_4, Km_6, Km_7, Km_9, Km_10, Km_12, Km_13, Km_15 and Km_16 perform mapPartitionswithIndex transformation. Stages up to Km_18 belong to initialization phase whereas the remaining ones belong to the iteration phase of K-Means.

At 4-threads case (see Figure 2a), all stages of a workload exhibit ideal scalability but in 12 and 24-threads, the scalability characteristics vary among the stages, e.g. Wc_0 shows better speed-up than Wc_1 in 24-threads case. The scalability of Sort is worst among all applications in 24-threads case because of So_2 stage that does not scale beyond 4 threads. In K-Means (see Figure 2b), stages where mapPartitionswithIndex transformations are performed show better scalability than map stages both in the initialization and iteration phases. The scalability of map transformations vary, e.g in 24-threads case, map stage in Word Count has better scalability than that in Sort, Naive Bayes and K-Means.This can be attributed to the complexity of user defined functions in map transformations.



(a) Word Count, Naive Bayes, Grep and Sort



(b) K-Means

Figure 2: Performance at Stage Level

4.3 Tasks Level

Figure 3a and 3b show the execution time of tasks in Wc_1 and Km_0 stage respectively. Note that the size of task set does not change with increase in threads in the executor pool because it depends on the size of input data set. The data set is split into chunks of 32 MB by default. The figures show that execution time of tasks increases with increase in threads in the executor pool. To quantify the increase, we calculate area under the curves (AUC) using trapezoidal approximation. Table 5 presents percentage increase in AUC for various workloads in multi-threaded cases over 1-thread case. For Wc_1, there is 17% and 61% increase in AUC 12-threads and 24-threads case over 1-thread case. For So_3, there is 24% and 68% increase where as for Km_0, the increase is 38% and 83%

Table 5: Percentage increase in AUC compared to 1-thread

Stage	**12-threads**	**24-threads**
Wc_1	17.03	61.50
So_3	24.58	68.50
Km_0	38.02	83.20

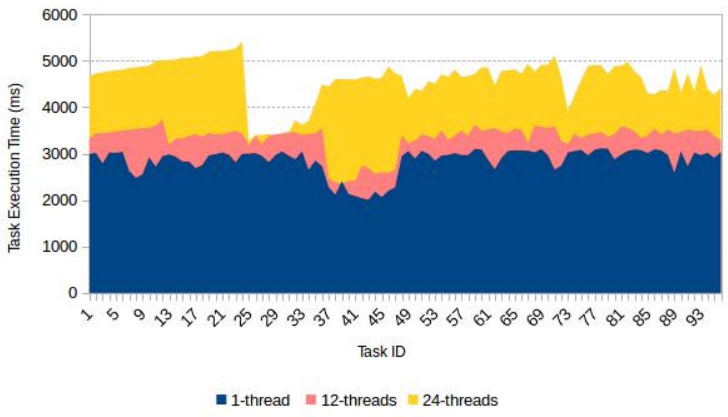
5 Scalability Limiters

5.1 CPU Utilization

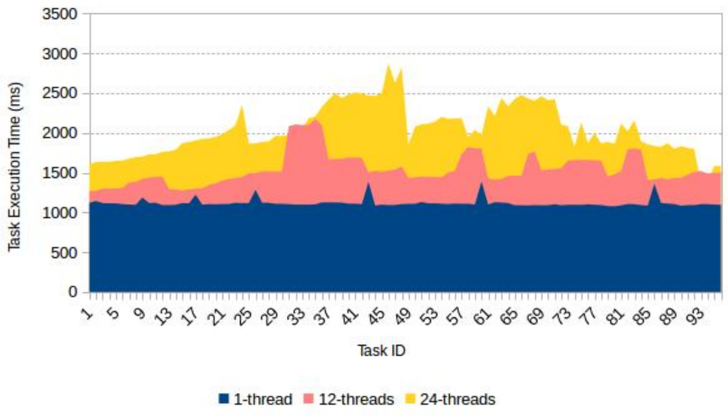
Figure 4 shows the average number of CPU's used during the execution time of benchmarks for different number of threads in the executor pool. By comparing this data with speed-up numbers in Figure 1, we see a strong correlation between the two for 4-threads case and 12-threads case. At 4-threads case, 4 cores are fully utilized in all benchmarks, At 12-threads case, Word Count, K-Means and Naive Bayes utilize 12 cores, whereas Grep and Sort utilize 10 and 8 cores respectively. At 24-threads case, none of the benchmarks utilize more than 20 cores. This utilization further drop to 16 for Grep and 6 for Sort. The performance numbers scale accordingly for these two benchmarks but for Word Count, K-Means and Naive Bayes, the performance is not scaling along with CPU utilization. We try to answer why such behaviour exists on these programs in subsequent sections

5.2 Load Imbalance on Threads

Load imbalance means that one or a few executor pools threads need (substantially) more CPU time than other threads, which limits the achievable speed-up, as the threads with less CPU time will have more wait time and if the CPU time across the threads is balanced, over-all execution time will decrease. Figure 5a breaks down elapsed time of each executor pool thread in K-Means in to CPU time and wait time for 24-threads case. The worker threads are shown in descending order of CPU time. The figure shows load imbalance. To quantify load imbalance, we compute the standard deviation of CPU time and show for 4, 12 and 24-threads case for all benchmarks in Figure 5b. The problem of load imbalance gets severe at higher number of threads. The major causes of load imbalance are; a non uniform division of the work among the threads,resource sharing, cache coherency or synchronization effects through barriers [5].



(a) Word Count (Wc_1)



(b) Kmeans (Km_0)

Figure 3: Performance at Task Level

5.3 Work Time Inflation

In this section, we drill down at threads level and analyse the behaviour of only executor pool threads because they contribute to 95% of total CPU time during the entire run of benchmarks. By filtering out executor pool threads in the concurrency analysis of Intel Vtune, we compute the total CPU time, spin time and wait time of worker threads and the numbers are shown in Figure 6a for K-Means at 1, 4, 12 and 24-threads case. The CPU time in 1-thread case is termed as sequential time, the additional CPU time spent by threads in a multi-threaded computation beyond the CPU time required to perform the same work in a sequential computation is termed as work time inflation as suggested by Oliver et-al [15].

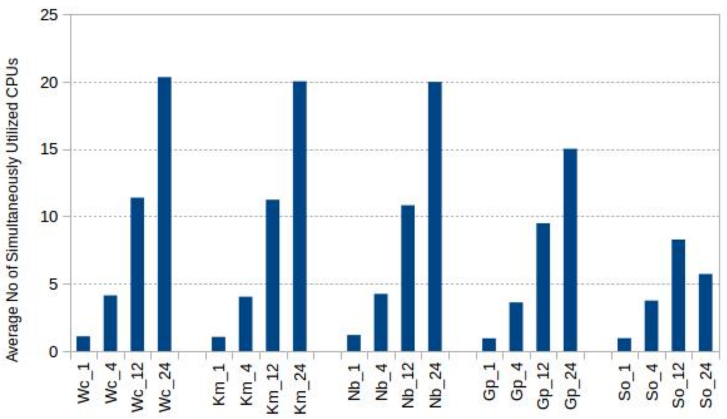


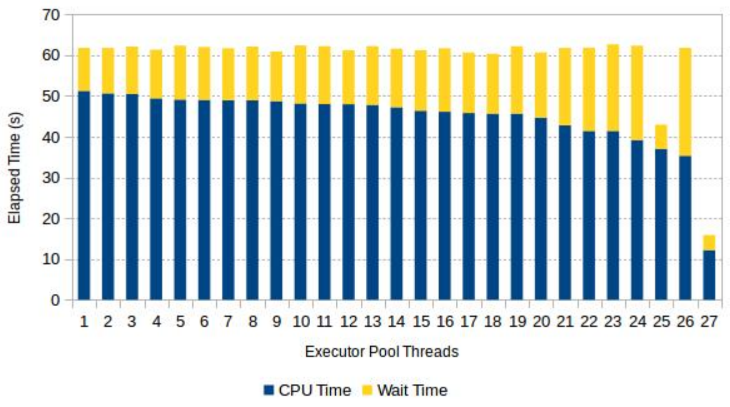
Figure 4: CPU Utilization of Benchmarks

Figure 6b shows the percentage contribution of sequential time, work time inflation, spin time and wait time towards the elapsed time of applications. The spinning overhead is not significant since it contribution is less than around 5% across all workloads in both sequential and multi-threaded cases. The contribution of wait time tends to increase with increase in threads in the executor pool. The percentage fractions are increased by, 20% in Word Count and K-Means, 15% in Naive Bayes, 25% in Grep and 70% in Sort. Word Count, K-Means and Naive Bayes see increase in fraction of work time inflation with increase in threads in the executor pool. At 24-threads, the contribution of work time inflation is 20%, 36% and 51% in Word count, K-Means and Naive Bayes respectively. For Grep and Sort, this overhead is between 5-6% at 24-threads case.

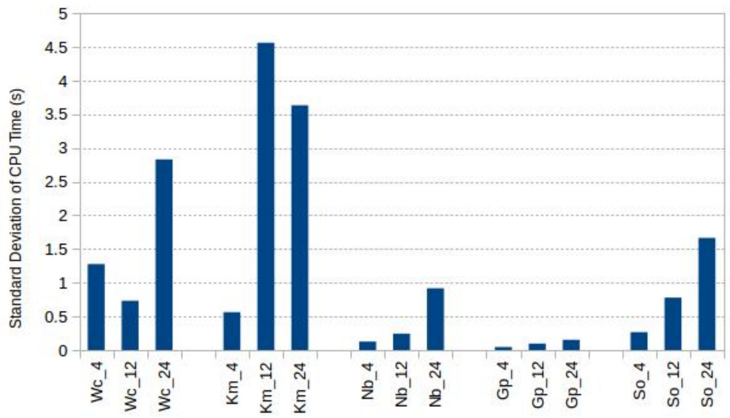
By comparing the data in Figure 6 with performance data in Figure 1, we see that Grep does not scale because of wait time overhead. Sort has the worst scalability because of significant contribution of wait time. In Word Count, there is equal contribution of work time inflation and wait time overhead where as K-Means and Naive Bayes are mostly dominant by work time inflation. Moreover the work time inflation overhead also correlates with speed-up numbers, i.e. Word Count having less work time inflation scales better than K-Means and Naive Bayes having largest contribution of work time inflation scales poorer than K-Means. In the next section, we try to find out the micro-architectural reasons that result in work time inflation.

5.4 Micro-architecture

Top Level Figure 7b shows the breakdown of pipeline slots for the benchmarks running with different number of executor pool threads. On average across the workloads; Retiring category increases from 33.4% in 1-thread case to 35.7% in



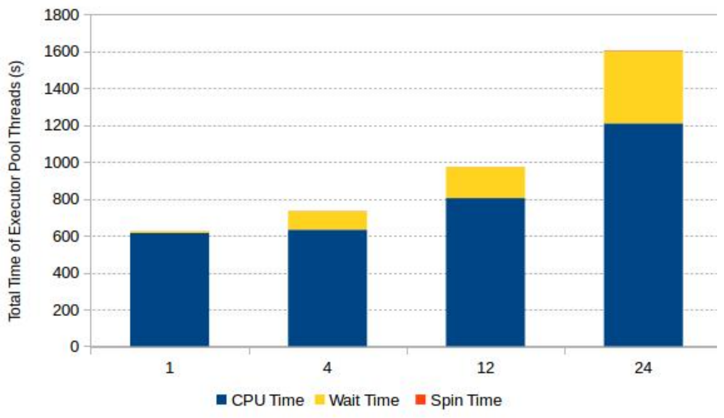
(a) K-Means



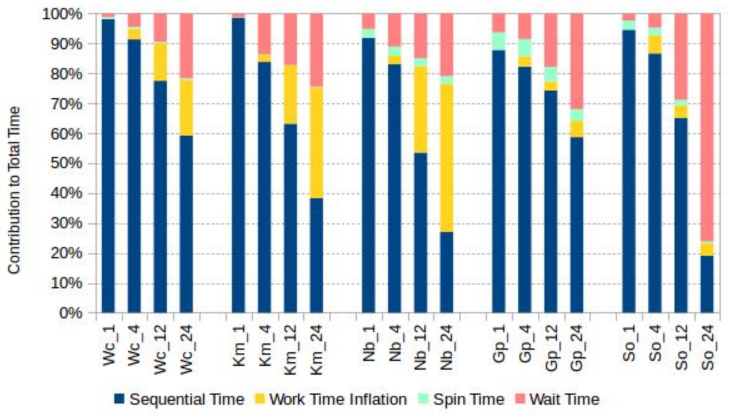
(b) Variation from Mean CPU Time for Different No of Executor Pool Threads

Figure 5: Load Imbalance in Spark Benchmarks

12-threads case (Note how well it correlates to IPC) and decreases to 31% in 24-threads case, Bad Speculation decreases from 4.7% 1-threads case to 3.1% in 24-threads case, Front-end bound decreases from 20.4% in 1-thread case to 12.6% in 24-threads case and Back-end bound increases from 42.9% in 1-thread case to 54.3% 12-threads case. This implies that workloads do not scale because of issues at the Back-end. The contribution of Back-end bound increases with increase in number of worker threads in workloads suffering with work time inflation and in 24-threads case, it correlates with speed-up, i.e. the higher the Back-end bound is, the lower the speed-up is.



(a) K-Means



(b) Elapsed Time Breakdown

Figure 6: Work Time Inflation in Spark Benchmarks

Backend Level Figure 7c shows the contribution of memory bound stalls and core bound stalls. On average across the workloads; the fraction of memory bound stalls increases from 55.6% in 1-thread case to 72.2% in 24-threads. It also shows that workloads exhibiting larger memory bound stalls results in higher work time inflation.

Memory Level Next we drill down into Memory level in Figure 7d. The Memory level breakdown suggests that on average across the workloads, fraction of L1 bound stalls decrease from 34% to 23%, fraction of L3 bound stalls decrease from 16% to 10%, fraction of Store bound stalls increase from 9% to 11% and the fraction of

DRAM bound stalls increase 42% to 56%, when comparing the 1-thread and 24-threads cases. The increase in fraction of DRAM bound stalls correlate to work time inflation, 30% increase in DRAM bound stalls yields higher work time inflation Naive Bayes that K-Means for 24-threads case where increase in contribution of DRAM bound stalls is 20%. Word Count with only 10% increase in DRAM bound stalls shows exhibit lower amount of work time inflation than K-Means.

Execution Core Level Figure 7e shows the utilization of execution resources for benchmarks at multiple no of executor pool threads. On average across the workloads, the fraction of clock cycles during which no port is utilized (execution resources were idle) increases from 42.3% to 50.7%, fraction of cycles during which 1, 2 and 3 + ports are used decrease from 13.2% to 8.9%, 15.7% to 12.8% and 29.3% to 27.1% respectively, while comparing 1 and 24-threads case.

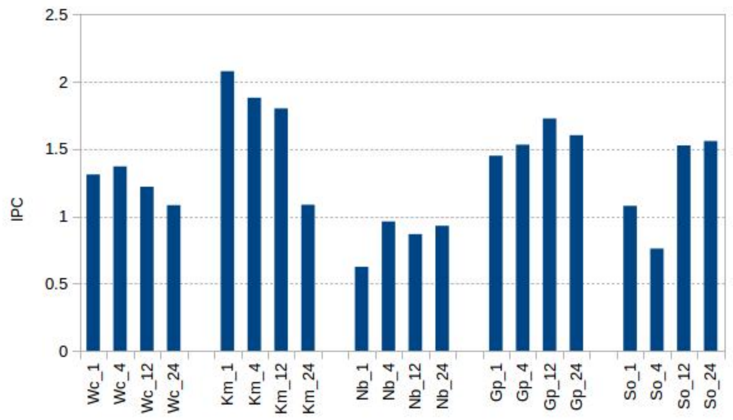
Frontend Level Figure 7f shows the fraction of pipeline slots during which CPU was stalled due to front-end latency and front-end bandwidth issues. At higher number of threads, front- end stalls are equally divided among latency and bandwidth issues. On average across the workloads; front-end latency bound stalls decrease from 11.8% in 1-thread case to 5.7% in 24-threads case where as front-end bandwidth bound stalls decrease from 8.6% to 6.9%.

5.5 Memory Bandwidth Saturation

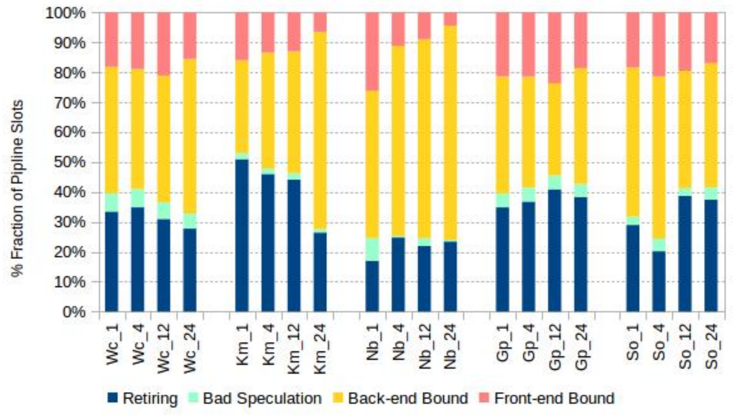
Figure 8 shows the amount of data read and written to each of the two DRAM packages via the processor's integrated memory controller. The bandwidth (Gigabytes/sec) to package_1 shows an increasing trend with increase in threads in the executor pool. The same trend can be seen for total memory bandwidth in most of the workloads. We also see an imbalance between memory traffic to two DRAM packages. Off-chip bandwidth requirements of Naive Bayes are higher than rest of the workloads but the peak memory bandwidth of all the workloads are with in the platform capability of 60 GB/s, hence we conclude that memory bandwidth is not hampering the scalability of in-memory data analysis workloads.

6 Related Work

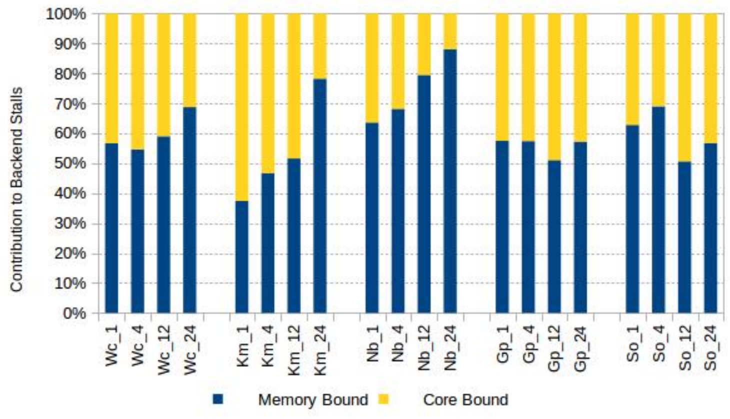
Oliver et al. [15] have shown that task parallel applications can exhibit poor performance due to work time inflation. We see similar phenomena in Spark based workloads. Ousterhout et al. [16] have developed blocked time analysis to quantify performance bottlenecks in the Spark framework and found out that CPU (and not I/O) is often the bottleneck. Our thread level analysis of executor pool threads also reveal that CPU time (and not wait time) is the dominant performance bottleneck in Spark based workloads.



(a) IPC

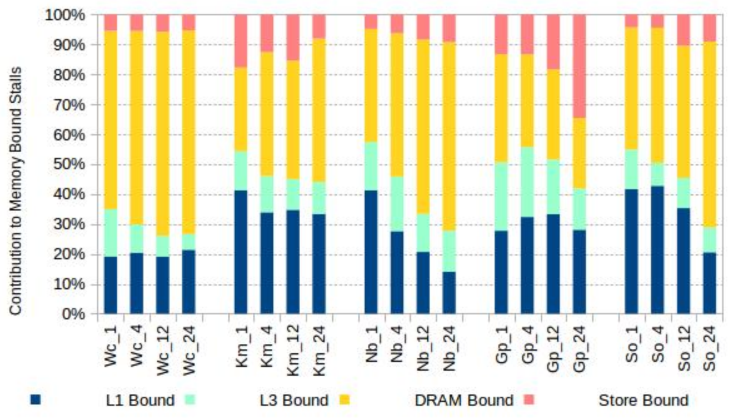


(b) Top Level

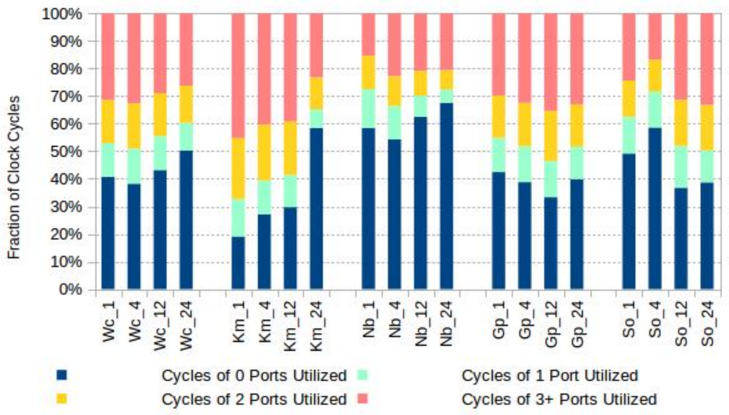


(c) Backend Level

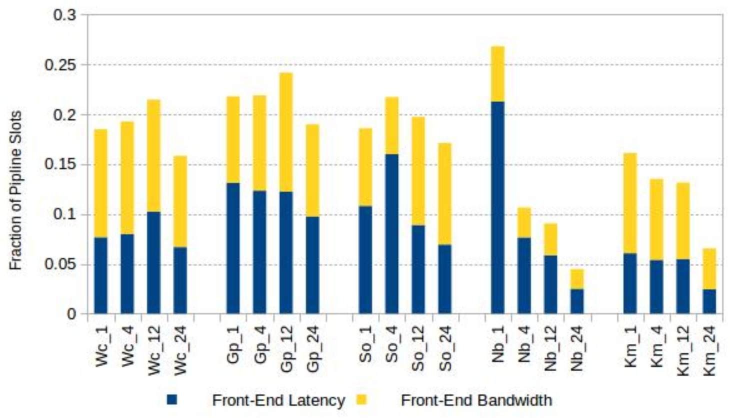
Figure 7: Top-Down Analysis Breakdown for Benchmarks with Different No of Executor Pool Threads



(d) Memory Level



(e) Core Level



(f) Frontend Level

Figure 7: Top-Down Analysis Breakdown for Benchmarks with Different No of Executor Pool Threads

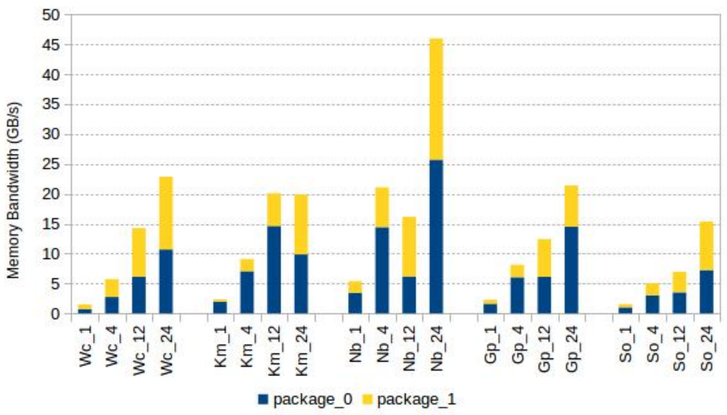


Figure 8: Memory Bandwidth Consumption of Benchmarks

Ferdman et al. [6] 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. [26] 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 [9]. They also suffer from notable from end stalls but L2 and L3 caches are effective for them. Wang et al. [21] conclude the same about L3 caches and L1 I Cache miss rates despite using larger data sets. Deep dive analysis [23] 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.

Jiang et al. [11] observe that memory access characteristics of the Spark and Hadoop workloads differ. At the micro-architecture level, they have roughly same behaviour and point current micro-architecture works for Spark workloads. Contrary to that, Jia et al. [10] conclude that Software stacks have significant impact on the micro-architecture behaviour of big data workloads. However both studies lack in quantifying the impact of micro-architectural inefficiencies on the performance. We extend the literature by identifying the bottlenecks in the memory subsystem.

7 Conclusion

We evaluated the performance of Spark based data analytic workloads on a modern scale-up server at application, stage, task and thread level. While performing experiments on a 24 core machine, we found that that most of the applications exhibit sub-linear speed-up, stages with map transformations do not scale, and execution time of tasks in these stages increases significantly. The CPU utilization for several workloads is around 80% but the performance does not scale along with

CPU utilization. Work time inflation and load imbalance on the threads are the scalability bottlenecks. We also quantified the impact of micro-architecture on the performance. Results show that issues in front end of the processor account for up to 20% of stalls in the pipeline slots, where as issues in the back end account for up to 72% of stalls in the pipeline slots. The applications do not saturate the available memory bandwidth and memory bound latency is the cause of work time inflation. We will explore pre-fetching mechanisms to hide the DRAM access latency in data analysis workloads, since Dimitrov et al. [3] show potential for aggressively pre-fetching large sections of the dataset onto a faster tier of memory subsystem.

Bibliography

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

[2] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony I. T. Rowstron. Scale-up vs scale-out for hadoop: time to rethink? In *ACM Symposium on Cloud Computing, SOCC*, page 20, 2013.

[3] Martin Dimitrov, Karthik Kumar, Patrick Lu, Vish Viswanathan, and Thomas Willhalm. Memory system characterization of big data workloads. In *BigData Conference*, pages 15–22, 2013.

[4] D.Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. In *Intel Performance Analysis Guide*, 2009.

[5] Stijn Eyerman, Kristof Du Bois, and Lieven Eeckhout. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*, ISPASS '12, pages 145–155, 2012.

[6] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak 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*, ASPLOS XVII, pages 37–48, 2012.

[7] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, 2007.

[8] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51, 2010.

[9] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. Characterizing data analysis workloads in data centers. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 66–76, 2013.

[10] Zhen Jia, Jianfeng Zhan, Lei Wang, Rui Han, Sally A. McKee, Qiang Yang, Chunjie Luo, and Jingwei Li. Characterizing and subsetting big data workloads. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 191–201, 2014.

[11] Tao Jiang, Qianlong Zhang, Rui Hou, Lin Chai, Sally A. McKee, Zhen Jia, and Ninghui Sun. Understanding the behavior of in-memory computing workloads. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 22–30, 2014.

[12] Vasileios Karakostas, Osman S. Unsal, Mario Nemirovsky, Adrian Cristal, and Michael Swift. Performance analysis of the memory management unit under scale-out workloads. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 1–12, Oct 2014.

[13] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[14] Zijian Ming, Chunjie Luo, Wanling Gao, Rui Han, Qiang Yang, Lei Wang, and Jianfeng Zhan. BDGS: A scalable big data generator suite in big data benchmarking. In *Advancing Big Data Benchmarks*, volume 8585 of *Lecture Notes in Computer Science*, pages 138–154. 2014.

[15] Stephen L. Olivier, Bronis R. de Supinski, Martin Schulz, and Jan F. Prins. Characterizing and mitigating work time inflation in task parallel programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 65:1–65:12, 2012.

[16] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, 2015.

[17] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[18] Jeremy Singer, George Kovoor, Gavin Brown, and Mikel Luján. Garbage collection auto-tuning for java mapreduce on multi-cores. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 109–118, 2011. ISBN 978-1-4503-0263-0.

[19] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce, pages 9–16, 2011.

[20] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, *et al.* Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[21] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. Bigdatabench: A big data benchmark suite from internet services. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA*, pages 488–499, 2014.

[22] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, pages 35–44, 2014.

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

[24] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, pages 198–207, 2009.

[25] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion 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)*, pages 15–28, San Jose, CA, 2012. ISBN 978-931971-92-8.

[26] Chen Zheng, Jianfeng Zhan, Zhen Jia, and Lixin Zhang. Characterizing os behavior of scale-out data center workloads. In *The SeventhAnnual Workshop on the Interaction amongst Virtualization,Operating Systems and Computer Architecture(WIVOSCA2013) held in conjunction with The 40th International Symposium on Computer Architecture*, 2013.

Publication B

How Data Volume Affects Spark Based Data Analytics on a Scale-up Server.
Ahsan Javed Awan, Mats Brorsson, Vladmir Vlassov and Eduard Ayguade.
6th International Workshop on Big data Benchmarks, Performance Optimization and Emerging Hardware (BpoE) held in conjunction with 41st International Conference on Very Large Data Bases (VLDB), Hawaii, USA, 2015.

How Data Volume Affects Spark Based Data Analytics on a Scale-up Server

Ahsan Javed Awan[1], Mats Brorsson[1], Vladimir Vlassov[1] and Eduard Ayguade[2]

[1]KTH Royal Institute of Technology,
Software and Computer Systems Department(SCS),
{ajawan,matsbror,vladv}@kth.se
[2]Technical University of Catalunya (UPC),
Computer Architecture Department,
eduard@ac.upc.edu

Abstract

Sheer increase in volume of data over the last decade has triggered research in cluster computing frameworks that enable web enterprises to extract big insights from big data. While Apache Spark is gaining popularity for exhibiting superior scale-out performance on the commodity machines, the impact of data volume on the performance of Spark based data analytics in scale-up configuration is not well understood. We present a deep-dive analysis of Spark based applications on a large scale-up server machine. Our analysis reveals that Spark based data analytics are DRAM bound and do not benefit by using more than 12 cores for an executor. By enlarging input data size, application performance degrades significantly due to substantial increase in wait time during I/O operations and garbage collection, despite 10% better instruction retirement rate (due to lower L1 cache misses and higher core utilization). We match memory behaviour with the garbage collector to improve performance of applications between 1.6x to 3x.

1 Introduction

With a deluge in the volume and variety of data collected, large-scale 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 huge data sets and that modern scale-up servers are adequate to run analytics jobs [4]. 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 [20], Ostrich [6] and Polymer [22] are specifically designed to exploit the potential of a single scale-up server, they don't scale-out to multiple scale-up servers. Apache Spark [21] is getting popular in 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 it's performance on modern scale-up servers is not fully understood. A recent study [5] characterizes the performance of Spark based data analytics on a scale-up server but it does not quantify the impact of data volume. Knowing the limitations of modern scale-up servers for Spark based data analytics will help in achieving the future goal of improving the performance of Spark based data analytics on small clusters of scale-up servers. In this paper, we answer the following questions concerning Spark based data analytics running on modern scale-up servers:

- Do Spark based data analytics benefit from using larger scale-up servers?
- How severe is the impact of garbage collection on performance of Spark based data analytics?
- Is file I/O detrimental to Spark based data analytics performance?
- How does data size affect the micro-architecture performance of Spark based data analytics?

To answer the above questions, we use empirical evaluation of Apache Spark based benchmark applications on a modern scale-up server. Our contributions are:

- We evaluate the impact of data volume on the performance of Spark based data analytics running on a scale-up server.
- We find the limitations of using Spark on a scale-up server with large volumes of data.
- We quantify the variations in micro-architectural performance of applications across different data volumes.

2 Background

Spark is a cluster computing framework that uses Resilient Distributed Datasets (RDDs) [21] 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. 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. A task is a combination of data and computation. Tasks are assigned to executor pool threads. Spark executes all tasks within a stage before moving on to the next stage.

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.

3 Methodology

3.1 Benchmarks

Table 1 shows the list of benchmarks along with transformations and actions involved. We used Spark versions of the following benchmarks from BigDataBench [17]. Big Data Generator Suite (BDGS), an open source tool was used to generate synthetic datasets based on raw data sets [15].

- **Word Count (Wc)** counts the number of occurrences of each word in a text file. The input is unstructured 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. It works on unstructured Wikipedia Entries.
- **Sort (So)** ranks records by their key. Its input is a set of samples. Each sample is represented as a numerical d-dimensional vector.
- **Naive Bayes (Nb)** uses semi-structured Amazon Movie Reviews data-sets for sentiment classification. We use only the classification part of the benchmark in our experiments.

K-Means (Km) clusters data points into a predefined number of clusters. We run the benchmark for 4 iterations with 8 desired clusters. Its input is structured records, each represented as a numerical d-dimensional vector.

Table 1: Benchmarks.

Benchmarks		**Transformations**	**Actions**
Micro-benchmarks	Word count	map, reduceByKey	saveAsTextFile
	Grep	filter	saveAsTextFile
	Sort	map, sortByKey	saveAsTextFile
Classification	Naive Bayes	map	collect saveAsTextFile
Clustering	K-Means	map, filter mapPartitions reduceByKey	takeSample collectAsMap collect

3.2 System Configuration

Table 2 shows details about our test machine. Hyper-Threading and Turbo-boost are disabled through BIOS because it is difficult to interpret the micro-architectural data with these features enabled [8]. 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. For our experiments, 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 three combinations: (1) Parallel Scavenge (PS) and Parallel Mark Sweep; (2) Parallel New and Concurrent Mark Sweep; and (3) G1 young and G1 mixed for young and old generations respectively. The details on each algorithm are available [2,7]. The heap size is chosen to avoid getting "Out of memory" errors while running the benchmarks. The open file limit in Linux is increased to avoid getting "Too many files open in the system" error. The values of Spark internal parameters after tuning are given in Table 3. Further details on the parameters are available [3].

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 [1] to perform concurrency analysis and general micro-architecture exploration. For scalability study, each benchmark is run 5 times within a single JVM invocation

Table 2: Machine Details.

Component	**Details**	
Processor	Intel Xeon E5-2697 V2, Ivy Bridge micro-architecture	
	Cores	12 @ 2.7 GHz (Turbo upto 3.5 GHz)
	Threads	2 per core
	Sockets	2
	L1 Cache	32 KB for instructions and 32 KB for data per core
	L2 Cache	256 KB per core
	L3 Cache (LLC)	30 MB per socket
Memory	2 x 32 GB, 4 DDR3 channels, Max BW 60 GB/s	
OS	Linux kernel version 2.6.32	
JVM	Oracle Hotspot JDK version 7u71	
Spark	Version 1.3.0	

Table 3: JVM and Spark Parameters for Different Workloads.

		Wc	Gp	So	Km	Nb
JVM	Heap Size (GB)	50				
	Old Generation Garbage Collector	PS MarkSweep				
	Young Generation Garbage Collector	PS Scavange				
Spark	spark.storage.memoryFraction	0.1	0.1	0.1	0.6	0.1
	spark.shuffle.memoryFraction	0.7	0.7	0.7	0.4	0.7
	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				

and the mean values are reported. For concurrency analysis, each benchmark is run 3 times within a single JVM invocation and Vtune measurements are recorded for the last iteration. This experiment is repeated 3 times and the best case in terms of execution time of the application is chosen. The same measurement technique is also applied in general architectural exploration, however the difference is that mean values are reported. Additionally, executor pool threads are bound to the cores before collecting hardware performance counter values.

We use the top-down analysis method proposed by Yasin [18] to study the micro-architectural performance of the workloads. Super-scalar processors can be conceptually divided into the "front-end" where instructions are fetched and decoded into constituent operations, and the "back-end" where the required computation is performed. A pipeline slot represents the hardware resources needed to process one micro-operation. The top-down method assumes that for each CPU core, there are four pipeline slots available per clock cycle. At issue point each pipeline slot is classified into one of four base categories: Front-end Bound, Back-end Bound, Bad Speculation and Retiring. If a micro-operation is issued in a given cycle, it would eventually either get retired or 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.

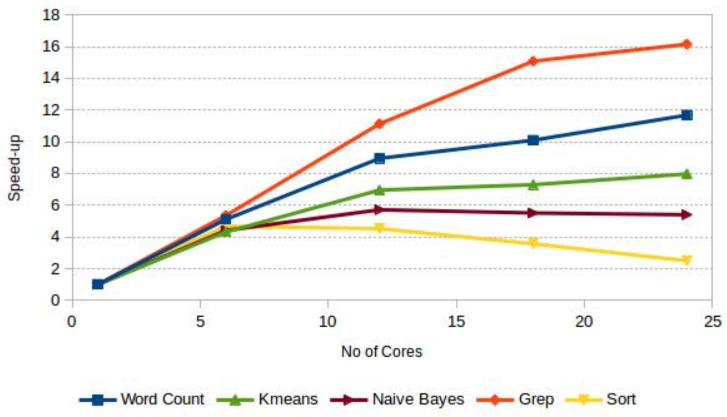
4 Scalability Analysis

4.1 Do Spark based data analytics benefit from using scale-up servers?

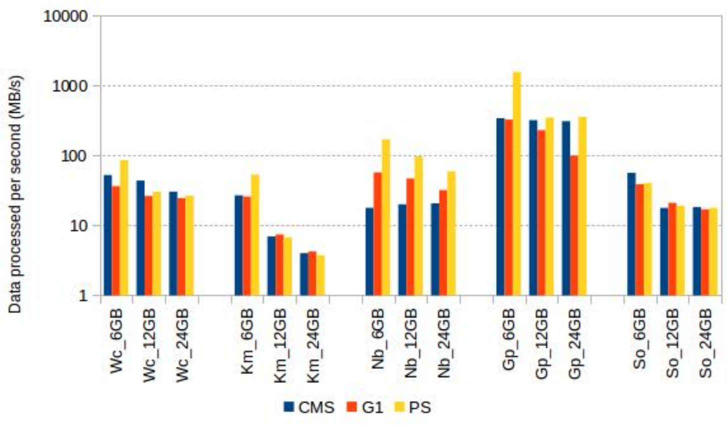
We configure spark to run in local-mode and used system configuration parameters of Table 3. Each benchmark is run with 1, 6, 12, 18 and 24 executor pool threads. The size of input data-set is 6 GB. For each run, we set the CPU affinity of the Spark process to emulate hardware with same number of cores 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 that 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 that Spark applications do not benefit significantly by using more than 12-core executors.

4.2 Does performance remain consistent as we enlarge the data size?

The benchmarks are configured to use 24 executor pool threads in the experiment. Each workload is run with 6 GB, 12 GB and 24 GB of input data and the amount of data processed per second (DPS) is calculated by dividing the input data size by the total execution time. The data sizes are chosen to stress the whole system and evaluate the system's data processing capability. In this regard, DPS is a relevant



(a) Benchmarks do not benefit by adding more than 12 cores.



(b) Data processed per second decreases with increase in data size.

Figure 1: Scale-up performance of applications: (a) when the number of cores increases and (b) when input data size increases.

metric as suggested in by Luo et al. [14]. We also evaluate the sensitivity of DPS to garbage collection schemes but explain it in the next section. Here we only analyse the numbers for Parallel Scavenge garbage collection scheme. By comparing 6 GB and 24 GB cases in Figure 1b, we see that K-Means performs the worst as its DPS decreases by 92.94% and Grep performs the best with a DPS decrease of 11.66%. Furthermore, we observe that DPS decreases by 49.12% on average across the workloads, when the data size is increased from 6 GB to 12 GB. However DPS decreases further by only 8.51% as the data size is increased to 24GB. In the next section, we will explain the reason for poor data scaling behaviour.

5 Limitations to Scale-up

5.1 How severe is the impact of garbage collection?

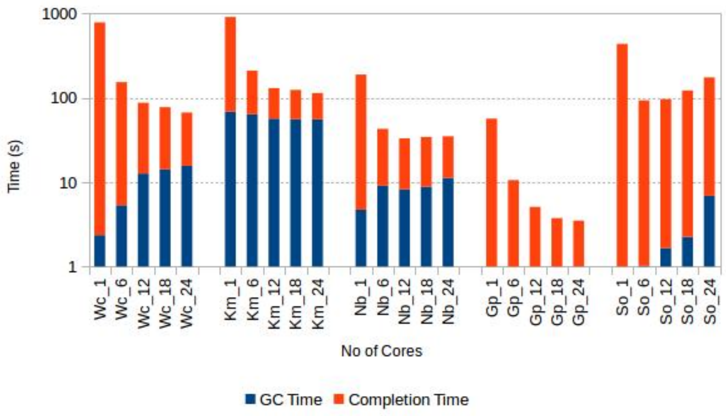
Because of the in-memory nature of most Spark computations, garbage collection can become a bottleneck for Spark programs. To test this hypothesis, we analysed garbage collection time of scalability experiments from the previous section. Figure 2a plots total execution time and GC time across the number of cores. The proportion of GC time in the execution time increases with the number of cores. At 24 cores, it can be as high as 48% in K-Means. Word Count and Naive Bayes also show a similar trend. This shows that if the GC time had at least not been increasing, the applications would have scaled better. Therefore we conclude that GC acts as a bottleneck.

To answer the question, "How does GC affect data processing capability of the system?", we examine the GC time of benchmarks running at 24 cores. The input data size is increased from 6 GB to 12 GB and then to 24 GB. By comparing 6 GB and 24 GB cases in Figure 2b, we see that GC time does not increase linearly, e.g., when input data is increased by 4x, GC time in K-Means increases by 39.8x. A similar trend is also seen for Word Count and Naive Bayes. This also shows that if GC time had been increasing at most linearly, DPS would not have decreased significantly. For K-Means, DPS decreases by 14x when data size increases by 4x. For similar scenario in Naive Bayes, DPS decreases by 3x and GC time increases by 3x. Hence we can conclude that performance of Spark applications degrades significantly because GC time does not scale linearly with data size.

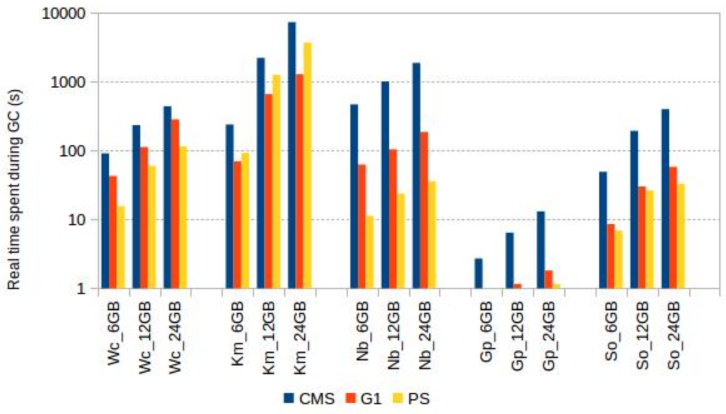
Finally we answer the question, "Does the choice of Garbage Collector impact the data processing capability of the system?". We look at impact of three garbage collectors on DPS of benchmarks at 6 GB, 12 GB and 24 GB of input data size. We study out-of-box (without tuning) behaviour of Concurrent Mark Sweep, G1 and Parallel Scavenge garbage collectors. Figure 2b shows that across all the applications, GC time of Concurrent Mark Sweep is the highest and GC time of Parallel Scavenge is the lowest among the three choices. By comparing the DPS of benchmarks across different garbage collectors, we see that Parallel Scavenge results in 3.69x better performance than Concurrent Mark Sweep and 2.65x better than G1 on average across the workloads at 6 GB. At 24 GB, Parallel Scavenge performs 1.36x better compared to Concurrent Mark Sweep and 1.69x better compared to G1 on average across the workloads.

5.2 Does file I/O become a bottleneck under large data volumes?

In order to find the reasons for poor performance of Spark applications under larger data volumes, we studied the thread-level view of benchmarks by performing concurrency analysis in Intel Vtune. We analyse only executor pool threads as they contribute to 95% of total CPU time during the entire run of the workloads. Fig-



(a) GC overhead is a scalability bottleneck.

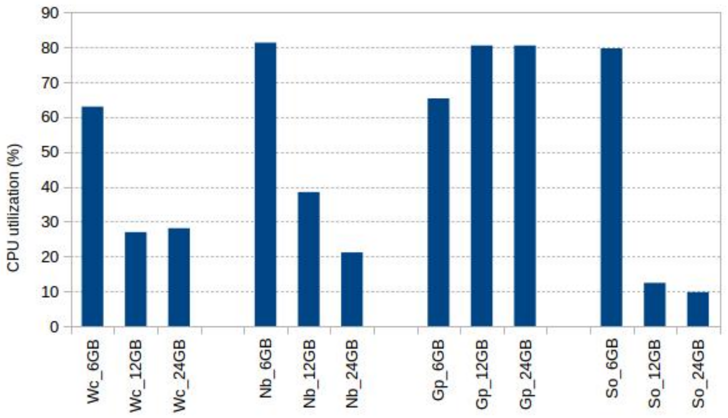


(b) GC time increases at a higher rate with data size.

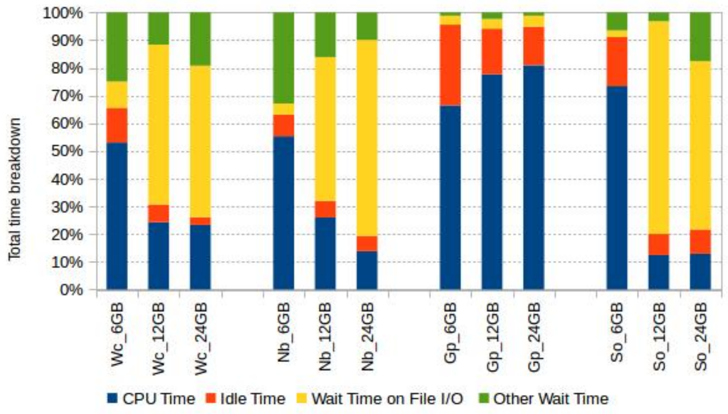
Figure 2: Impact of garbage collection on application performance: (a) when the number of cores increases and (b) when input data size increases.

ure 3b shows that CPU time and wait time of all executor pool threads. CPU time is the time during which the CPU is actively executing the application on all cores. Wait time occurs when software threads are waiting on I/O operations or due to synchronization. The wait time is further divided into idle time and wait on file I/O operations. Both idle time and file I/O time are approximated from the top 5 waiting functions of executor pool threads. The remaining wait time comes under the category of "other wait time".

It can be seen that the fraction of wait time increases with increase in input data size, except in Grep where it decreases. By comparing 6 GB and 24 GB case, the data shows that the fraction of CPU time decreases by 54.15%, 74.98% and 82.45%



(a) CPU utilization decreases with data size.



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

Figure 3: Time breakdown under executor pool threads.

in Word Count, Naive Bayes and Sort respectively; however it increases by 21.73% in Grep. The breakdown of wait time reveals that contribution of file I/O increases by 5.8x, 17.5x and 25.4x for Word Count, Naive Bayes and Sort respectively but for Grep, it increases only 1.2x. The CPU time in Figure 3b also correlates with CPU utilization numbers in Figure 3a. On average across the workloads, CPU utilization decreases from 72.34% to 39.59% as the data size is increased from 6 GB to 12 GB which decreases further by 5% in 24 GB case.

5.3 Is micro-architecture performance invariant to input data size?

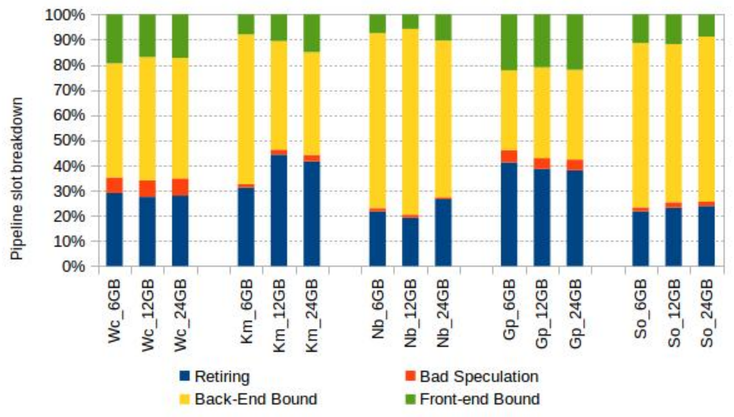
We study the top-down breakdown of pipeline slots in the micro-architecture using the general exploration analysis in Vtune. The benchmarks are configured to use 24 executor pool threads. Each workload is run with 6 GB, 12 GB and 24 GB of input data. Figure 4a shows that benchmarks are back-end bound. On average across the workloads, retiring category accounts for 28.9% of pipeline slots in 6 GB case and it increases to 31.64% in the 24 GB case. Back-end bound fraction decreases from 54.2% to 50.4% on average across the workloads. K-Means sees the highest increase of 10% in retiring fraction in 24 GB case in comparison to 6 GB case.

Next, we show the breakdown of memory bound stalls in Figure 4b. The term DRAM Bound refers to how often the CPU was stalled waiting for data from main memory. L1 Bound shows how often the CPU was stalled without missing in the L1 data cache. L3 Bound shows how often the CPU was stalled waiting for the L3 cache, or contended with a sibling core. Store Bound shows how often the CPU was stalled on store operations. We see that DRAM bound stalls are the primary bottleneck which account for 55.7% of memory bound stalls on average across the workloads in the 6 GB case. This fraction however decreases to 49.7% in the 24 GB case. In contrast, the L1 bound fraction increase from 22.5% in 6 GB case to 30.71% in 24 GB case on average across the workloads. It means that due to better utilization of L1 cache, the number of simultaneous data read requests to the main memory controller decreases at larger volume of data. Figure 4d shows that 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.

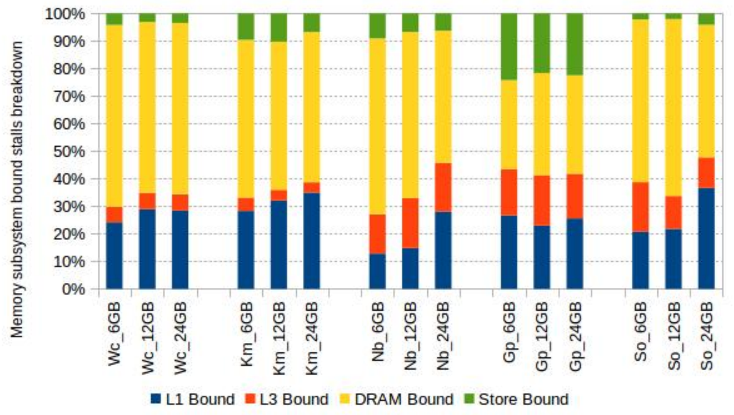
Figure 4c shows the fraction of cycles during execution ports are used. Ports provide the interface between instruction issue stage and the various functional units. By comparing 6 GB and 24 GB cases, we observe that cycles during which no port is used decrease from 51.9% to 45.8% on average across the benchmarks and cycles during which 1 or 2 ports are utilized increase from 22.2% to 28.7% on average across the workloads.

6 Related Work

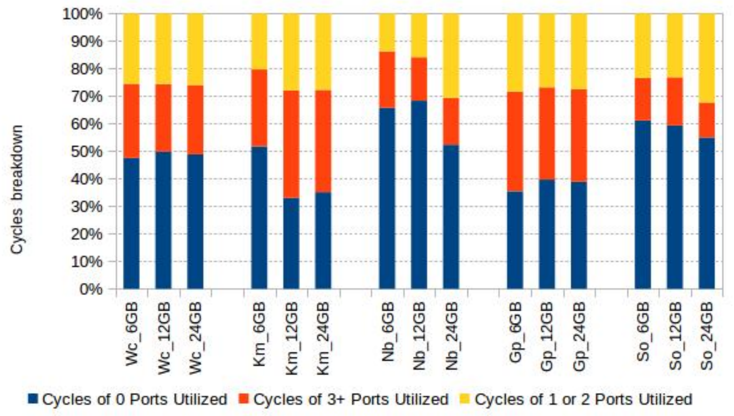
Several studies characterize the behaviour of big data workloads and identify the mismatch between the processor and the big data applications [9–13, 17, 19]. However these studies lack in identifying the limitations of modern scale-up servers for Spark based data analytics. Ousterhout et al. [16] have developed blocked time analysis to quantify performance bottlenecks in the Spark framework and have found out that CPU and not I/O operations are often the bottleneck. Our thread level analysis of executor pool threads shows that the conclusion made by Ousterhout et al. is only valid when the the input data-set fits in each node's memory in a scale-out setup. When the size of data set on each node is scaled-up, file I/O becomes the bottleneck again. Wang et al. [17] have shown that the volume of



(a) Retiring rate increases at larger datasets.

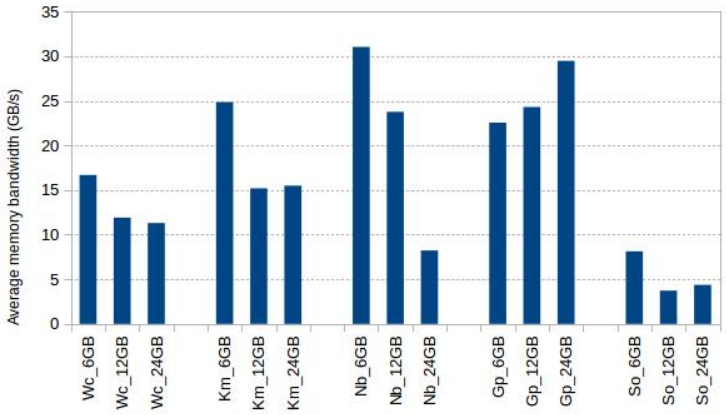


(b) L1 Bound stalls increase with data size.



(c) Port utilization increases at larger datasets.

Figure 4: Micro-architecture performance is inconsistent across different data sizes.



(d) Memory traffic decreases with data size.

Figure 4: Micro-architecture performance is inconsistent across different data sizes.

input data has considerable affect on the micro-architecture behaviour of Hadoop based workloads. We make similar observation about Spark based data analysis workloads.

7 Conclusions

We have reported a deep dive analysis of Spark based data analytics on a large scale-up server. The key insights we have found are as follows:

- Spark workloads do not benefit significantly from executors with more than 12 cores.
- The performance of Spark workloads degrades with large volumes of data due to substantial increase in garbage collection and file I/O time.
- With out any tuning, Parallel Scavenge garbage collection scheme outperforms Concurrent Mark Sweep and G1 garbage collectors for Spark workloads.
- Spark workloads exhibit improved instruction retirement due to lower L1 cache misses and better utilization of functional units inside cores at large volumes of data.
- Memory bandwidth utilization of Spark benchmarks decreases with large volumes of data and is 3x lower than the available off-chip bandwidth on our test machine.

We conclude that Spark run-time needs node-level optimizations to maximize its potential on modern servers. Garbage collection is detrimental to performance of in-memory big data systems and its impact could be reduced by careful matching of garbage collection scheme to workload. Inconsistencies in micro-architecture performance across the data sizes pose additional challenges for computer architects. Off-chip memory buses should be optimized for in-memory data analytics workloads by scaling back unnecessary bandwidth.

Bibliography

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

[2] Memory manangement in the java hotspot virtual machine. URL `http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`.

[3] Spark configuration. URL `https://spark.apache.org/docs/1.3.0/configuration.html`.

[4] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony I. T. Rowstron. Scale-up vs scale-out for hadoop: time to rethink? In *ACM Symposium on Cloud Computing, SOCC*, page 20, 2013.

[5] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguadé. Performance characterization of in-memory data analytics on a modern cloud server. *arXiv preprint arXiv:1506.07742*, 2015.

[6] Rong Chen, Haibo Chen, and Binyu 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*, PACT '10, pages 523–534, 2010.

[7] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48. ACM, 2004.

[8] D.Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. In *Intel Performance Analysis Guide*, 2009.

[9] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak 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*, ASPLOS XVII, pages 37–48, 2012.

[10] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. Characterizing data analysis workloads in data centers. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 66–76, 2013.

[11] Zhen Jia, Jianfeng Zhan, Lei Wang, Rui Han, Sally A. McKee, Qiang Yang, Chunjie Luo, and Jingwei Li. Characterizing and subsetting big data workloads. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 191–201, 2014.

[12] Tao Jiang, Qianlong Zhang, Rui Hou, Lin Chai, Sally A. McKee, Zhen Jia, and Ninghui Sun. Understanding the behavior of in-memory computing workloads. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 22–30, 2014.

[13] Vasileios Karakostas, Osman S. Unsal, Mario Nemirovsky, Adrian Cristal, and Michael Swift. Performance analysis of the memory management unit under scale-out workloads. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 1–12, Oct 2014.

[14] Chunjie Luo, Jianfeng Zhan, Zhen Jia, Lei Wang, Gang Lu, Lixin Zhang, Cheng-Zhong Xu, and Ninghui Sun. Cloudrank-d: benchmarking and ranking cloud computing systems for data processing applications. *Frontiers of Computer Science*, 6(4):347–362, 2012.

[15] Zijian Ming, Chunjie Luo, Wanling Gao, Rui Han, Qiang Yang, Lei Wang, and Jianfeng Zhan. BDGS: A scalable big data generator suite in big data benchmarking. In *Advancing Big Data Benchmarks*, volume 8585 of *Lecture Notes in Computer Science*, pages 138–154. 2014.

[16] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, 2015.

[17] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. Bigdatabench: A big data benchmark suite from internet services. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA*, pages 488–499, 2014.

[18] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, pages 35–44, 2014.

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

[20] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, pages 198–207, 2009.

[21] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion 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)*, pages 15–28, San Jose, CA, 2012. ISBN 978-931971-92-8.

[22] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 183–193. ACM, 2015.

Publication C

Architectural Impact on Performance of In-memory Data Analytics: Apache Spark Case Study.
Ahsan Javed Awan, Mats Brorsson, Vladmir Vlassov and Eduard Ayguade.
(submitted).

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

Ahsan Javed Awan[1], Mats Brorsson[1], Vladimir Vlassov[1] and Eduard Ayguade[2]

[1]KTH Royal Institute of Technology,
Software and Computer Systems Department(SCS),
{ajawan,matsbror,vladv}@kth.se
[2]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

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 data sets and that modern scale-up servers are adequate to run analytics jobs [11]. 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 [33], Ostrich [15] and Polymer [36] 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 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 [12, 19] 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.

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. 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.

2.2 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.

2.3 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.

2.4 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.

2.5 Spark Streaming

Spark Streaming [35] 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.

2.6 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.

2.7 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 [29]. Moder scale-up servers like Ivy Bridge machines are also part of Google data centers as recent study [23] 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 [6] to improve the memory and CPU performance of Spark applications on modern servers

Our recent efforts on identifying the bottlenecks in Spark [12, 19] 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?

3 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.

3.1 Workloads

This study uses batch processing and stream processing workloads, described in Table 1 and Table 2 respectively. Benchmarking big data analytics is an open research area, we however chose the workloads carefully. Batch processing workloads are a subset BigdataBench [30] and HiBench [17] which are highly referenced benchmark suites in Big data domain. Stream processing workloads used in the paper are super set of StreamBench [25] and also cover the solution patterns for real-time streaming analytics [28].

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 [26]) 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 [27].

3.2 System Configuration

Table 3 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 [10]. With Hyper-Threading and Turbo-boost disabled, there are 24 cores in the system operating at the frequency of 2.7 GHz.

Table 4 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 [12]. 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 [3] to perform general micro-architecture exploration and to collect hardware performance counters. We use numactl [5] to control the process and memory allocation affinity to a particular socket. We use hwloc [14] to get the CPU ID of hardware threads. We use msr-tools [4] to read and write model specific registers (MSRs).

Table 1: 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	

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 [31] 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

Table 2: Stream Processing Workloads

Workload	Description	Input data stream
Streaming Kmeans (Skm)	uses streaming version of K-Means clustering algorithm from Spark Mllib.	Numerical Records
Streaming Linear Regression (Slir)	uses streaming version of Linear Regression algorithm from Spark Mllib.	
Streaming Logistic Regression (Slogr)	uses streaming version of Logistic Regression algorithm from Spark Mllib.	
Network Word Count (NWc)	counts the number of words in text,received from a data server listening on a TCP socket every 2 sec and print the counts on the screen. A data server is created by running Netcat (a networking utility in Unix systems for creating TCP/UDP connections)	Wikipedia data
Network Grep (Gp)	counts how many lines,have the word the in them every sec and prints the counts on the screen.	
Windowed Word Count (WWc)	generates every 10 seconds, word counts over the last 30 sec of data received on a TCP socket every 2 sec.	
Stateful Word Count (StWc)	counts words cumulatively in text received from the network every sec starting with initial value of word count.	
Sql Word Count (SqWc)	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.	

Table 3: 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	

bottlenecks [19,23,32]. 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 that for each CPU core, there are four pipeline slots available per clock cycle. At issue point each pipeline slot is classified into one of four base categories: Front-end Bound, Back-end Bound, Bad Speculation and Retiring. If a micro-operation is issued in a given cycle, it would eventually either get retired or 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 5, whose definition are taken from Intel Vtune on-line help [3].

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		

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.

4 Evaluation

4.1 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. 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).

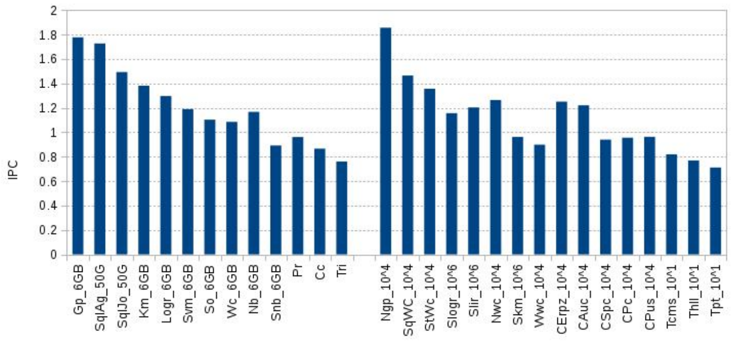
CErpz and CAuc both use "window", "map" and "groupbyKey" transformations but the front-end bound fraction and icache miss imapct in CAuc is larger than in CErpz. However, back-end bound fraction, DRAM bound stalled cycles, memory bandwidth consumption are larger in CErpz than in CAuC. The retiring fraction 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.

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

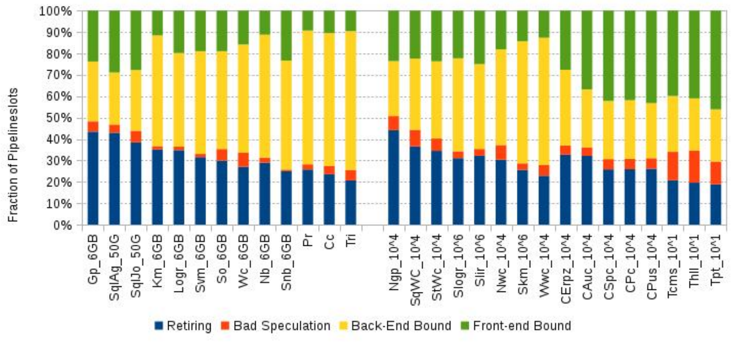
In order to answer the question, we compare the micro-architectural characterization of stream processing workloads at input data rates of 10, 100, 1000 and 10,000 events per second. Figure 2a shows that CPU utilization increases only modestly up-to 1000 events/s after which it increases up-to 20%. Like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

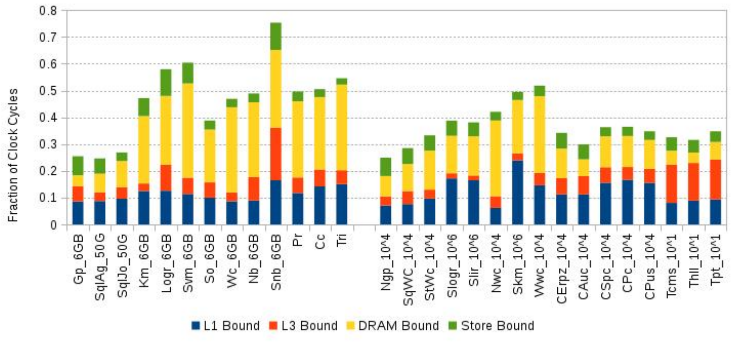
The memory subsystem stalls break down in Figure 2d show that L1 bound stalls increase, L3 bound stalls decrease and DRAM bound stalls increase at high data input rate, e.g in CErpz, L3 bound stall and DRAM bound stalls remain roughly constant at 10, 100 and 1000 events/s because the working sets are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



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

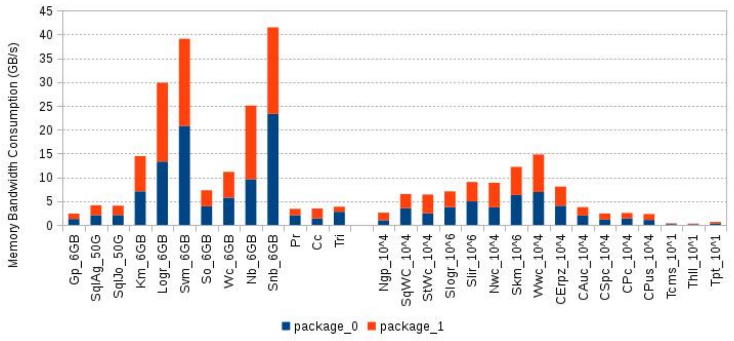


(b) Majority of stream processing workloads are back-end bound as that of batch processing workloads

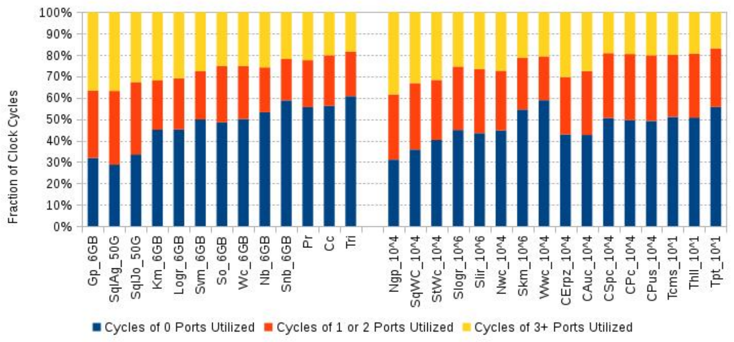


(c) Stream processing workloads are also DRAM bound but their fraction of DRAM bound stalled cycles is lower than that of batch processing workloads

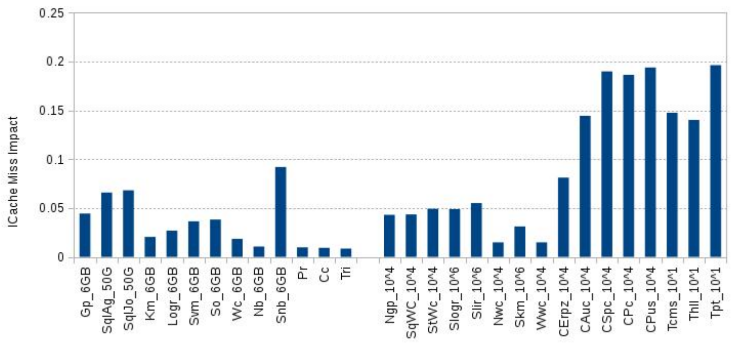
Figure 1: Comparison of micro-architectural characteristics of batch and stream processing workloads



(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

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

Figure 2f, where the memory bandwidth consumption is constant at 10, 100 and 1000 events/s and then increases significantly at 10,000 events/s. Larger working sets translate into better utilization of functional units as the number of clock cycles during which no ports are utilized decrease at higher input data rates. Hence input data rates should be high enough to provide working sets large enough to keep the execution units busy.

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

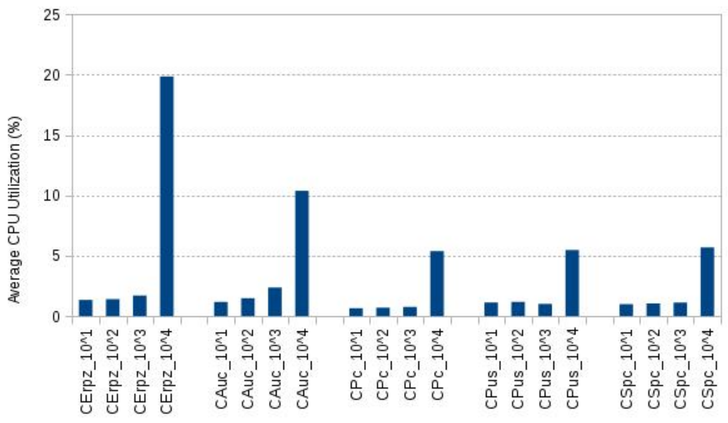
Ivy Bridge Server is a NUMA multi-socket system. Each socket has 2 on-chip memory controllers and a part of the main memory is directly connected to each socket. This layout offers high bandwidth and low access latency to the directly connected part of the main memory. The sockets are connected by two 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 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 6. Equation 1 and 2 in Appendix give the formulae for fraction of clock cycles, CPU stalled on local DRAM and remote DRAM respectively.

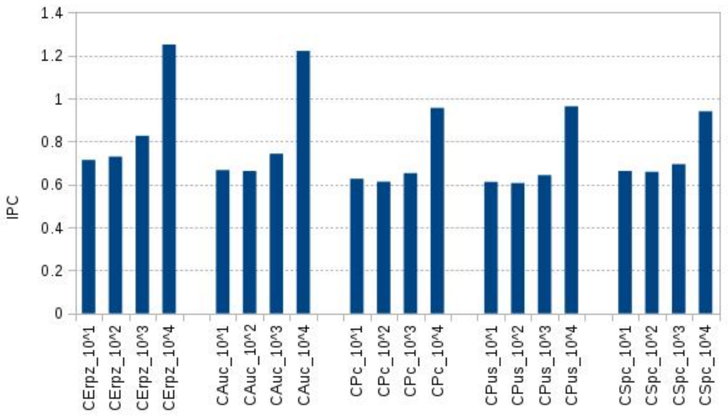
Figure 3a shows remote memory accesses can degrade the performance of Spark workloads by 10% on average. This is because despite the stalled cycles on remote memory accesses double (see Figure 3c), retiring category degrades by only 8.7%, Back-end bound stalls increases by 19.45%, bad speculation decreases 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).

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

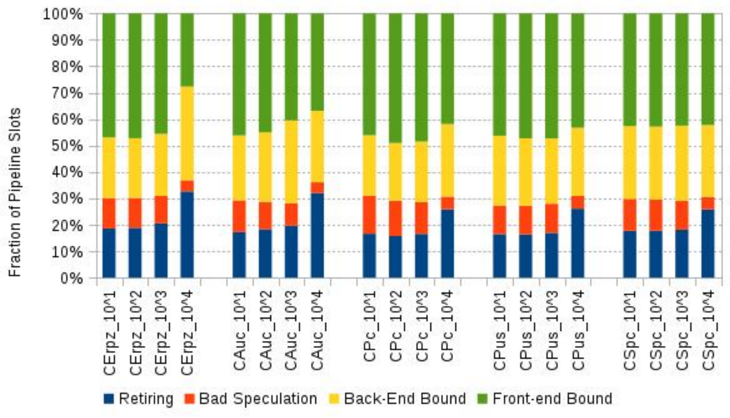
Ivy Bridge Machine uses Simultaneous Multi-threading(SMT), which enables one processor core to run two software threads simultaneously to hide data access latencies. To evaluate the effectiveness of Hyper-Threading, we run Spark process in the three different configurations a) ST:2x1, the base-line single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



(a) CPU utilization increases with data velocity

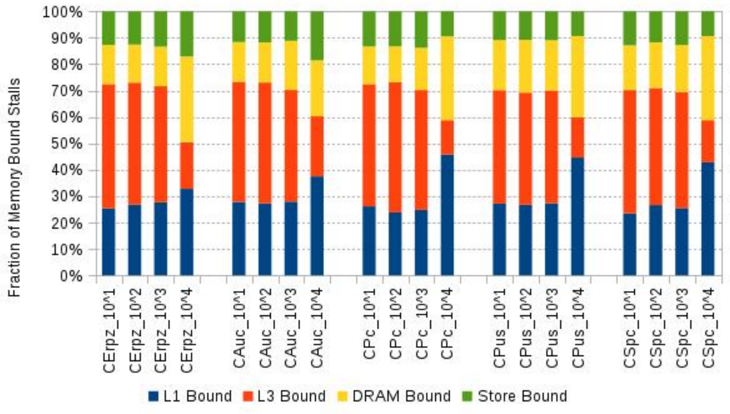


(b) Better IPC at higher data velocity

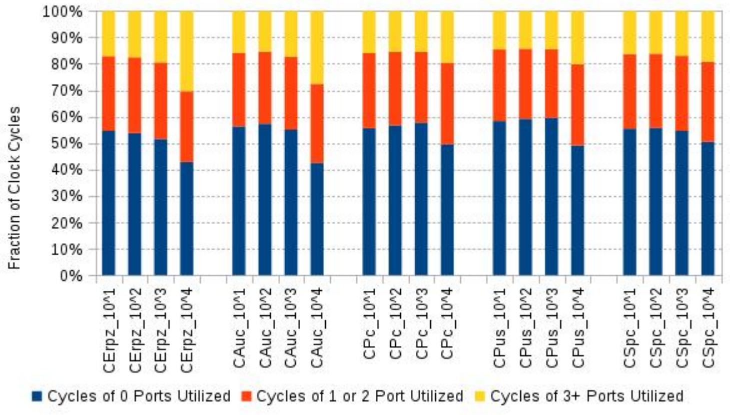


(c) Front-end bound stalls decrease and fraction of retiring slots increases with data velocity

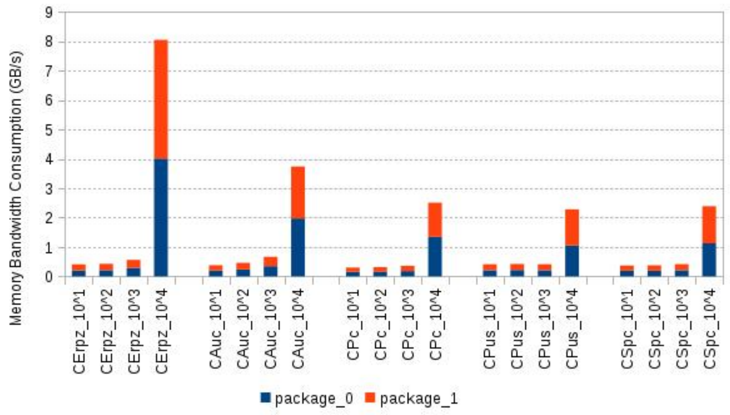
Figure 2: Impact of Data Velocity on Micro-architectural Performance of Spark Streaming Workloads



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

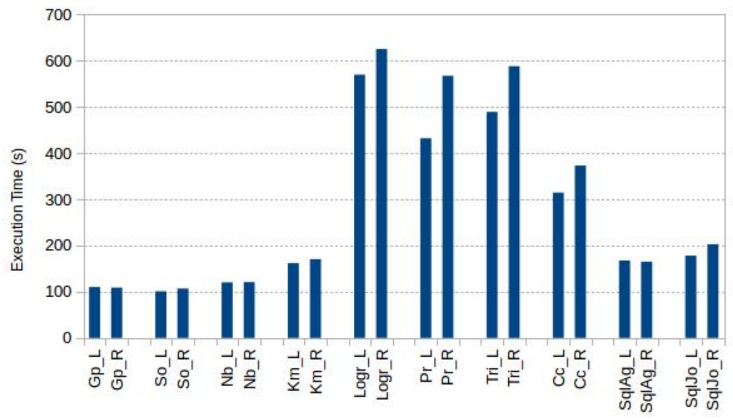


(e) Functional units inside exhibit better utilization at higher data velocity

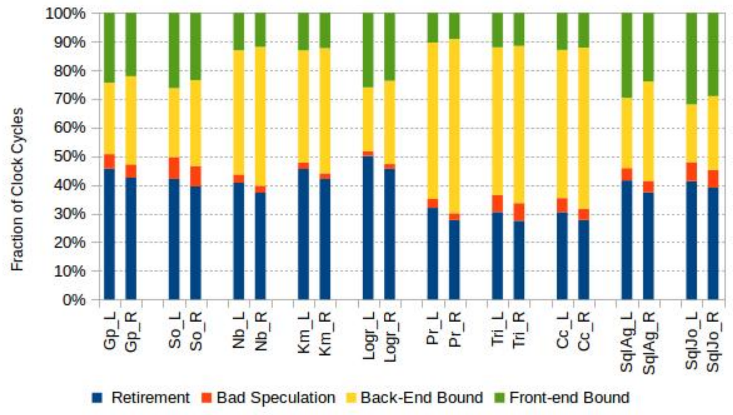


(f) Memory bandwidth consumption increases with data velocity

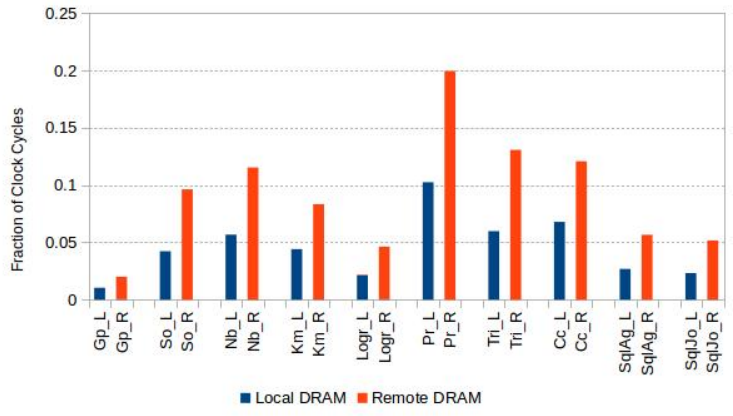
Figure 2: Impact of Data Velocity on Micro-architectural Performance of Spark Streaming Workloads



(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.

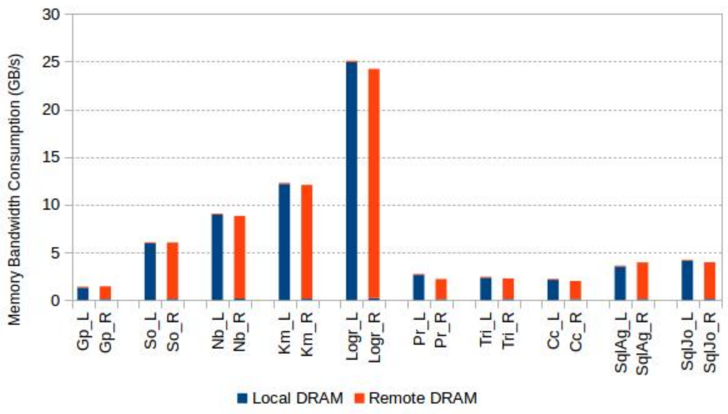


(c) Stalled Cycles doubles in remote memory case

Figure 3: NUMA Characterization of Spark Benchmarks

Table 6: 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



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

Figure 3: NUMA Characterization of Spark Benchmarks

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 7. 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

Table 7: 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

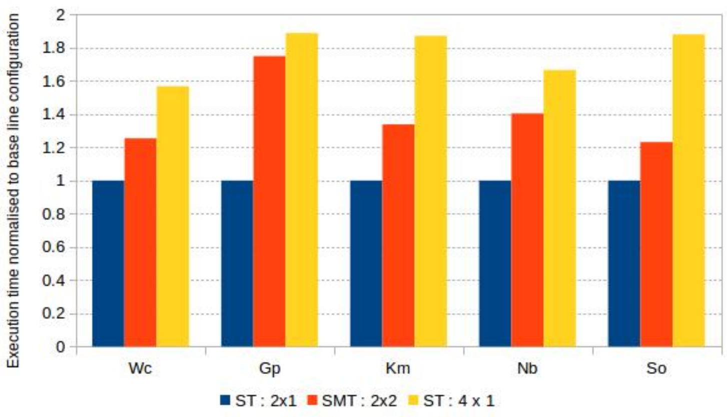
average across the workloads 4c. Figure 4b presents HT Effectiveness at different baseline configurations. HT Effectiveness of 1 is desirable as it implies 30% performance improvement in Hyper-Threading case over the baseline single threaded configuration [2]. Equation 3 in Appendix gives the formula 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 [19]

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

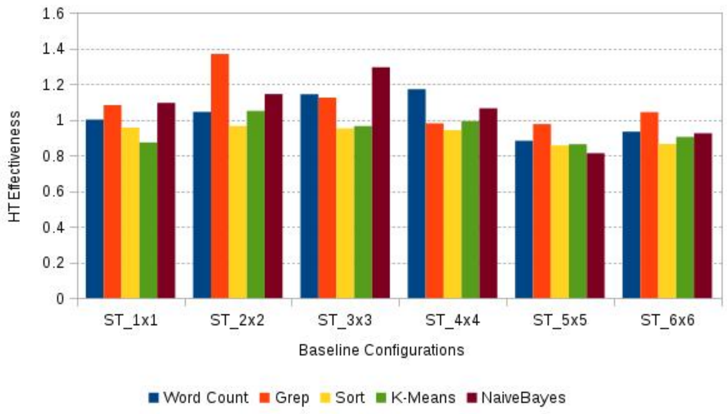
4.5 Are existing hardware prefetchers in modern scale-up servers effective for in-memory data analytics?

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 8. This information is taken from Intel software forum [1].

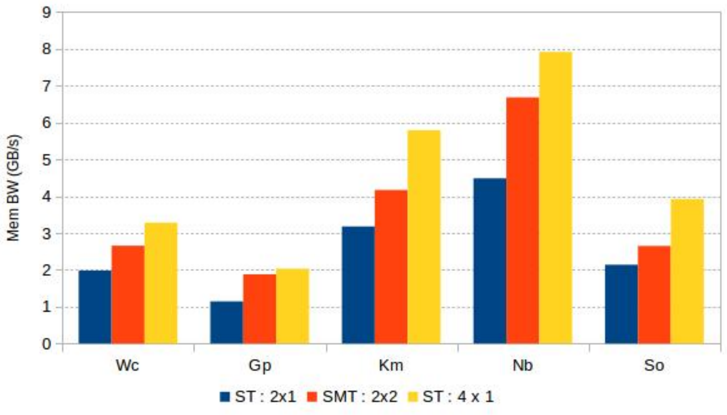
To evaluate the effectiveness of L1-D prefetchers, we measure L1-D miss impact for the benchmarks at four configurations: a) all processor prefetchers are enabled, b) DCU prefetcher is disabled only, c) DCU IP prefetcher is disabled only and d) both L1-D prefetchers are disabled. To assess the effectiveness 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.



(a) Multi-core vs Hyper-Threading

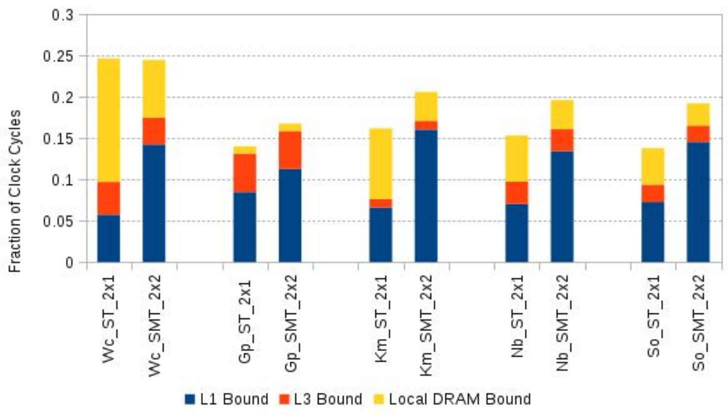


(b) HT Effectiveness is around 1



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

Figure 4: Hyper Threading is Effective



(d) DRAM Bound decreases and L1 Bound increases

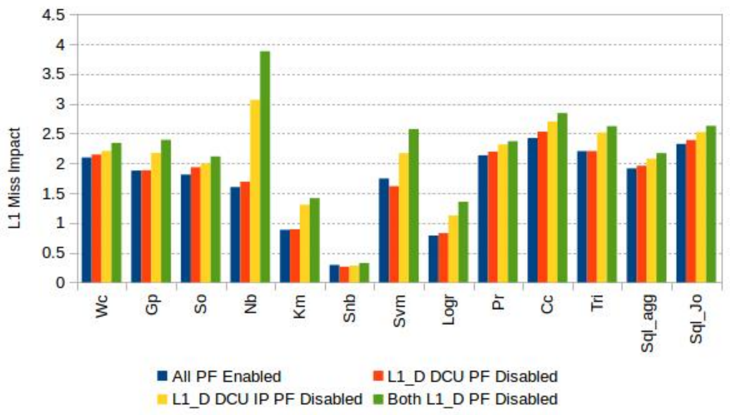
Figure 4: Hyper Threading is Effective

Table 8: Hardware Prefetchers Description

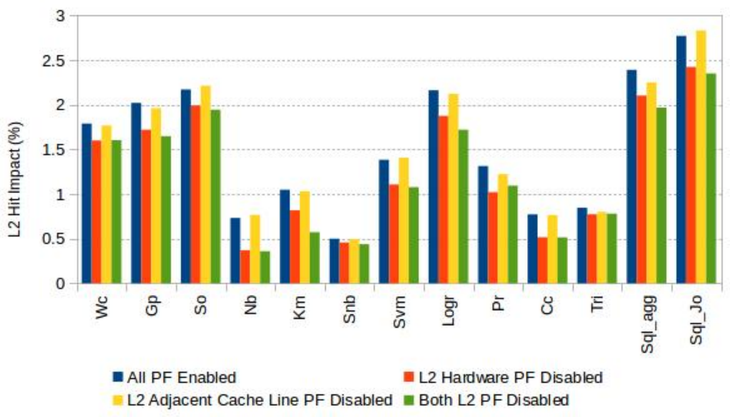
Prefetcher	**Bit No. in MSR (0x1A4)**	**Description**
L2 hardware prefetcher	0	Fetches additional lines of code or data into the L2 cache
L2 adjacent cache line prefetcher	1	Fetches the cache line thatcomprises a cache line pair(128 bytes)
DCU prefetcher	2	Fetches the next cache line into L1-D cache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

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.

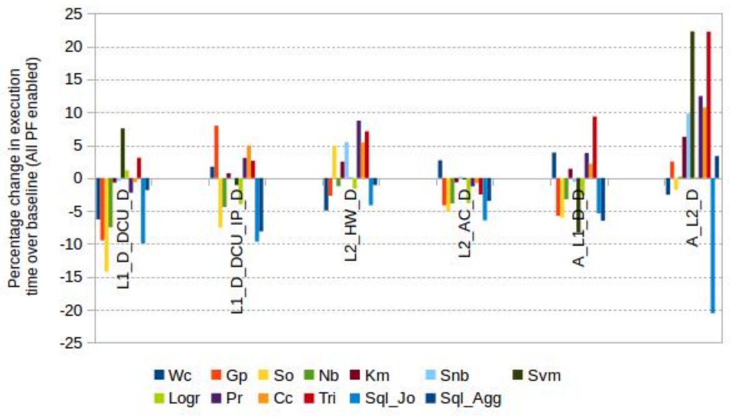
Figure 5b shows that L2 hit impact decreases by 18% on average across the



(a) L1-D DCU Prefetcher is ineffective



(b) Adjacent Cache Line L2 Prefecher is ineffective



(c) Disabling L1-D next-line and L2 Adjacent Cache Line Prefetchers can reduce the execution of Spark jobs up-to 14% and 4% respectively

Figure 5: Evaluation of Hardware Prefetchers

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 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.

4.6 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 [18], 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 [10], 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 [9] 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.

Figure 6 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 7,it can be seen that even when the peak bandwidth utilization

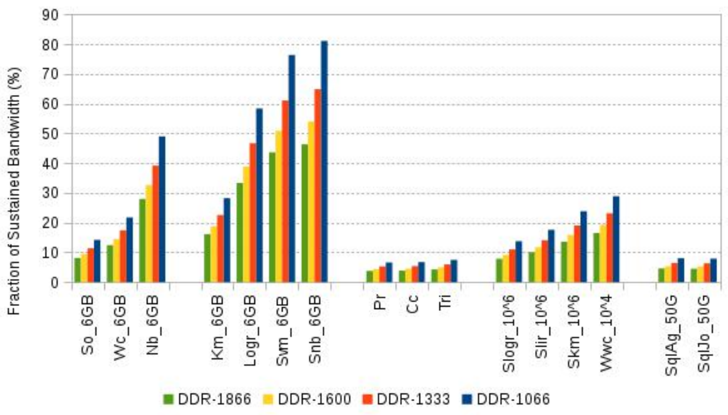


Figure 6: Spark workloads dont experience loaded latencies

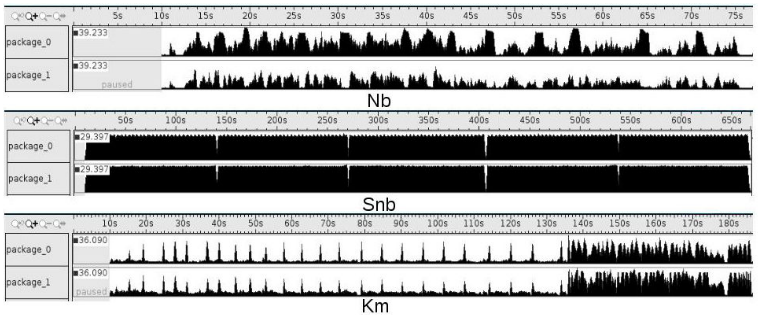


Figure 7: Bandwidth Consumption over time

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.

4.7 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

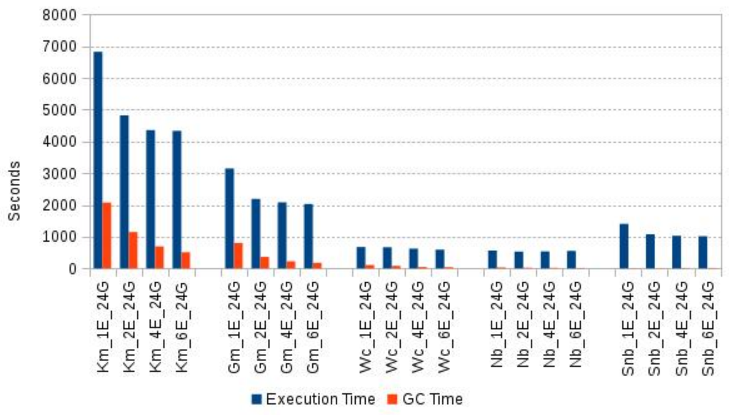


Figure 8: Multiple small executors are better than single large executor due to reduction in GC time

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 9: 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 8 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

5 Related Work

Several studies characterize the behaviour of big data workloads and identify the mismatch between the processor and the big data applications [16, 20–22, 24, 30, 32]. However these studies lack in identifying the limitations of modern scale-up servers for Spark based data analytics. Ferdman et al. [16] 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. [37] 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 [20]. They also suffer from notable from end stalls but L2 and L3 caches are effective for them. Wang et al. [30] conclude the same about L3 caches and L1 I Cache miss rates despite using larger data sets. Deep dive analysis [32] 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. [29] have shown that NUMA has significant impact on Gmail backend and web.search frontend. Researchers at IBM's Spark technology center [8] has only explored the thread affinity, bind only JVMs to sockets but does not limit the cross socket accesses. Beamer et al. [13] 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. [23] 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 [32], the same conclusion could not be translated about Spark workloads as previous work shows that as memory access characteristics of Spark and Hadoop differ [21] and software stacks have significant impact on the micro-architecture behaviour of big data workloads [21]. 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 [16]. We show for the first time the they have negative impact on the performance of in-memory data analytics with Spark.

6 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.

7 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.

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

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.

$$\begin{aligned}&Remote\ DRAM\ Bound = (310 * MEM\_LOAD\_UOPS\_\\&LLC\_MISS\_RETIRED.REMOTE\_DRAM)/\\&CPU\_CLK\_UNHALTED.THREAD\end{aligned} \tag{2}$$

Equation 3 gives the formula for $HT\ Effectiveness$, which is taken from Intel's on-line forum [2]. $HT\ Scaling_{obs}$ 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.

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

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.

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

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.

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

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

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

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

Bibliography

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

[4] msr-tools. `https://01.org/msr-tools`.

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

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

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

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

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

[10] 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`.

[11] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony I. T. Rowstron. Scale-up vs scale-out for hadoop: time to rethink? In *ACM Symposium on Cloud Computing, SOCC*, page 20, 2013.

[12] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. *Big Data Benchmarks, Performance Optimization, and Emerging Hardware: 6th Workshop, BPOE 2015, Kohala, HI, USA, August 31 - September 4, 2015. Revised Selected Papers*, chapter How Data Volume Affects Spark Based Data Analytics on a Scale-up Server, pages 81–92. Springer International Publishing, 2016.

[13] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 56–65. IEEE, 2015.

[14] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond 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*, pages 180–186. IEEE, 2010.

[15] Rong Chen, Haibo Chen, and Binyu 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*, PACT '10, pages 523–534, 2010.

[16] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak 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, ASPLOS XVII, pages 37–48, 2012.

[17] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51, 2010.

[18] Bruce Jacob. 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):1–77, 2009.

[19] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard 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*, pages 1–8. IEEE, 2015.

[20] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. Characterizing data analysis workloads in data centers. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 66–76, 2013.

[21] Zhen Jia, Jianfeng Zhan, Lei Wang, Rui Han, Sally A. McKee, Qiang Yang, Chunjie Luo, and Jingwei Li. Characterizing and subsetting big data workloads. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 191–201, 2014.

[22] Tao Jiang, Qianlong Zhang, Rui Hou, Lin Chai, Sally A. McKee, Zhen Jia, and Ninghui Sun. Understanding the behavior of in-memory computing workloads. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 22–30, 2014.

[23] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, David Brooks, Simone Campanoni, Kevin Brownell, Timothy M Jones, *et al.* Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 158–169. ACM, 2015.

[24] Vasileios Karakostas, Osman S. Unsal, Mario Nemirovsky, Adrian Cristal, and Michael Swift. Performance analysis of the memory management unit under scale-out workloads. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 1–12, Oct 2014.

[25] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 69–78. IEEE, 2014.

[26] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, *et al.* Mllib: Machine learning in apache spark. *arXiv preprint arXiv:1505.06807*, 2015.

[27] Zijian Ming, Chunjie Luo, Wanling Gao, Rui Han, Qiang Yang, Lei Wang, and Jianfeng Zhan. BDGS: A scalable big data generator suite in big data benchmarking. In *Advancing Big Data Benchmarks*, volume 8585 of *Lecture Notes in Computer Science*, pages 138–154. 2014.

[28] Srinath Perera and Sriskandarajah Suhothayan. Solution patterns for realtime streaming analytics. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 247–255. ACM, 2015.

[29] Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. Optimizing google's warehouse scale computers: The numa experience. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 188–197. IEEE, 2013.

[30] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. Bigdatabench: A big data benchmark suite from internet services. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA*, pages 488–499, 2014.

[31] Ahmad 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.

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

[33] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, pages 198–207, 2009.

[34] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion 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)*, pages 15–28, San Jose, CA, 2012. ISBN 978-931971-92-8.

[35] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion 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*, pages 10–10. USENIX Association, 2012.

[36] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 183–193. ACM, 2015.

[37] Chen Zheng, Jianfeng Zhan, Zhen Jia, and Lixin 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.