UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Performance characterization and optimization of in-memory data analytics on a scale-up server

Ahsan Javed Awan

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

AHSAN JAVED AWAN

Doctoral Thesis in Computer Architecture
Universitat Politècnica de Catalunya
Departament d'Arquitectura de Computadors,
Barcelona, Spain 2017
and
Doctoral Thesis in Information and Communication Technology
KTH Royal Institute of Technology
School of Information and Communication Technology
Stockholm, Sweden 2017

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

TRITA-ICT 2017:23
ISBN: 978-91-7729-584-6

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i informations och kommunikationsteknik fredagen den 15 december 2017 klockan 09.00 i Sal-C, Electrum, Kungl Tekniska högskolan, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

Joint Thesis Defence Committee

The joint thesis defence committee comprises following members

- **Moderator:**
  Assoc. Prof. Jim Dowling, KTH Royal Institute of Technology, Sweden.
- **Opponent:**
  Prof. Lieven Eckhout, Ghent University, Belgium.
- **Grading Board for both KTH and UPC:**
  Prof. Babak Falsafi, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland.
  Prof. Dimitrios S. Nikolopoulos, Queen's University of Belfast, Northern Ireland, UK.
  Prof. Avi(Abraham) Mendelson, Technion Israel University of Technology, Israel.
- **External Reviewers for UPC:**
  Prof. Avi(Abraham) Mendelson, Technion Israel University of Technology, Israel.
  Assoc. Prof. Johnny Öberg, KTH Royal Institute of Technology, Sweden.
- **Advance Reviewer for KTH:**
  Assoc. Prof. Johnny Öberg, KTH Royal Institute of Technology, Sweden.

Abstract

The sheer increase in the 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 to 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 (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). We have also found that workloads are bound by the latency of frequent data accesses to the memory. By enlarging input data size, application performance degrades significantly due to the 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 garbage collection impact, we match memory behavior with the garbage collector to improve the performance of applications between 1.6x to 3x. and recommend using multiple small Java virtual machines (JVMs) that can provide up-to 36% reduction in execution time over single large JVM. Based on the characteristics of workloads, the thesis envisions near-memory and near storage hardware acceleration to improve the single-node performance of scale-out frameworks like Apache Spark. Using modeling techniques, it estimates the speed-up of 4x for Apache Spark on scale-up servers augmented with near-data accelerators.

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 testprogram 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.

Abstracto

El gran aumento del volumen de datos en la última década (conocido como Big Data) ha desencadenado un gran esfuerzo de investigación en el marco de la computación en clúster, permitiendo a las empresas de Internet extraer conocimiento de dicho Big Data. Actualmente Apache Spark representa el estado del arte en cuanto a plataformas de análisis de Big Data (i) por su capacidad de explotar la computación data-flow con datos in-memory, y (ii) por un rendimiento y escalado con el numero de procesadores adecuado en las arquitecturas comerciales actuales. Sin embargo, poco esfuerzo se ha dedicado a entender dicho rendimiento y justificar los principales factores que estarían limitándolo en los servidores escalables de hoy en día. Esta tesis caracteriza el rendimiento de estos entornos de análisis de datos en memoria basados en Spark.

A través de una evaluación empírica, utilizando cargas de trabajo de referencia y representativas, en un servidor de doble socket, se ha concluido una falta de escalabilidad más allá de los 12 núcleos, básicamente debida a desequilibrios de la carga de trabajo a nivel de threads (flujos de ejecución), al aumento del tiempo de ejecución de cada core respecto a la ejecución secuencial, y la latencia de los accesos de los datos que residen en la memoria DRAM. Al aumentar el tamaño de los datos de entrada, el rendimiento de la aplicación se reduce significativamente debido al aumento en el tiempo de espera durante las operaciones de E/S y el garbage collector (GC), a pesar que la finalización de las instrucciones mejora en un 10% (debido a una menor tasa de fallo de la memoria caché L1 y mayor utilización de los núcleos del procesador).Para tolerar la latencia en los accesos a los datos, se ha concluido que el multi-threading simultáneo resulta eficaz y que (i) la mejora de la localidad a nivel de nodos NUMA puede mejorar el rendimiento en un 10% en promedio y (ii) deshabilitar la pre-búsqueda de datos a nivel de L1-D puede reducir el tiempo de ejecución del orden del 14%. En cuanto al impacto del GC, se observa que es posible mejorar el rendimiento de las aplicaciones en un factor entre 1.6x y 3x a base de utilizar múltiples pequeños ejecutores en vez de un único gran ejecutor.

En base a los resultados y análisis realizados en las cargas de trabajo utilizadas, la tesis doctoral analiza las posibilidades de utilizar unidades de cálculo cercanas a la memoria (near-memory) y a los dispositivos de almacenamiento (near-storage), mejorando así el rendimiento de los nodos que se utilicen para la ejecución de Apache Spark. Se presenta un modelo que estima una mejora de 4x en el rendimiento de Apache Spark usando aceleradores hardware cercanos a los datos.

To my grandfather, "Attah Muhammad" and my grandmother, "Azam Attah" whom I lost during this PhD

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 Ph.D. students; Roberto Castañeda Lozano for providing the thesis template, Georgios Varisteas, Artur Podobas, Vamis Xhagjika, Navaneeth Rameshan and Muhammad Anis-ud-din Nasir for discussions on my work and to Ananya Muddukrishna and Artur Podobas for improving the first draft of my papers. With-out Thomas Sjöland taking care of the financial aspects and Sandra Gustavsson Nylén, Susy Mathew, Madeleine Prints looking after the administration stuff, it would not have been a smooth journey. I also thank Jim Dowling for reviewing the Licentiate thesis and acting as the internal quality controller. I am also obliged to Christian Schulte, David Broman, Seif Haridi and Sverker Jansson for their feedback on my progress. I also acknowledge Moriyoshi Ohara, Kazuaki Ishizaki, Lydia Chen, Frank Liu, Thomas Parnell and Peter Hofstee from IBM Research. Finally, I am grateful to Marcel Van De Burgwal in particular and Recore Systems in general for providing me both technical input and FPGA device to finish the research work. Johnny Öberg and Ingo Sander from Embedded Systems, Dept, KTH are also deeply appreciated for providing access to Xilinx Tool Chain. I am also obliged to Petar Radojković, David Carrera Perez, and Jordi Torres Vinals from UPC/BSC for providing detailed feedback on the initial draft of the thesis.

This work was supported by Erasmus Mundus Joint Doctorate in Distributed processing (EMJD-DC) program funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission. It was also supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology through the TIN2015-65316-P project and by the Generalitat de Catalunya (contract 2014-SGR-1051). European Network on High Performance and Embedded Architecture and Compilation (HiPEAC) supported the research through it's Industrial Ph.D. Mobility Programme and collaboration grants. Databricks supported the research in disseminating the results to the industry by granting academic passes and invited talks to its industrial events.

Contents

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 those modern scale-up servers are adequate to run analytics jobs [21]. Additionally, commonly used predictive analytics such as machine learning algorithms, work on filtered datasets that easily fit into the memory of modern scale-up servers. Moreover, the today's scale-up servers can have CPU, memory and persistent storage resources in abundance at affordable prices. Thus we envision a small cluster of scale-up servers to be the preferable choice of enterprises in near future.

While Phoenix [188], Ostrich [40] and Polymer [195] 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 [190] is getting popular in the industry because it enables in-memory processing, scales out to a large number of commodity machines and provides a unified framework for batch and stream processing of big data workloads. 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. The scale-up server used for characterization studies in thesis is shown in Fig 1.1

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.

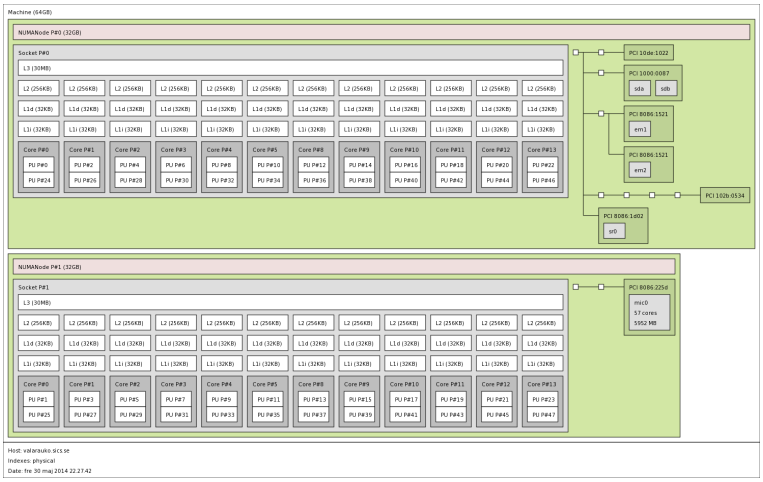


Figure 1.1: Scale-up Server used for Characterization Studies in the thesis

- 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 Spark-core, Spark Mllib, Spark SQL, GraphX and Spark Streaming. We quantify the impact of data velocity on the 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.
- We study which aspect of Near-Data Processing (in-storage processing, processing in memory) suits better the characteristics of Apache Spark workloads. To answer this, we characterize Apache Spark workloads into compute bound, memory bound and I/O bound. We use hardware performance counters to identify the memory bound applications and OS level metrics like CPU utilization, idle time and wait time on I/O to filter out the I/O bound applications in Apache Spark.

The thesis is based on following publications. Chapter 2 discusses background information and related work. Chapter 3, 4, 5, 6 and 7 are the reformatted versions

of published papers. Chapter 8 covers the unpublished work. Conclusions and future work are presented in Chapter 9.

1.1 List of Publications

- **Chapter 3:** [**24**] 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)

- **Chapter 4:** [**25**] 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.

- **Chapter 5:** [**26**] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Micro-architectural Characterization of Apache Spark on Batch and Stream Processing Workloads", in 6th IEEE International Conference on Big Data and Cloud Computing (BDCloud), Atlanta, USA, 2016.

- **Chapter 6:** [**27**] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "Node Architecture Implications for In-Memory Data Analytics on Scale-in Clusters" in 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT), Shanghai, China, 2016

- **Chapter 7:** [**28**] A. J. Awan, E. Ayguade, M. Brorsson, M. Ohara, K. Ishizaki, V. Vlassov, "Identifying the Potential of Near Data Processing for Apache Spark" in ACM International Symposium on Memory Systems, Washington, D.C., USA, 2017.

The individual contribution of authors is asunder,

- **Ahsan Javed Awan:** contributes to, literature review, problems identification, hypothesis formulation, experiment design, data analysis and paper writing.

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

- **Mats Brorsson:** contributes to, problem selection and feedback on experiment design, results, conclusions, and draft of the paper.

- **Moriyoshi Ohara:** contributes to, problem selection and feedback on the draft of the paper.

- **Kazuaki Ishizaki:** contributes to, problem selection and feedback on the draft of the paper.
- **Vladimir Vlassov:** contributes to, problem selection and feedback on the draft of the paper.

1.2 Chapter Highlights

Chapter 3:

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 an 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

Chapter 4:

This chapter augments chapter 3 by quantifying the impact of data volume on the performance of in-memory data analytics with Spark on scale-up servers. In this chapter, 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 the 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 the substantial increase in garbage collection and file I/O time.
- Without 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.

Chapter 5

The scope of previous two chapters is limited to batch processing workloads only, assuming that Spark streaming would have same micro-architectural bottlenecks. We revisit this assumption in chapter 5.

- Does micro-architectural performance remain consistent across batch and stream processing data analytics?
- How does data velocity affect the micro-architectural performance of in-memory data analytics with Spark?

The key insights are:

- Batch processing and stream processing has same micro-architectural behavior 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.

Chapter 6

Simultaneous multi-threading and hardware prefetching 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.

- 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?

The key insights are:

- Exploiting data locality on NUMA nodes can only reduce the job completion time by 10% on average as it reduces the back-end bound stalls by 19%, which improves the instruction retirement only by 9%.
- Hyper-Threading is effective to reduce DRAM bound stalls by 50%, HT effectiveness is 1.
- Disabling next-line L1-D and Adjacent Cache line L2 prefetchers can improve the performance by up to 14% and 4% respectively.

- Spark workloads do not experience loaded latencies and it is better to lower down the DDR3 speed from 1866 MT/s to 1333 MT/s
- Multiple small executors can provide up-to 36% speedup over single large executor.

Chapter 7:

The concept of near-data processing (NDP) is regaining the attention of researchers partially because of technological advancement and partially because moving the compute closer to the data where it resides, can remove the performance bottlenecks due to data movement. The umbrella of NDP covers 2D-integrated Processing-In-Memory, 3D-stacked Processing-In-Memory (PIM) and In-Storage Processing (ISP). Existing studies show efficacy of processing-in-memory (PIM) approach for simple map-reduce applications [83, 137], graph analytics [15, 127], machine learning applications [**?**, 31] and SQL queries [125, 177]. Researchers also show the potential of processing in non-volatile memories for I/O bound big data applications [36, 143, 173]. However, it is not clear which aspect of NDP (high bandwidth, improved latency, reduction in data movement, etc..) will benefit state-of-art big data frameworks like Apache Spark. Before quantifying the performance gain achievable by NDP for Spark, it is pertinent to answer which form of NDP (PIM, ISP) would better suit Spark workloads?

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

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

Chapter 8:

Traditionally, cluster computing frameworks like Apache Flink [35], Apache Spark [190], Apache Storm [164] etc, are being increasingly used to run real-time streaming analytics. These frameworks have been designed to use the cluster of commodity

machines. Keeping in view the poor multi-core scalability of such frameworks [27], we hypothesize that coherently attached processor interface (CAPI) [161] based scale-up machines can deliver enhanced performance for in-memory big data analytics.

Our contributions are

- We propose system design for FPGA acceleration of big data processing frameworks on CAPI based scale-up servers.
- We estimate 4x speedup in the scale-up performance of Apache Spark on CAPI based scale-up machines using roof-line model.

1.3 Thesis Statement

Scale-out big data processing frameworks fail to fully exploit the potential of modern off-the-shelf commodity machines (scale-up servers) and require modern servers to be augmented with programmable accelerators near-memory and near-storage.

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 [146].

2.1 Horizontally Scaled Systems

MapReduce [51] 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 [57, 110, 147]. Sakr et al. [147] provide a comprehensive survey of 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. [57] 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 process-

ing can be found in [75] and [29] respectively. Spark [190] provides a unified framework for batch and stream processing [191]. Graph processing [183], predictive analytics using machine learning approaches [122] and SQL query analysis [184] is also supported in Spark.

Spark

Spark is a cluster computing framework that uses Resilient Distributed Datasets (RDDs) [190], 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 [191] 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 as Kafka, Twitter, or TCP sockets. It then divides the data into batches, which are further 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 [21,40,104,105,130,144,145, 155,158,162,188,195], GPUs [60,63,77,86,140], the coupled CPU-GPU architecture [38,101], FPGA [54,95,152], Xeon Phi co-processor [117,118] and Cell processors [50].

2.3 GPU based Heterogeneous Clusters

Shirahata et al. [154] 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 [74] addresses the problem of integrating GPUs into existing MapReduce framework (Hadoop). OpenCL with Hadoop has been proposed in [131,182] for the same problem. Zhai et al. [192] 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 [197].

El-Helw et al. [59] 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. [160] propose standalone MapReduce library written in C++ and CUDA for GPU clusters. Xie et al. propose Moim [178] 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. [72] 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 [171, 172].

Shirahata et al. [153] argue that the capacity of device memory on GPUs limits the size of the 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. [46] 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

A detailed survey on re-configurable accelerators for cloud computing [96] outlines that in most of the cases, the speedup is low and emphasis has been given to the energy efficiency of the system. The speedup also depends on whether the proposed accelerator is used as a co-processor or it is used as a complete replacement for the processor. In the latter case, the speedup is usually higher since the whole application is running on the FPGA. Usually, FPGAs are used for the batch processing applications, where a large amount of data are offloaded to the FPGA for acceleration. In the co-processor case, then the overall energy efficiency is lower. In such cases, the interface is either PCIe offering a total throughput of 16 GB/s or the AXI4 bus that provides an aggregated throughput of 25.6 GB/s when clocked at 200 MHz. The latter case is when the accelerator is part of multiprocessor system-on-chip (MPSoC). In the former case, multiple FPGA cards can be added in the PCIe allowing easier scalability. In the cases that solution is proposed as a complete replacement for the typical processor, Ethernet is used for the reception and transmission of data packets and a TCP/IP-offload-engine

is used to speed up the processing requirements at the network level.

Acceleration on a CPU-FPGA Heterogeneous Platform

Acceleration using CAPI

Giefers et al. [68] study a fast fourier transform (FFT) accelerator on FPGA, attached via CAPI to a Power 8 processor and show that a coherently attached accelerator outperforms device driver based approaches in terms of latency. Experimental results show that bypassing the device driver significantly reduces the communication and control overhead for a PCIe attached co-processor. When using a zero copy mapping of device buffers, the FFT data is instantly passed from the host to the FFT kernel implementation of the FPGA via PCIe link. The streaming of data from host hampers performance so that the zero-copy version of the OpenCL [159] program only achieves 4Gflops. When the kernel can read and write the data from local SDRAM, the performance is much higher. As the accelerator function unit (AFU) sends and receives data through the PCIe link, the performance is not as high as for the DDR OpenCL kernel, but still more than 5x better than the zero-copy version of the OpenCL.

They also compare the latency of software and hardware accelerated FFT kernels. In all cases, the input data is initialized on the host and resides in the cache memory. On the CPU, 4k FFT kernel takes 47.5us on average. The zero copy version of the OpenCL kernel has poor performance and takes 539us. Copying the data to the accelerator memory improves runtime and reduces latency to 344us. Raw kernel operation without the data copy consumes 124us. The CAPI version of the 4k FFT accelerator takes 69us on average and with that, is 5x faster than the OpenCL runtime.

Lee et-al [106] propose ExtraV, a framework for near storage graph processing. It efficiently utilizes a cache-coherent hardware accelerator at the storage side to achieve performance and flexibility at the same time. ExtraV consists of four main components: 1) host processor, 2) main memory, 3) AFU (Accelerator Function Unit) and 4) storage. The AFU, a hardware accelerator, sits between the host processor and storage. Using a coherent interface that allows main memory accesses, it performs graph traversal functions that are common to various algorithms while the program running on the host processor (called the host program) manages the overall execution along with more application-specific tasks.

Acceleration using Intel Heterogeneous Architecture Research Platform (HARP) [82]

Zhang et al. [194] have implemented CNNs on CPU-FPGA platform with coherent shared memory. They exploit FFT and overlap-and-add to reduce the computational requirements of the convolution layer. They map the 2D convolver design on the FPGA using floating point numbers, propose a data layout in shared memory for efficient communication between the CPU and the FPGA. Their design employs double buffering to reduce the memory access latency and sustain the peak performance of the FPGA. They also exploit concurrent processing on the CPU and FPGA. They implement a fully connected layer on CPU using 16 threads as it can be viewed as matrix-vector multiplication, which is bounded by memory bandwidth. In other words, they exploit data parallelism of 2D-convolver and task parallelism to scale the overall system performance.

Abdelrahman et al. [14] present a case study of the design of an FPGA accelerator for a tightly-coupled shared-memory processor-FPGA system, They use K-means as an example of computationally-intensive application and design a pipe-lined accelerator for calculating minimum distances between points and centroids. The presence of a shared memory and the ability of the accelerator to directly read from and write to memory allowed a design in which data is only logically partitioned between concurrent CPU threads and the FPGA accelerator. This provides an improvement in performance over using only CPU threads or only the FPGA accelerator. Experimental evaluation shows that the combined use of accelerator and a single CPU thread results in 2.9x performance improvement over the CPU thread alone and 1.1x over the accelerator alone. With 4 CPU threads, the improvements are 1.6x and 1.9x respectively. Moreover, increased memory traffic has minimal impact on performance.

Zhang et al. [193] propose to speed up large-scale sorting using a CPU-FPGA heterogeneous platform. They optimize a fully pipe-lined merge sort based accelerator and employ several such designs working in parallel on FPGA. The partial results from the FPGA are then merged on the CPU. They target Intel HARP as the experimental platform and show improvement in the throughput by 2.9x and 1.9x compared to CPU-only and FPGA-only baselines. They employ divide and conquer based strategy to exploit task parallelism on FPGA and the thread parallelism through overlapping an FPGA computation. The divide and conquer strategy is based on the shared memory scheme on HARP such that it allows CPU and FPGA to manipulate data concurrently by continuous data transfer through quick point interconnect (QPI) [198]. The detailed strategy is as under, i) Divide: Break the whole sequence into K blocks. Each block contains M cache lines. ii) Acceleration: CPU continuously

sends blocks of data to FPGA while FPGA keeps sorting unit block data and send back to shared memory through QPI, iii) Conquer: As soon as CPU detects sorted data blocks in shared memory, it will start merging data blocks.

Chen et al. [39] alleviate the memory burden of sorting on FPGA by developing a hybrid CPU-FPGA based sorting design. Parallel bitonic sorting network based accelerators with flexible data parallelism are developed to exploit the massive parallelism on FPGA. Merge-sort tree-based design with less computation load is employed on the CPU. A decomposition-based task partition approach is proposed to partition the input data set into several sub data sets sorted by FPGA accelerators in parallel, and then the partial results are merged on the CPU. Based on this hybrid sorting design, they propose streaming join algorithms, which are tailored to the target hybrid platform by optimizing the classic CPU-based nested loop join and sort-merge join algorithms. They show that hybrid CPU-FPGA based design outperforms both CPU only and accelerator only approaches.

Weisz et al. [175] argue that FPGA acceleration platforms with direct coherent access to processor memory create an opportunity for accelerating applications with irregular parallelism governed by large in-memory pointer-based data structures. They use the simple reference behavior of a linked list traversal as a proxy to study the performance potentials of accelerating these applications on the shared-memory processor-FPGA system. The linked list is parameterized by node layout in memory, per-node data payload size, payload independence and travel concurrency to capture the main performance effects of the different pointer-based data structures and algorithms. The key results show: i) the FPGA fabric is least efficient when traversing a single list with non-sequential node layout and a small payload size; (ii) processor assistance can help alleviate this shortcoming and (iii) when appropriate, a fabric only approach that interleaves multiple linked list traversals is an effective way to maximize traversal performance. Irregular parallel applications operate on very large memory resident, pointer-based data structures (i.e. lists, trees, and graphs), Databases use tree-like structures to store indices for fast searches and combine information from different tables, Similarly machine learning algorithms in big data applications rely on graphs, which use pointers to represent the relationships between data items. The parallelism and memory access patterns are dictated by point-to relationships which can be irregular and sometimes time-varying. This reliance on pointer chasing imposes stringent requirements on memory latency (in addition to bandwidth) over a large main-memory footprint. As such these applications are poorly matched for traditional add-on FPGA accelerator cards attached to the I/O bus, which can only operate on a

limited window of locally buffered data at a time. In pointer chasing, the computation is required to de-reference a pointer to retrieve each node from memory, which contains both a data payload to be processed and a pointer to subsequent nodes. The exact computation on the payload and the determination of the next pointer to follow depend on the specific data structure and algorithm in use. In this paper, the authors ignore these differences and focus on only the basic effects of memory access latency and bandwidth on pointer chasing.

Ojika et al [132] propose SWiF for integrating FPGA-based accelerators into heterogeneous datacenters. To implement an offload infrastructure, SWiF is prototyped on the Intel Xeon+FPGA hardware platform using the software development kit (SDK): Accelerator Abstraction Layer (AAL). By using SWiF's API, CPU-intensive workloads are transparently offloaded to the FPGA. The authors demonstrate as a case study the feasibility of accelerating data compression in Apache Spark. Using the AAL SDK and Quartus Prime software, the authors implement the DEFLATE compression algorithm (at the highest compression level) in hardware and program the AFU with the resulting FPGA bitstream. They create a Java-based library that uses Java Native Interface (JNI) to indirectly invoke the FPGA accelerator through AAL and subsequently schedules a CPU thread for offloading to the FPGA. Scheduling a single thread can potentially leave the FPGA underutilized, another challenge therefore is, efficient sharing of the FPGA accelerator among multiple threads. By overlapping computation and memory accesses among threads, and minimizing the number of buffer instantiations, the proposed solution hides many of the data communication overheads between the JVM's heap memory and the FPGA's kernel space.

Acceleration using Xilinx Zynq SoC [181]

Umuroglu et al. [166] studied a breadth-first graph traversal using a Xilinx Zynq SoC FPGA that share memory between the ARM cores and programmable logic. Their work assigned different phases of the breadth-first traversal processing on the ARM cores and programmable fabric. Hurkat et al. [80] studied FPGA acceleration of a machine learning algorithm that took advantage of Convey's special high-throughput interface to a large pool of memory. It is an irregular parallel application, which requires irregular access over a large memory footprint and creates the opportunity for a tightly coupled processor-FPGA computer system.

Integration of Accelerators into Big Data Frameworks

Apache Spark [190] acceleration

Segal et al. [149] present SparkCL, which uses Aparapi [18] to automatically generate OpenCL kernel for Altera FPGAs [81] from Java bytecode and supports FPGAs acceleration on Spark by providing new kernel function types and modified Spark transformations and actions. Their prior work on accelerating Hadoop framework [150] identifies that due to architectural limitations, data transfer overhead and inefficient integration of accelerators into compute fabric, only highly compute intensive tasks should be offloaded. A combination of high computation complexity and a large amount of data is required for efficient acceleration and big data frameworks lack in architecture-aware memory allocation and device friendly data types.

Ghasemi et al. [67] create a custom RTL MapReduce framework that is capable of combining map-reduce HLS (high-level synthesis) kernels with a template interface. They provide a mechanism to efficiently share applications data between Spark and the custom accelerator inside the FPGA. Each worker incorporates a processor connected to an FPGA device through a physical medium, providing a two-way data-transfer and a communication link between the CPU and the hardware accelerator. In this approach, Spark master controls the workers at the highest level of granularity, with each worker monitoring the status of the hardware accelerators inside the FPGA. Computation on each unit is assumed to be independent of the rest of the workers. The executor process responds to read and write data blocks from a distributed file system. The executor is able to access the device driver through the software interface layer to transfer the input data to the FPGA. The driver also controls the Direct Memory Access (DMA) engine, instantiated inside the FPGA, for fast data transfer between the host memory and custom hardware. They implement a distributed memory architecture where the CPU and the FPGA are using two separate physical memories. Therefore transferring data between CPU and FPGA invokes memory copies between the physical memories. The custom accelerator uses the DMA to access data from the FPGA's memory. They use JNI to build data transfer link between JVM and FPGA. The memory buffer represents a contiguous block of physical memory that is accessible by the DMA engine inside the FPGA. The driver builds the mmap() system call through which physical memory can be mapped to the virtual address of the calling user process. It outperforms sequential I/O access when transferring high volumes of data.

Huang et.al [78] present Blaze framework, which provides programming API and run-time support for easy and efficient deployment of FPGA

accelerators in data-center. It abstracts FPGA accelerators as a service (FAAS) to efficiently share FPGA accelerators among multiple heterogeneous threads on a single node and extend YARN with accelerator center scheduling to efficiently share them among multiple computing tasks. Even though the work improves the utilization of the accelerator by sharing it between jobs, it does not show how many applications would be required to co-run to fully utilized the FPGA resources. It offloads the compute-intensive kernels e.g. Hamming distance calculation to the FPGA. A map phase can have different compute intensive kernels. When map tasks are scheduled to the executor pool threads, they all process the same compute-intensive kernels but on different data set. In Blaze, the input data from different executor-pool threads performing one compute-intensive kernel is batched, and a task is created that offloads the data to the accelerator corresponding to the compute kernel. So such tasks are created corresponding to their specific kernel. Those tasks are added to the task queue and it adopts a synchronous communication scheme that overlaps JVM to FPGA data communication with FPGA accelerator communication. Moreover, they persist the RDDs on the device memory. Thus, the capacity of FPGA device external memory limits the amount of data to be processed by the accelerators. Furthermore, the data processed by one of the accelerators have to be feedback to the host memory before the next task can start processing.

Nakamura et al. [128] propose to offload various one-at-a-time methodology operations onto FPGA based 10 GbE network interface card (NIC) and combine it with Spark streaming to complement its negative aspect, i.e. micro-batch processing methodology incurs high latency for detecting anomaly conditions and change points. This is because incoming data is accumulated into a micro-batch and then data analysis is performed for that micro-batch. Their rationale is that most stream processing frameworks are executed as a software program on microprocessors. When high bandwidth stream data is processed by an application program, all the received data are transferred from NIC to an application layer via TCP/IP network protocol stack. That is in a conventional software-based stream data processing, all the data, which may not be necessary for the applications are transferred to the application layer and then data processing tasks, such as filtering and detection, are performed. If data processing can be done at the NIC, the amount of data copied from NIC to the application layer can be reduced drastically and thus in-NIC processing can improve the performance of stream processing.

Kohei et al. [76] explore the management of data partition size to avoid excessive CPU-FPGA communication that can quickly diminish benefits of FPGA acceleration of Apache Spark. Using SVM training as a case study, they show that managing the partition size and restricting

data access to facilitate onboard data reuse play a crucial role in the resulting overall application performance. To accelerate only mappers, i.e. computations that are scheduled in parallel over the cluster in the map-reduce frameworks. In order to store one input image in the FPGA, they allow the input training vectors to stay as 8-bit integers for each element since the input images are 256-level grayscale images. Their system consists of CPU cluster where each node has a PCIe-attached FPGA. When the design is kick-started, it sends a command to Xilinx data mover core, and the initial weight vector is streamed in from on-board DDR3-RAM through the 512-bit stream interface, followed by all the training vectors in the RAM. Since the amount of data in the RDD partition might not fit into the DDR3 RAM, it is possible that the core is started multiple times within a single iteration of the algorithm. After processing all the training vectors in the DDR3, the core outputs the loss and the cumulative (sub) gradient back to the DDR3. They provide FPGA driver function to the mapPartitions call of the Spark. The driver function iterates over the RDD partition, stores each input entries into the Java "Byte Buffer", starts a RIFFA Direct Memory Access (DMA) (which avoids copying between kernel and userspace memory) to transfer it from host memory to the FPGA onboard DDR3 memory, and at last transfer and return the completed results when FPGA finishes.

Morcel et al. [126] present a system that consists of FPGA-augmented computing nodes. Each computing node is equipped with system-on-chip, which contains ARM processing system coupled with an FPGA, and uses external DDR3-RAM for external storage, non-volatile memory storage device for storing files, and the Ethernet controller to provide network connectivity. The cluster is managed by HDFS and Spark. They implement custom-designed FPGA-based accelerator for the 2D multi-layer convolution. The middleware layer is responsible for distributing the convolution horizontally while vertically offloading the convolution on each node to the FPGA. They augment Spark middle-ware with custom transformations to offload the execution of convolution tasks sent from the master to FPGA accelerators.

Apache Hadoop [156] Acceleration

Neshatpour [129] shows that offloading the compute-intensive kernels in machine learning algorithms results in more than 100x kernel speedup, which only translates into less than 3x performance improvement in an end-end Hadoop MapReduce environment. This is due to high communication cost of moving hotspot functions to the FPGA. Their system architecture consists of high-performance CPU as the master node, which is connected to several Zynq devices as slave nodes. The master node runs the HDFS and is responsible for job scheduling between all

the slave nodes. Each worker/slave node has a fixed number of map and reduces slots which are statically configured. The Zed board [181] used has ARM Cortex-A9 processor-based system (PS) and the FPGA being the programmable logic (PL). The connections between the PL and PS are established through AXI interconnect. The master and slave nodes communicate with each other through all to all switching network. which is implemented with the PCIe. In this architecture, the overheads include data transfer time between the nodes in the network through the PCI-express, the overhead of the switching network, and the data transfer time between the ARM core (PS) and the FPGA (PL). PCIe is used for the communication between the nodes in the system. Based on the number of nodes, the data is transferred among the various nodes in the system. They also assume that the entire input data is passed from the master and slave nodes. Their experiments show that overhead included in speed-up is considerably lower than the zero-overhead speed-up, if the data being transferred is large in size or the acceleration function is called multiple times.

MapReduce Acceleration

Shan et. al [152] presents FPMR, which implements an entire MapReduce framework on FPGAs so that data communication overhead can be eliminated. However, FPMR still requires the users to write customized map/reduce functions in RTL. It first generates $key, value$ pairs on the host and writes the configuration parameters to register on FPGA. It then initializes DMA data transferring, copy the $key, value$ pairs from the CPU to FPGA board. The processor scheduler then assigns the tasks to each mapper. Mappers process the assigned $key, value$ and store the generated intermediate $key, value$ in the local memory under the control of data controller. When a mapper finishes its job and there are jobs left, the processor scheduler will assign another job to it. When all the tasks are finished, the results are returned to the host main memory by the data controller.

Axel [165] and [187] are C-based MapReduce frameworks for not only FPGAs but also for GPUs. Both frameworks have straightforward scheduling mechanisms to allocate tasks to either FPGAs or GPUs. Melia [174] presents a C++ based MapReduce framework on OpenCL-based Altera FPGAs. It generates Altera FPGA accelerators from user-written C/C++ map/reduce functions and executes them on FPGA, but they do not exploit concurrent processing and CPUs are running roughly idle during the OpenCL kernel execution on FPGA.

In [95], an HW-SW co-design is presented where the map tasks are executed in the processors and a specialized hardware accelerator is implemented for the efficient processing of reduce tasks. The reduce function

in most of the applications is the same (e.g. accumulation or calculation of the average value). Therefore an efficient hardware accelerator is implemented that performs fast indexing of the $key, value$ pairs using cuckoo hashing scheme. In the second architecture [94], an integrated framework is proposed where the whole application is mapped to the FPGA. The map computational kernels, that are application specific, are created using high-level synthesis tools and the reduce tasks that are common to most of the applications, are executed using the common reduce hardware accelerator.

Li et al. [112] present map-reduce architecture to implement the k-means algorithm on an FPGA. The optimization they considered includes algorithmic segmentation (assignment + accumulation in the map and cluster generation in the reduce phase), data path elaboration, and automatic control (host program is also implemented on FPGA). Moreover, high level synthesis technique is utilized to reduce the development cycle and complexity. Each iteration in k-means is a map-reduce job. This job is segmented into the map phase and reduce phase. Map phase is responsible for the sample clustering and accumulation, which are executed by m mappers. The reduce phase mainly takes charge of generating new cluster centroids by division and is executed by the single reducer. Each mapper clusters the input samples by distance calculation and comparison then accumulates the samples in each cluster. Each mapper generates two outputs: one consists of labels indicating the cluster that each sample is assigned to, while the other comprises of intermediate results, including the number and the partial sums of samples in each cluster. The intermediate results from mappers are grouped and then sent into the reducer. The reducer accumulates the numbers and the partial sums of samples for each cluster, then generates new cluster centroids by calculating the mean of the samples in each cluster. The new cluster centroids will override the old ones and be used in the next iteration.

Approaches to integrate native code in java virtual machine based frameworks

Anderson et al. [20] show that offloading computation to a message processing interface(MPI) [6] environment from within Spark provides more than 10x speed-up including the overheads. Their approach is to serialize data from Spark RDDs and transfer the data from Spark to inter-process shared memory for MPI processing, Using information from the Spark driver, they execute plain MPI binaries on Spark workers with input and output paths in shared memory. The results of the MPI process are copied back to HDFS [32] and then into Spark for further processing. Their work contrast with another approach of integrating native code with Spark, which is to accelerate user-defined

functions (UDFs) in Spark by calling native C++ code through the Java native interface (JNI), while retaining the use of Spark for distributed communication and scheduling, e.g many operations in Spark machine-learning and graph analysis libraries offload computation to the Breeze library [33], which contains optimized numerical routines implemented either in Java or JNI-wrapped C. They implemented Spark+MPI using the Linux shared memory file system /dev/shm for exchanging data between Spark and MPI. They compare me) Spark based libraries, ii) offloading key computations into optimized C++ routines and calling these routines using JNI while retaining the use of Spark for distributed communication and scheduling and iii) offloading entire computations from Spark into the MPI environment using Spark + MPI approach. They show that Spark + MPI has run-time overheads, which depends on the size of input and output of the operation. In their experiments, it takes 3-5 seconds to transfer inputs to MPI through shared memory, and between 9-19 seconds to transfer outputs back into Spark using HDFS. Thus their work is not applicable for algorithms for fewer iterations or less work per iteration.

Dünner et al. [58] offload the computationally intense local solvers of Spark based learning algorithms into compiled C++ modules and show an order of magnitude performance improvement by reducing the computational cost and communication related overheads and argue that carefully tuning a distributed algorithm to trade-off communication and computation can improve the performance by orders of magnitude. They replace the local solver of CoCoA algorithm [167] with a JNI call to a compiled and optimized C++ module. The RDD data structure is modified so that each partition consists of a flattened representation of the local data. This modification allows one to execute the local solver using a map operation in Spark instead of a mapPartitions operation. In that manner, one can pass the local data into the native function call as pointers to contiguous memory regions rather having to pass an iterator over a more complex data structure. The C++ code is able to directly operate on the RDD (with no copy) by making use of the GetPrimitiveArrayCritical functions provided by the JNI.

2.5 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. [115] 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 provides 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. [100] present FPGA-based prototype in order to evaluate the impact of SIMT (single instruction multiple threads) 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 a 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. [137] 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, the 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 up to 15x reduction in execution time and 18x reduction in system energy. Islam et al. [83] 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 [148].

PIM for Graph Analytics

Ahn et al. [15] 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. [127] 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 lookups

PIM for Machine Learning Workloads

Lee et al. [107] 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 et al. [31] use a variant of the 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. [52] 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. [125] study 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 within one stack as much as possible, because a memory access may require traversing multiple SerDes links to reach the appropriate HMC target and further SerDEs link traversal is more expensive than 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. Furthermore, this 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 modeled as a simple microcontroller 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. [177] 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. [17] 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 DRAM-aware reshape accelerator integrated within 3D-stacked DRAM and a mathematical framework that is used to represent and optimize the reorganization operations.

Gokhale et al. [69] 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 stridden 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.

2.6 Processing in Nonvolatile Memory

Ranganathan et al. [143] propose nano-stores that co-locate 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. [36] examine the potential and limitations 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. [173] 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.

Choi et al. [44] 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 up to 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 accounts 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 [45].

Jun et al. [92] presents flash-based platform, called BlueDBM, built of flash storage devices augmented with application specific FPGA based in-storage processor. The data-sets are stored in the flash array and are read by the FPGA accelerators. Each accelerator implements an array of application-specific distance comparators, used in the high-dimensional nearest-neighbor search algorithms. The authors present results of comparative evaluation of the flash-based platform with FPGA-based accelerators against a disk-based system and a DRAM-based multi-core system. The evaluation shows that the flash-based system with in-storage embedded FPGA accelerators 10 times faster than the disk-based system and sometimes outperforms the DRAM-based system. Having comparative performance, the flash-based platform consumes half of power per node compared to a DRAM-based system [92]. Another study by the same authors [93] focus on spare pattern processing.In the presented system architecture, A FPGA-based application-specific accelerator of the BlueDBM node resides between the host server and the flash memory, and hence enables in-storage processing. The accelerator implements the sparse pattern matching kernel, that operates on the target dataset stored in the flash storage. Evaluation of the prototype accelerator against a software solution on a multicore system indicates its higher or matching performance and the gain in power and cost [93].

Hyeokjun et al. [43] evaluate the potential of NDP for ML using a full-fledged simulator of multi-channel SSD that can execute various ML algorithms on data stored on the SSD. They implement three stochastic

gradient descent (SGD) variants (synchronous, downpour, and elastic averaging) in the simulator, exploiting the multiple NAND channels to parallelize SGD. In addition, the authors compare the performance of ISP and that of conventional in-host processing, revealing the advantages of ISP.

Jo et al. [91] develop the iSSD simulator based on the gem5 simulator. iSDD equips a general processing core with a flash memory controller in each channel and also more powerful SSD cores in SSD. It also has SRAM and DRAM to read/write the data from/to cells. Data are transferred between DRAM in iSSD and the main memory in the host through a host interface, which is controlled by the host interface controller. They implement apriori, k-means, PageRank, and decision tree on top of the iSSD simulator. The results reveal that data mining with iSSD outperforms that with host CPUs up to 3x.

Quero et al [141] present an active SSD architecture called SelfSorting SSD that targets to offload sorting operations which are characterized by heavy data transfer. Self Sorting SSD exploits the internal hardware infrastructure of SSDs. SSDs employ the flash translation layer abstraction, which separates the logical block device seen by the host from the physical flash media by translating logical block address (LBA) requests into physical block page requests that are serviced internally by the SSD controller. This address translation information is stored inside the DRAM in the form of translation tables. The logical-physical separation not only helps to hide the complexities of flash memory but also enables a wide range of options to optimize SSD performance without having to modify anything in the host. The authors modify the FTL abstraction and implement indexing algorithm based on the B++tree data structure to support sorting directly in the SSD. Experiments on a real SSD platform reveal that the proposed architecture outperforms traditional external merge sort by up to 60.75%, reduces energy consumption by up to 58.86%, and eliminates all the data transfer overhead to compute sorted results. By performance merge operations on-the-fly in active SSDs results in 39% performance improvement compared to traditional external sorting [108]

Zsolt et al. [84] explore offloading part of the computation in database engines directly to the storage. They implement a cuckoo hash table with a slab-based memory allocator to improve the handling of collisions and various values sizes in hardware-based key-value stores. Lookups and scans are performed on the same data to minimize data transferred over the network. They implement runtime parametrizable selection operator both for structured and unstructured data in an effort to reduce data movement further.

Vermi et al. [168] focus on sorting big data using mergesort algorithm

on a heterogeneous system composed of a CPU and near-data processors (NDPs) located on the system memory channels. NDPs are implemented as workload-optimized processors on FPGA. For configurations with an equal number of active CPU cores and near-data processors, they experiments show performance speedup of up to 2.5.

Koo et al. [102] design APIs that can be used by the host application to offload a data-intensive task to the SSD processor. These APIs can be implemented by simple modifications to the existing Non-Volatile Memory Express (NVMe) command interface between the host and the SSD processor. They quantify the computation versus communication tradeoffs for near storage computing with TPC-H queries. Using a fully functional SSD evaluation platform they authors perform design space exploration of the proposed approach by varying the bandwidth and computation capabilities of the SSD processor. They evaluate static and dynamic approaches for dividing the work between the host and SSD processor and show that their design improves the performance by up to 20% when compared to processing at the host processor only, and 6% when compared to processing at the SSD processor only.

Do et al. [135] focus on exploring the opportunities and challenges associated with exploiting the compute capabilities of Smart SSDs for relational analytic query processing. They extend Microsoft SQL Server to offload database operations onto a Samsung Smart SSD. The selection and aggregation operators are compiled into the firmware of the SSD. The results show 2.7x improvement in end-to-end performance compared to using the same SSDs without compute functionality and 3.0x reduction in energy consumption.

Kang et al. [98] propose a model that demonstrates the use of an SSD as data processing node that can achieve higher performance and lower energy consumption by enabling efficient data flow and consuming small amounts of host system resources. They evaluate in-storage processing (ISP) on a real multi-level cell SSD device and perform an end-to-end evaluation of performance and energy consumption covering the entire system. The proposed smart SSD harnesses the processing power of the device using an object-based communication protocol. Smart SSDs rely upon tasks: independent I/O tasks of an application running on the device. To allow applications to better use SSDs, they develop a programming interface to execute tasks based on MapReduce. They implement the Smart SSD features in the firmware of a Samsung SSD and modified the Hadoop core and MapReduce framework to use tasklets as a map or a reduce function. To evaluate the prototype, they use a micro-benchmark and log analysis application on both a device and a host. They find that under the current SSD architecture, excessive memory accesses will make the task execution slower than in the host due to

the high memory latency and low processing power. The experiments show that total energy consumption is reduced by 50% due to the low-power processing inside a Smart SSD. Moreover, a system with a Smart SSD can outperform host-side processing by a factor of two or three by efficiently utilizing internal parallelism when applications have light traffic to the device DRAM under the current architecture

Cho et al. [42] propose SSD architecture that integrates a graphics processing unit (GPU). They provide API sets based on the MapReduce framework that allows users to express parallelism in their application, and that exploit the parallelism provided by the embedded GPU. They develop a performance model to tune the SSD design. The experimental results show that the proposed XSD is approximately 25 times faster compared to an SSD model incorporating a high-performance embedded CPU and up to 4 times faster than a model incorporating a discrete GPU.

Seshadri et al. [151] propose programmability a central feature of the SSD interface. The resulting prototype system called Willow, contains storage processor units, each of which includes a microprocessor, an interface to the inter-SPU interconnect, and access to the array of non-volatile memory. Each SPU runs a small operating system that manages and enforces security. On the host-side, the Willow driver creates and manages a set of objects that allow the OS and applications to communicate with SPUs. The programmable functionality is provided in the form of SSD Apps. Each SSD App consists of handlers that the Willow kernel driver installs at each SPU on behalf of the application, a library that an application uses to access the SSD App, and a kernel module. Willow allows programmers to augment and extend the semantics of an SSD with application-specific features without compromising file system protections. The SSD Apps running on Willow give applications low latency, high-bandwidth access to the SSD's contents while reducing the load that IO processing places on the host processor. The programming model for SSD Apps provides great flexibility, supports the concurrent execution of multiple SSD Apps in Willow, and supports the execution of trusted code in Willow. The authors implement six SSD Apps to demonstrate the effectiveness and flexibility of Willow. Their findings show that defining SSD semantics in software is easy and beneficial and that Willow makes it feasible for a wide range of IO-intensive applications to benefit from a customized SSD interface.

Gu et al. [71] present Biscuit, a near-data processing framework designed for SSDs. It allows programmers to write a data-intensive application to run on the host system and the storage system in a distributed manner. Biscuit offers a high-level programming model, built on the concept of data flow. Data processing tasks communicate through typed and data

ordered ports. Biscuit does not distinguish tasks that run on the host system and the storage system. As a result, biscuit has desirable traits like generality and expressiveness, while promoting code reuse and naturally exposing concurrency. They implement Biscuit on a host-system that runs Linux OS and a high-performance solid-state drive. When data filtering is done is by hardware in the solid-state drive, the average speed-up obtained for top five queries of TPC-H is over 15x.

Biscuit is inspired by flow-based programming model. The application is constructed of tasks and data pipes connecting tasks. Tasks may run on a host computer or an SSD. Biscuit allows the user to dynamically load user tasks to run on the SSD. Resources needed to run user tasks are allocated at runtime. Biscuit supports full C++11 features and standard libraries. Biscuit implements light-weight multi-threading and comes natively with multi-core support. The authors report through measurement performance of key operations of NDP on a real, high-performance SSD. For example, it sustains sequential read bandwidth in excess of 3GB/s using PCIe Gen.3 x4 links. The SSD internal bandwidth is shown to be higher than this bandwidth by more than 30%. On top of Biscuit, the authors ported a version of MySQL. They modified its query planner to automatically identify and offload certain data scan operations to the SSD. Portions of its storage engine are rewritten so that an offload operation is passed to the SSD at runtime and data are exchanged with the SSD using Biscuit APIs. The SSD hardware also incorporates a pattern matcher IP designed for NDP. They write NDP codes that take advantage of this IP. When this hardware IP is applied, modified MySQL significantly improves TPC-H performance. The total execution time of all TPC-H queries is reduced by 3.6x.

2.7 Processing in Hybrid 3D-Stacked DRAM and NVRAM

Huang et al. [139] 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 [116], 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.

2.8 Interoperability of PIM with Cache and Virtual Memory

Challenges of PIM architecture design are the cost-effective integration of logic and memory, unconventional programming models and lack of interoperability with caches and virtual memory. Ahn et al. [16] 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 the locality-aware execution of PIM operations. Evaluations show good adaptivity across randomly generated workloads

2.9 Profiling Bigdata Platforms

Oliver et al. [133] 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. [134] 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 reveals 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 [64, 88–90, 99, 170, 186, 189]. However, these studies lack in identifying the limitations of modern scale-up servers for Spark-based data analytics. Ferdman et al. [64] 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. [196] 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 [88]. They also suffer from notable from end stalls but L2 and L3 caches are effective for them. Wang et al. [170] conclude the same about L3 caches and L1 I-Cache miss rates despite using larger datasets. Deep dive analysis [186] reveal that big data analysis workload is bound on memory latency, but the conclusion cannot be generalized. None of the abovementioned works consider frameworks that enable in-memory computing of data analysis workloads.

Jiang et al. [90] observe those memory access characteristics of the Spark and Hadoop workloads differ. At the micro-architecture level, they have roughly same behavior and point current micro-architecture works for Spark workloads. Contrary to that, Jia et al. [89] conclude that Software

stacks have a significant impact on the micro-architecture behavior of big data workloads. Our findings of hyper-threaded cores for big data applications is also corroborated by [87] even though they have adopted a different methodology for the workload characterization.

Tang et al. [163] have shown that NUMA has a significant impact on Gmail backend and web.search frontend. Beamer et al. [30] have shown NUMA has moderate performance penalty and SMT has limited potential for graph analytics running on Ivy bridge server. Kanev et al. [97] have argued in favor of SMT after profiling live data center jobs on 20,000 google machines. Our work extends the literature by profiling Spark jobs. Researchers at IBM's Spark technology center [41] have also shown moderate performance gain from NUMA process affinity. Our work gives micro-architectural reasons for this moderate performance gain.

Ruirui et al. [119] have compared throughput, latency, data reception capability and performance penalty under a node failure of Apache Spark with Apache Storm. Miyuru et al. [49] have compared the performance of five streaming applications on System S and S4. Jagmon et al. [37] have analyzed the performance of S4 in terms of scalability, lost events, resource usage and fault tolerance. Our work analyzes the micro-architectural performance of Spark Streaming.

2.10 Project Tungsten

The inventors of Spark have a roadmap for optimizing the single node performance of Spark under the project name Tungsten [9]. 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.

2.11 Hardware Prefetching

With the emergence of big data analytics, which are placing ever-growing demands on the need for effective data prefetching, it is obligatory to address the inefficiencies of the current state of the art history based prefetchers that are lower in accuracy and higher meta-data storage requirements [62]. The aim is to understand the spatiotemporal memory access patterns of MapReduce based analytics applications and incorporate this information to improve the accuracy of data prefetchers while reducing the on-chip meta-data storage. Wang et al. [169] study how prefetching schemes affect cloud workloads. They conduct detailed analysis on address patterns to explore the correlation between prefetching performance and intrinsic cloud workload characteristics. They focus particularly on the behavior of memory accesses at the last-level cache and beyond. They find that cloud workloads, in general, do not have dominant strides. State-of-the-art prefetching schemes are only able to improve performance for some cloud applications such as Web search and cloud workloads with long temporal reuse patterns often get negatively impacted by prefetching, especially if their working set is larger than the cache size.

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-an 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 [113]. A 3000-node cluster simulation driven by a real-world trace from Facebook shows that on average a cluster comprising ARM-based micro-servers, 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 [114]. 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 ruin 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 [64, 73, 116]. 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 [116]. FAWN architecture [19] 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 the capital cost and 35% of power costs at 16 nm [111].

Chapter 3

Identifying the Performance bottlenecks for In-Memory Data Analytics

3.1 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 [156], graph processing [121] and stream processing [164]. 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 [21]. 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 [64, 88–90, 99, 170, 186]. 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.

3.2 Background

Spark

Spark is a cluster computing framework that uses Resilient Distributed Datasets (RDDs) [190], 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 3.1 describe the parameters necessary to configure Spark properly in local mode on a scale-up server.

Table 3.1: 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

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 [185].

3.3 Methodology

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 4.1 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 [170], DCBench [88], HiBench [79] and Cloudsuite [64]. Phoenix++ [162], Phoenix rebirth [188] and Java MapReduce [158] 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 [124]. 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.

Table 3.2: 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

System Configuration

Table 7.3 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 [56]. With Hyper-Threading and Turbo-boost disabled, there are 24 cores in the system operating at the frequency of 2.7 GHz.

Table 7.4 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 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 7.4.

Table 3.3: 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	

Table 3.4: 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				

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 [4] 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 [66], we believe, it is enough for Big Data applications. We use a top-down analysis method proposed by Yasin [185] to identify the micro-architectural inefficiencies.

Metrics

The definition of metrics used in this chapter, are taken from Intel Vtune online help [4].

* **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.

3.4 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.

Application Level

Figure 3.1 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.

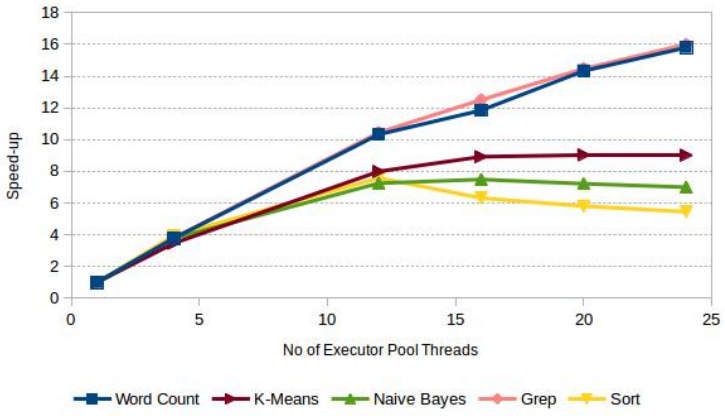


Figure 3.1: Scalability of Spark Workloads in Scale up Configuration

Stage Level

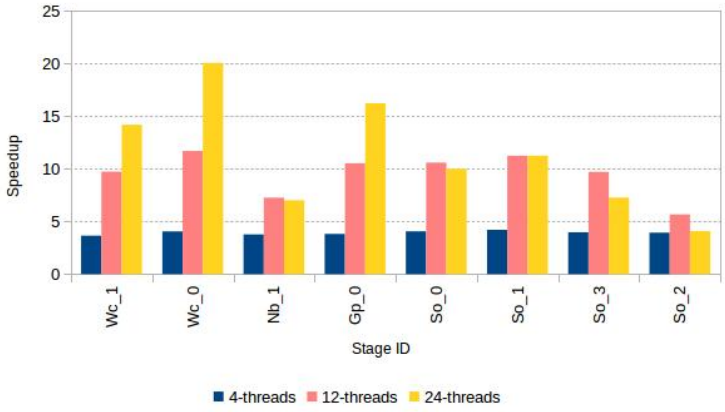
Next we drill down to stage level and observe how different stages scale with the number of executor pool threads. We only study those stages

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.

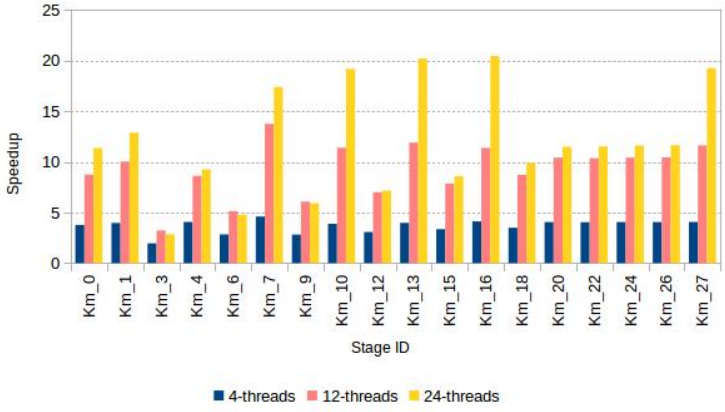
At 4-threads case (see Figure 3.2a), all stages of a workload exhibit ideal scalability but in 12 and 24-threads, the scalability characteristics vary among the stages, e.g. Wc_0 shows better speed-up than Wc_1 in 24-threads case. The scalability of Sort is worst among all applications in 24-threads case because of So_2 stage that does not scale beyond 4 threads. In K-Means (see Figure 3.2b), 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.

Tasks Level

Figure 3.3a and 3.3b 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 3.5 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%



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

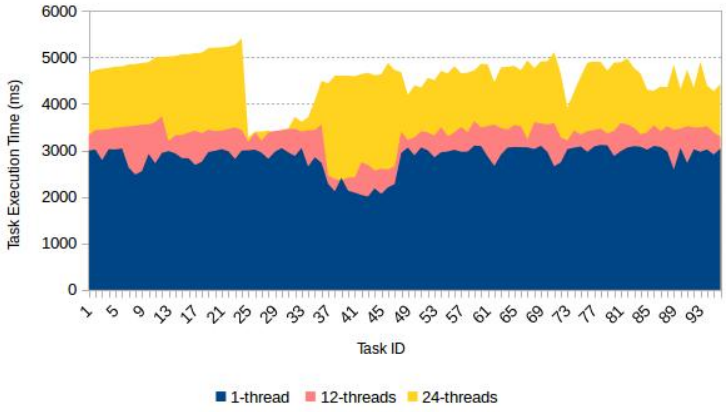


(b) K-Means

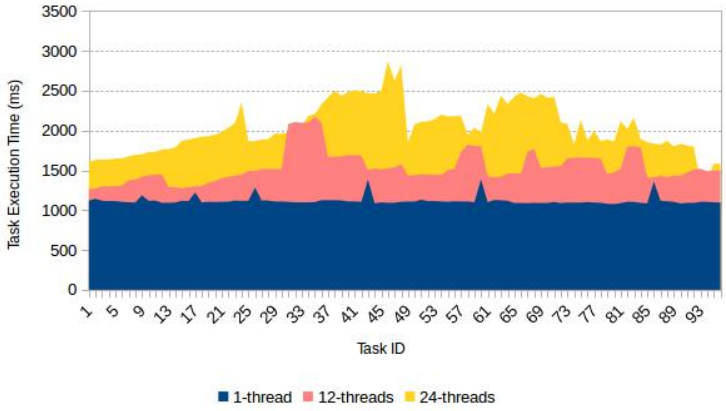
Figure 3.2: Performance at Stage Level

Table 3.5: 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



(a) Word Count (Wc_1)



(b) Kmeans (Km_0)

Figure 3.3: Performance at Task Level

3.5 Scalability Limiters

CPU Utilization

Figure 3.4 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 3.1, 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

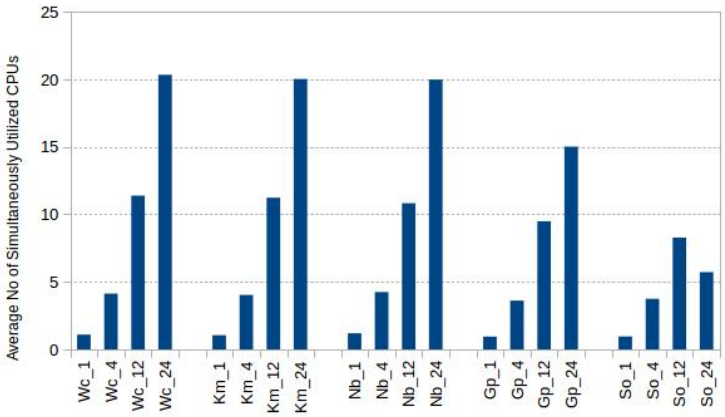
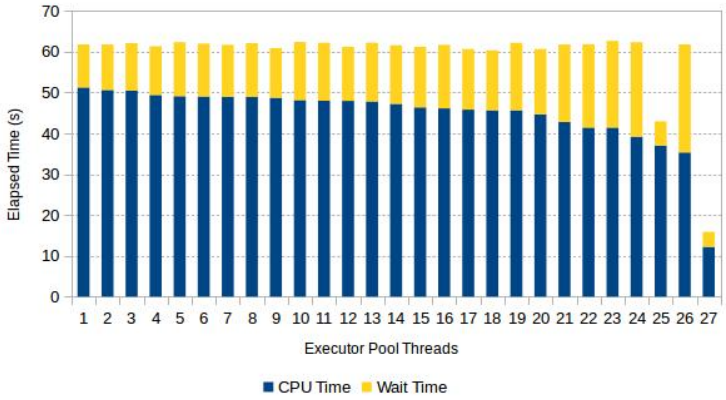


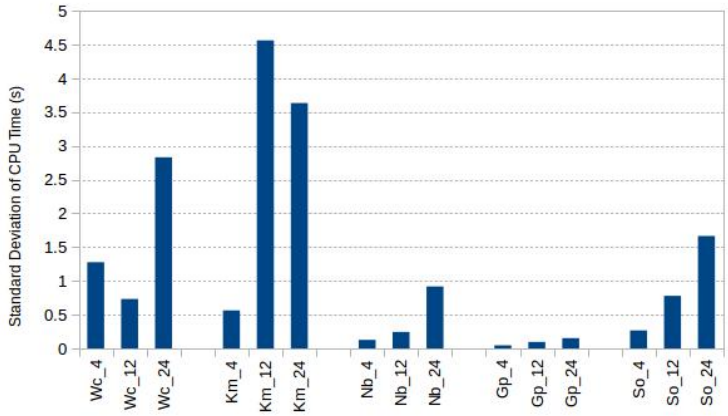
Figure 3.4: CPU Utilization of Benchmarks

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 3.5a 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 3.5b. 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 [61].



(a) K-Means



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

Figure 3.5: Load Imbalance in Spark Benchmarks

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 3.6a 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

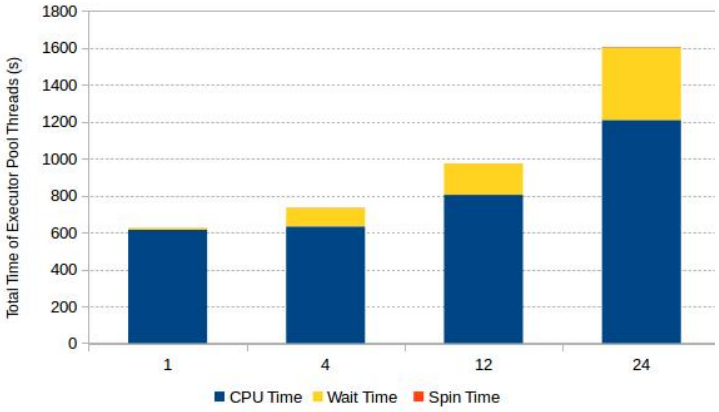
beyond the CPU time required to perform the same work in a sequential computation is termed as work time inflation as suggested by Oliver et-al [133].

Figure 3.6b 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.

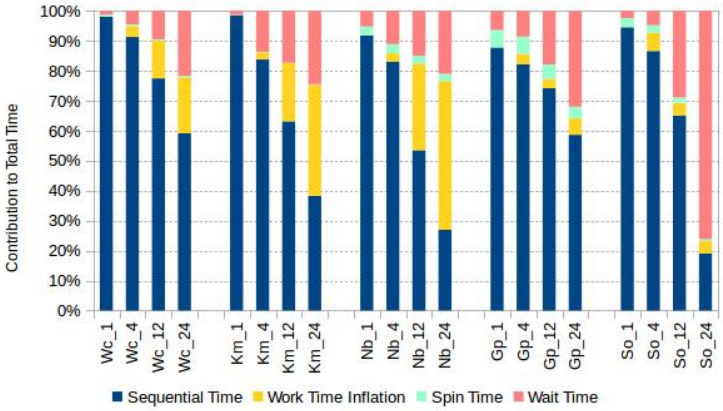
By comparing the data in Figure 6.7 with performance data in Figure 3.1, we see that Grep does not scale because of wait time overhead. Sort has the worst scalability because of significant contribution of wait time. In Word Count, there is equal contribution of work time inflation and wait time overhead 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.

Micro-architecture

Top Level Figure 4.4a 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 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 3.6: Work Time Inflation in Spark Benchmarks

Backend Level Figure 3.7c 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 4.4b. 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 4.4c 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.

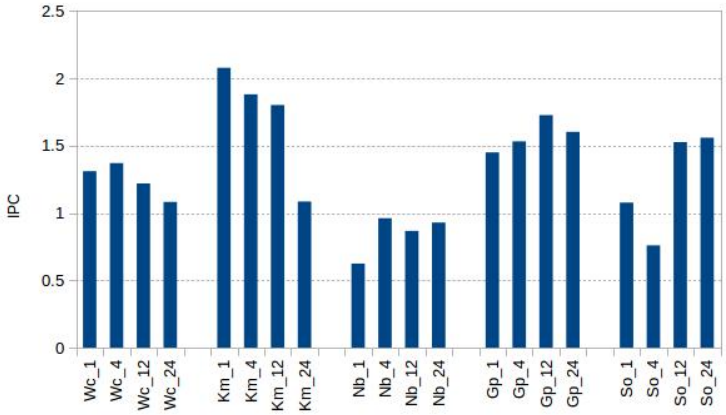
Frontend Level Figure 3.7f shows the fraction of pipeline slots during which CPU was stalled due to front-end latency and front-end bandwidth issues. At higher number of threads, front- end stalls are equally divided among latency and bandwidth issues. On average across the workloads; front-end latency 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%.

Memory Bandwidth Saturation

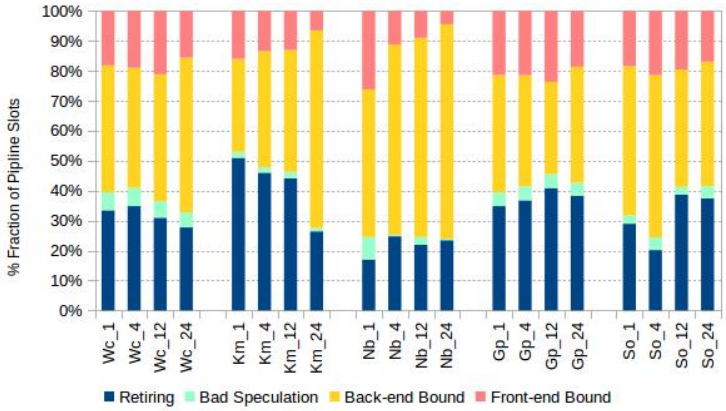
Figure 4.4d shows the amount of data read and written to each of the two DRAM packages via the processor's integrated memory controller. The bandwidth (Gigabytes/sec) to package_1 shows an increasing trend with increase in threads in the executor pool. The same trend can be seen for total memory bandwidth 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.

3.6 Related Work

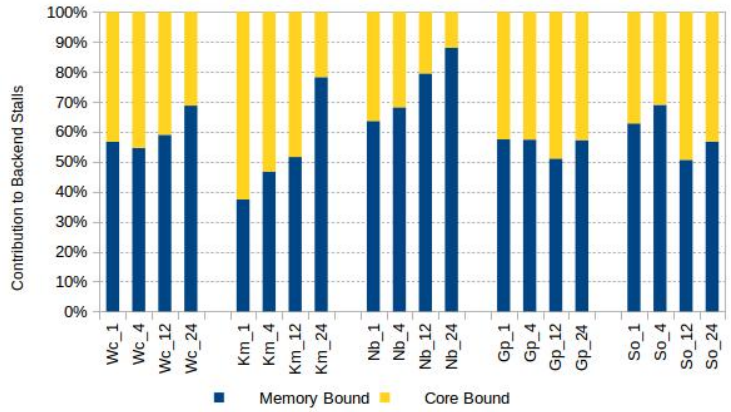
Oliver et al. [133] have shown that task parallel applications can exhibit poor performance due to work time inflation. We see similar phenom-



(a) IPC

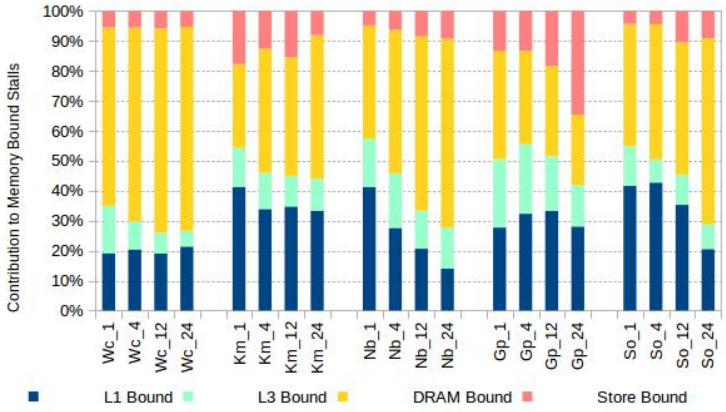


(b) Top Level

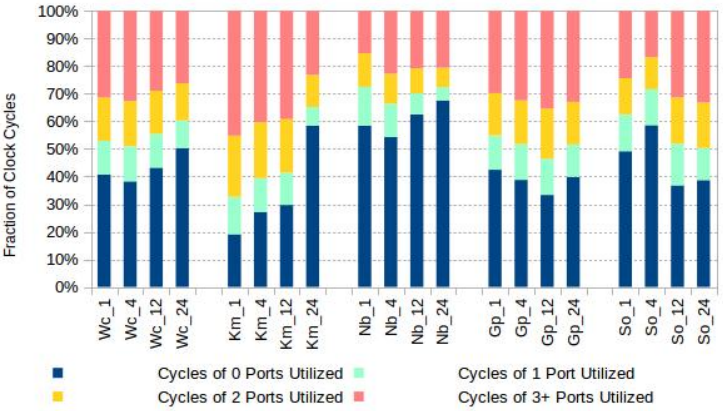


(c) Backend Level

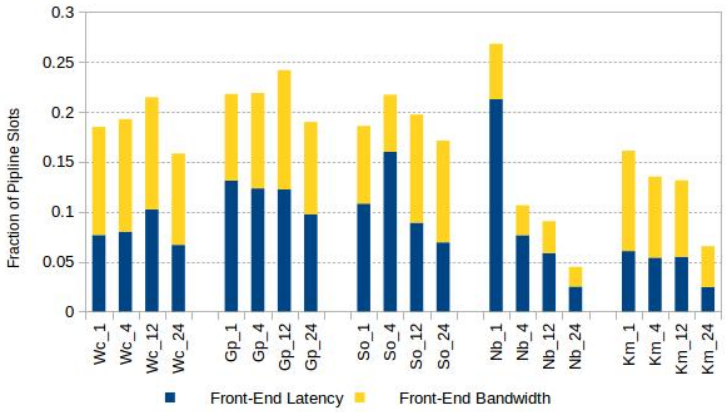
Figure 3.7: Top-Down Analysis Breakdown for Benchmarks with Different No of Executor Pool Threads



(d) Memory Level



(e) Core Level



(f) Frontend Level

Figure 3.7: Top-Down Analysis Breakdown for Benchmarks with Different No of Executor Pool Threads

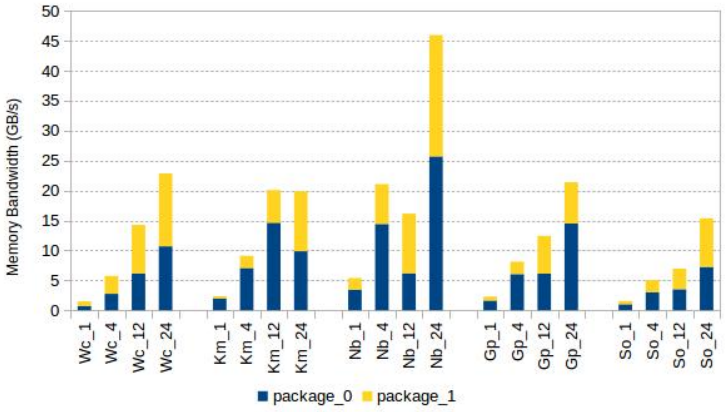


Figure 3.8: Memory Bandwidth Consumption of Benchmarks

ena in Spark based workloads. Ousterhout et al. [134] 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.

Ferdman et al. [64] 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. [196] 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 [88]. They also suffer from notable from end stalls but L2 and L3 caches are effective for them. Wang et al. [170] conclude the same about L3 caches and L1 I Cache miss rates despite using larger data sets. Deep dive analysis [186] 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. [90] 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. [89] 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.

3.7 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. [55] show potential for aggressively pre-fetching large sections of the dataset onto a faster tier of memory subsystem.

Chapter 4

Understanding the Impact of Data Volume on In-Memory Data Analytics

4.1 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 [21]. 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 [188], Ostrich [40] and Polymer [195] 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 [190] 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 [24] 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 chapter, 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.

4.2 Background

Spark is a cluster computing framework that uses Resilient Distributed Datasets (RDDs) [190] 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.

4.3 Methodology

Benchmarks

Table 4.1 shows the list of benchmarks along with transformations and actions involved. We used Spark versions of the following benchmarks from BigDataBench [170]. Big Data Generator Suite (BDGS), an open source tool was used to generate synthetic datasets based on raw data sets [124].

* **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 4.1: 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

System Configuration

Table 7.3 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 [56]. With Hyper-Threading and Turbo-boost disabled, there are 24 cores in the system operating at the frequency of 2.7 GHz.

Table 7.4 also lists the parameters of JVM and Spark. For our experiments, we use HotSpot JDK version 7u71 configured in server mode (64 bit). The 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 [5, 53]. 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

Table 4.2: 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	

after tuning are given in Table 7.4. Further details on the parameters are available [10].

Table 4.3: 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				

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 [4] to perform concurrency analysis and general micro-architecture exploration. For scalability study, each benchmark is run 5 times within a single JVM invocation 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 [185] 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.

4.4 Scalability Analysis

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 7.4. 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 4.1a 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.

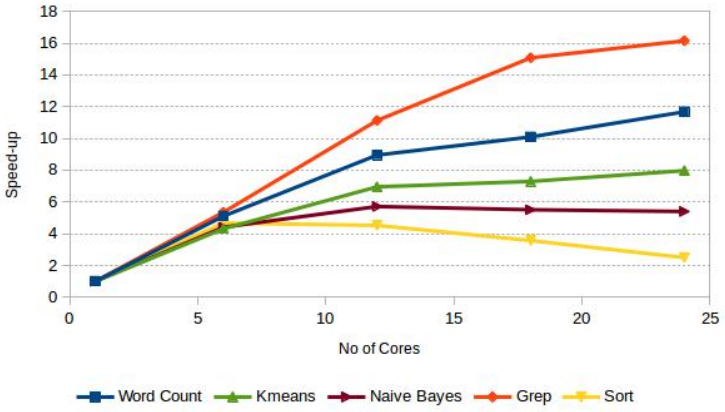
Does performance remain consistent as we enlarge the data size?

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 metric as suggested in by Luo et al. [120]. 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 4.1b, 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.

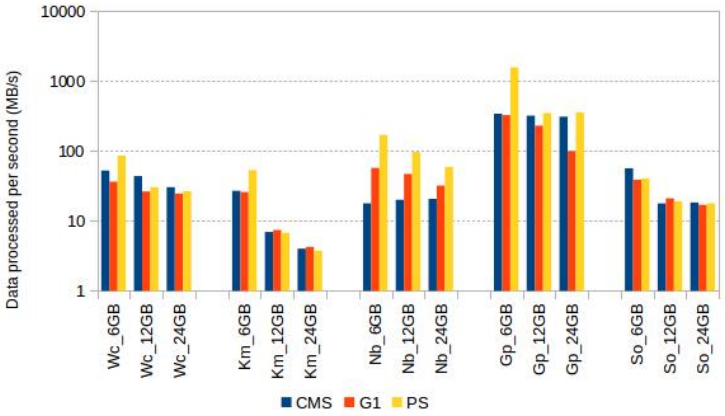
4.5 Limitations to Scale-up

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 4.2a 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



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



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

Figure 4.1: Scale-up performance of applications: (a) when the number of cores increases and (b) when input data size increases.

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 4.2b, 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

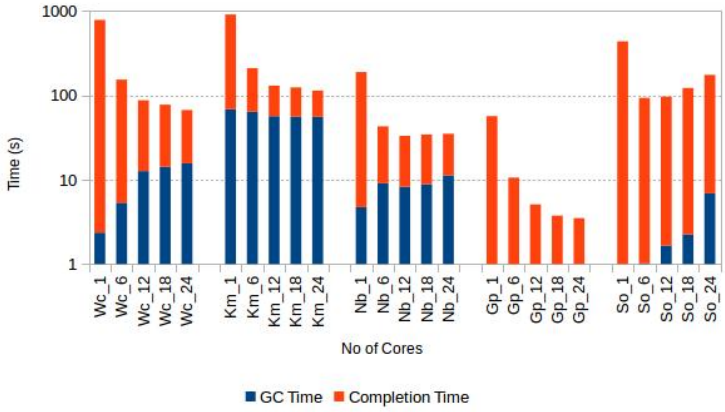
that if GC time had been increasing at most linearly, DPS would not have decreased significantly. For K-Means, DPS decreases by 14x when data size increases by 4x. For similar scenario in Naive Bayes, DPS decreases by 3x and GC time increases by 3x. Hence we can conclude that performance of Spark applications 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 4.2b 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.

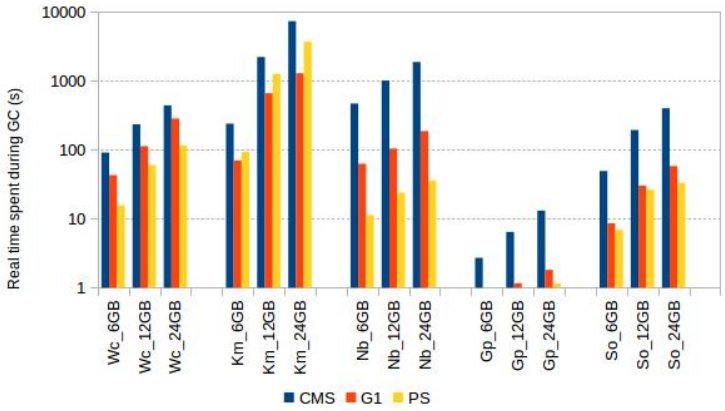
Does file I/O become a bottleneck under large data volumes?

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. Figure 6.7b 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% 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 6.7b also



(a) GC overhead is a scalability bottleneck.



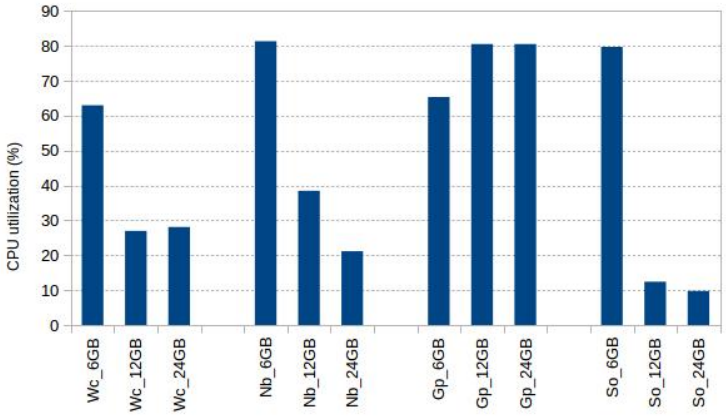
(b) GC time increases at a higher rate with data size.

Figure 4.2: Impact of garbage collection on application performance: (a) when the number of cores increases and (b) when input data size increases.

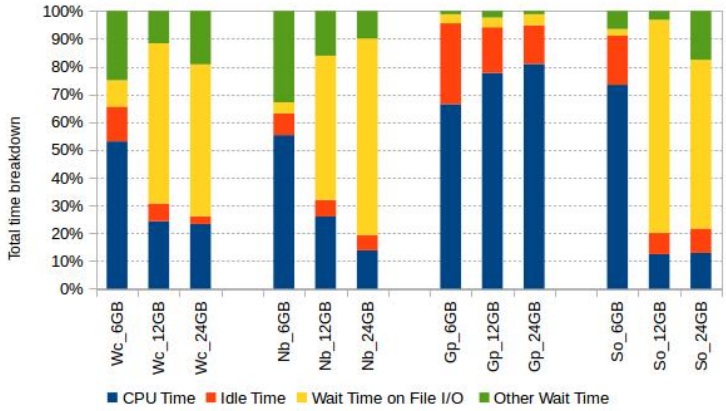
correlates with CPU utilization numbers in Figure 4.3a. On average across the workloads, CPU utilization decreases from 72.34% to 39.59% as the data size is increased from 6 GB to 12 GB which decreases further by 5% in 24 GB case.

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



(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 4.3: Time breakdown under executor pool threads.

configured to use 24 executor pool threads. Each workload is run with 6 GB, 12 GB and 24 GB of input data. Figure 4.4a 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 4.4b.

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 4.4d 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.

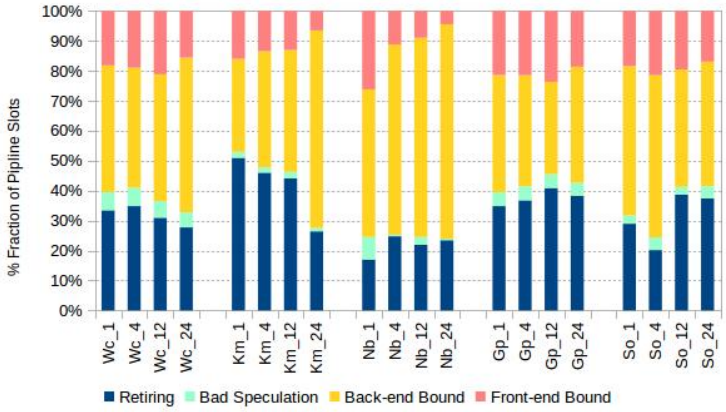
Figure 4.4c shows the fraction of cycles during execution ports are used. Ports provide the interface between instruction issue stage and the various functional units. By comparing 6 GB and 24 GB cases, we observe that cycles during which no port is used decrease from 51.9% to 45.8% on average across the benchmarks and cycles during which 1 or 2 ports are utilized increase from 22.2% to 28.7% on average across the workloads.

4.6 Related Work

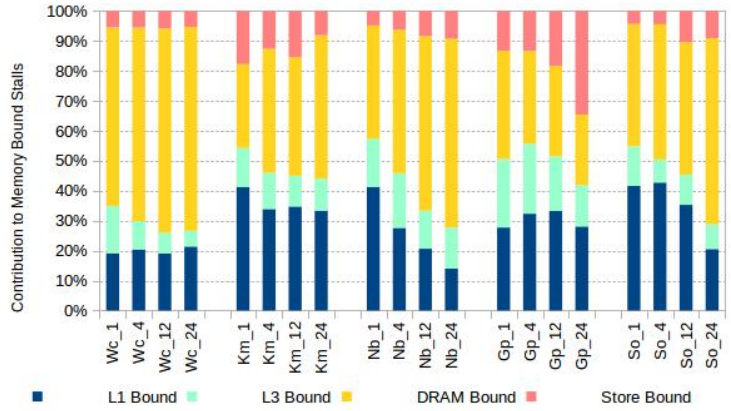
Several studies characterize the behaviour of big data workloads and identify the mismatch between the processor and the big data applications [64, 88–90, 99, 170, 186]. However these studies lack in identifying the limitations of modern scale-up servers for Spark based data analytics. Ousterhout et al. [134] 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. [170] have shown that the volume of input data has considerable affect on the micro-architecture behaviour of Hadoop based workloads. We make similar observation about Spark based data analysis workloads.

4.7 Conclusions

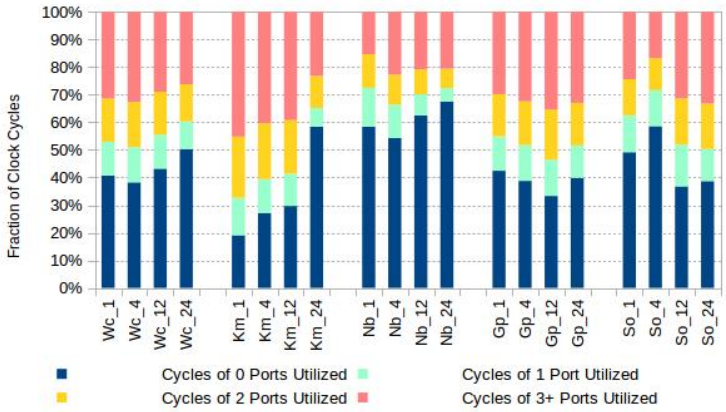
We have reported a deep dive analysis of Spark based data analytics on a large scale-up server. The key insights we have found are as follows:



(a) Retiring rate increases at larger datasets.

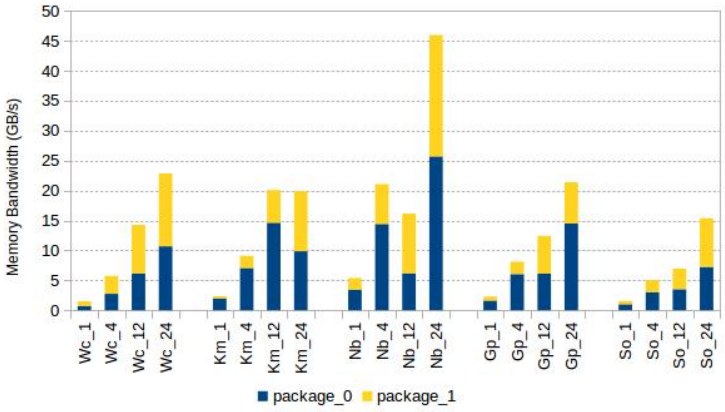


(b) L1 Bound stalls increase with data size.



(c) Port utilization increases at larger datasets.

Figure 4.4: Micro-architecture performance is inconsistent across different data sizes.



(d) Memory traffic decreases with data size.

Figure 4.4: Micro-architecture performance is inconsistent across different data sizes.

* 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.

Chapter 5

Understanding the Impact of Data Velocity on In-Memory Data Analytics

5.1 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 most analytics jobs do not process really huge data sets and those modern scale-up servers are adequate to run analytics jobs [21]. Additionally, commonly used predictive analytics such as machine learning algorithms, work on filtered datasets that easily fit into the memory of modern scale-up servers. Moreover, the today's scale-up servers can have CPU, memory, and persistent storage resources in abundance at affordable prices. Thus we envision the small cluster of scale-up servers will be the preferable choice of enterprises in near future.

While Phoenix [188], Ostrich [40] and Polymer [195] 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 [190] is getting popular in the industry because it enables in-memory processing, scales out to many 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 [24, 25] characterize the micro-architectural performance of in-memory data analytics with Spark on a scale-up server but they cover only batch processing workloads and they also do not quantify the impact of data velocity on the micro-architectural performance of Spark workloads. Knowing the limitations of modern scale-up servers for real-time streaming data analytics with Spark will help in achieving the future goal of improving the performance of real-time streaming data analytics with Spark on small clusters of scale-up servers.

Our contributions are:

* We characterize the micro-architectural performance of Spark-core, Spark MLlib, Spark SQL, GraphX and Spark Streaming.
* We quantify the impact of data velocity on the micro-architectural performance of Spark Streaming.

The rest of this chapter is organized as follows. Firstly, we provide background and formulate the 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 in section 6.

5.2 Background

Spark

Spark is a cluster computing framework that uses Resilient Distributed Datasets (RDDs) [190] 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 MLlib

Spark MLlib [122] 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.

Graph X

GraphX [70] 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.

Spark SQL

Spark SQL [22] 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.

Spark Streaming

Spark Streaming [191] 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 as 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.

Spark on Modern Scale-up Servers

Our recent efforts on identifying the bottlenecks in Spark [24, 25] 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 access 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 chapter.

In this chapter, we answer the following questions concerning real-time streaming data analytics running on modern scale-up servers using 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 the micro-architectural behaviour of stream processing data analytics?

5.3 Methodology

Our study of micro-architectural characterization of real-time streaming data analytics is based on an empirical study of performance of batch and stream processing with Spark using representative benchmark workloads.

Workloads

This study uses batch processing and stream processing workloads, described in Table 5.1 and Table 5.2 respectively. Benchmarking big data analytics is an open research area, we, however, choose the workloads carefully. Batch processing workloads are the subset of BigdataBench [170] and HiBench [79], which are highly referenced benchmark suites in the big data domain. Stream processing workloads used in the chapter are the superset of StreamBench [119] and also cover the solution patterns for real-time streaming analytics [136].

The source codes for Word Count, Grep, Sort, and NaiveBayes are taken from BigDataBench [170], whereas the source codes for K-Means, Gaussian, and Sparse NaiveBayes are taken from Spark MLlib 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 is used to generate synthetic data sets based on raw data sets [124].

System Configuration

Table 7.3 shows details about our test machine. 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 [13]. With Hyper-Threading and Turbo-boost disabled, there are 24 cores in the system operating at the frequency of 2.7 GHz.

Table 7.4 lists the parameters of JVM and Spark after tuning. For our experiments, we configure Spark in local mode in which driver and executor run inside a single JVM. We use HotSpot JDK version 7u71 configured in server mode (64 bit). 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 [25]. The heap size is chosen such that the memory consumed is within the system. The details on Spark internal parameters are available [10].

Table 5.1: 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
	Grep (Gp)	searches for the keyword "The" in a text file and filters out the lines with matching strings to the output file	
	Sort (So)	ranks records by their key	Numerical Records
	NaiveBayes (Nb)	runs sentiment classification	Amazon Movie Reviews
Spark MLlib	K-Means (Km)	uses K-Means clustering algorithm from Spark MLlib. The benchmark is run for 4 iterations with 8 desired clusters	Numerical Records
	Sparse NaiveBayes (Snb)	uses NaiveBayes classification algorithm from Spark MLlib	
	Support Vector Machines (Svm)	uses SVM classification algorithm from Spark MLlib	
	Logistic Regression (Logr)	uses Logistic Regression algorithm from Spark MLlib	
Graph X	Page Rank (Pr)	measures the importance of each vertex in a graph. The benchmark is run for 20 iterations	Live Journal Graph
	Connected Components (Cc)	labels each connected component of the graph with the ID of its lowest-numbered vertex	
	Triangles (Tr)	determines the number of triangles passing through each vertex	
Spark SQL	Aggregation (SqlAg)	implements aggregation query from BigdataBench using DataFrame API	Tables
	Join (SqlJo)	implements join query from BigdataBench using DataFrame API	

Measurement Tools and Techniques

We use Intel Vtune Amplifier [4] to perform general micro-architecture exploration and to collect hardware performance counters. 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 find 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 [185] to study the

Table 5.2: 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)	uses DataFrames and SQL to count words in text received from the network every 2 sec.	
Click stream Error Rate Per Zip Code (CErpz)	returns the rate of error pages (a non 200 status) in each zipcode over the last 30 sec. A page view generator generates streaming events over the network to simulate page views per second on a website.	Click streams
Click stream Page Counts (CPc)	counts views per URL seen in each batch.	
Click stream Active User Count (CAuc)	returns number of unique users in last 15 sec	
Click stream Popular User Seen (CPus)	look for users in the existing dataset and print it out if there is a match	
Click stream Sliding Page Counts (CSpc)	counts page views per URL in the last 10 sec	
Twitter Popular Tags (TPt)	calculates popular hashtags (topics) over sliding 10 and 60 sec windows from a Twitter stream.	Twitter Stream
Twitter Count Min Sketch (TCms)	uses the Count-Min Sketch, from Twitter's Algebird library, to compute windowed and global Top-K estimates of user IDs occurring in a Twitter stream	
Twitter Hyper Log Log (THll)	uses HyperLogLog algorithm, from Twitter's Algebird library, to compute a windowed and global estimate of the unique user IDs occurring in a Twitter stream.	

Table 5.3: Converted Spark Operations in Workloads

Workload	**Converted Spark Operation**
Wc	Map, ReduceByKey, SaveAsTextFile
Gp	Filter, SaveAsTextFile
So	Map, SortByKey, SaveAsTextFile
Nb	Map, Collect, SaveAsTextFile
Snb	Map, RandomSplit, Filter, CombineByKey
Km	Map, MapPartitions, MapPartitionsWithIndex, FlatMap,Zip, Sample, ReduceByKey,
Svm	Map, MapPartitions, MapPartionswithIndex, Zip, Sample,
Logr	RandomSplit,Filter,MakeRDD,Union, TreeAggregate, CombineByKey, SortByKey
Pr	Coalesce, MapPartitionswithIndex, MapPartitions, Map, PartitionBy, ZipPartitions
Cc	
Tr	
SqlAgg	Map, MapPartitions, TungstenProject, TungstenExchange, TungstenAggregate, ConvertToSafe
SqlJo	Map, MapPartitions, SortMergeJoin, TungstenProject, TungstenExchange, TungstenSort, ConverToSafe
SqWc	FlatMap, ForeachRDD, TungstenExchange, TungstenAggregate, ConvertToSafe
NWc	FlatMap, Map, ReduceByKey
NGp	Filter, Count
WWc	FlatMap, Map, ReduceByKeyAndWindow
StWc	FlatMap, Map, UpdateStateByKey
CErPz	FlatMap, Map, Window, GroupByKey
CAuc	FlatMap, Map, Window, GroupByKey, Count
CPus	FlatMap, Map, Parallelize, ForeachRDD
CPc	FlatMap, Map, CountByValue
CSPc	FlatMap, Map, CountByValueAndWindow
Tpt	FlatMap, Map, ReduceByKeyAndWindow, Transform
Tcms	Map, MapPartitions, Reduce, ForeachRDD, ReduceByKey,
Thll	Map, MapPartitions, Reduce

Table 5.4: Machine Details.

Component	**Details**	
Processor	Intel Xeon E5-2697 V2, Ivy Bridge micro-architecture	
	Cores	12 @ 2.7GHz (Turbo up 3.5GHz)
	Threads	2 per Core (when Hyper-Threading is enabled)
	Sockets	2
	L1 Cache	32 KB for Instruction and 32 KB for Data per Core
	L2 Cache	256 KB per core
	L3 Cache (LLC)	30MB per Socket
Memory	2 x 32GB, 4 DDR3 channels, Max BW 60GB/s per Socket	
OS	Linux Kernel Version 2.6.32	
JVM	Oracle Hotspot JDK 7u71	
Spark	Version 1.5.0	

Table 5.5: 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		

micro-architectural performance of the workloads. Earlier studies on profiling of big data workloads show the efficacy of this method in identifying the micro-architectural bottlenecks [24, 97, 186]. 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 the metrics described in Table 7.5, whose definition are taken from Intel Vtune on-line help [4].

5.4 Evaluation

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 behavior. Figure 5.1a shows the IPC values of batch processing workloads range between 1.78 to 0.76, whereas 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 5.1b 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. Aggregation (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 5.1b. Sql Word Count (SqWc) exhibits 25.56% less back-end bound slots than

Table 5.6: Metrics for Top-Down Analysis of Workloads

Metrics	**Description**
IPC	average number of retired instructions per clock cycle
DRAM Bound	how often CPU was stalled on the main memory
L1 Bound	how often machine was stalled without missing the L1 data cache
L2 Bound	how often machine was stalled on L2 cache
L3 Bound	how often CPU was stalled on L3 cache, or contended with a sibling Core
Store Bound	how often CPU was stalled on store operations
Front-End Bandwidth	fraction of slots during which CPU was stalled due to front-end bandwidth issues
Front-End Latency	fraction of slots during which CPU was stalled due to front-end latency issues
ICache Miss Impact	fraction of cycles spent on handling instruction cache misses
Cycles of 0 ports Utilized	the number of cycles during which no port was utilized.

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 the fraction of clock cycles during which no ports are utilized, is 5.23% less than in network wordcount. The difference in performance is because RDDs use Java objects based row representation, which have high space overhead whereas DataFrames use new Unsafe Row format where rows are always 8-byte word aligned (size is multiple of 8 bytes) and equality comparison and hashing are performed on raw bytes without additional interpretation. 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 (see Table 7.2) 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). The fraction of cycles during which 0 port is utilized, however, differ only by 2.94%. Icache 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 (see Table 7.2) 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).

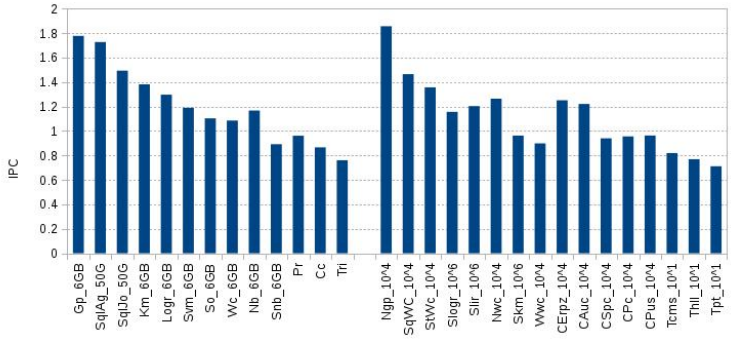
CErpz and CAuc both use "window", "map" and "groupbyKey" trans-

formations (see Table 7.2) but the front-end bound fraction and icache miss impact 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 the window length of 30 seconds is 15 x 10,000 events which are 3x larger than in CAuc with the window length of 10 seconds. This implies that with larger working sets, Icache miss impact can be reduced.

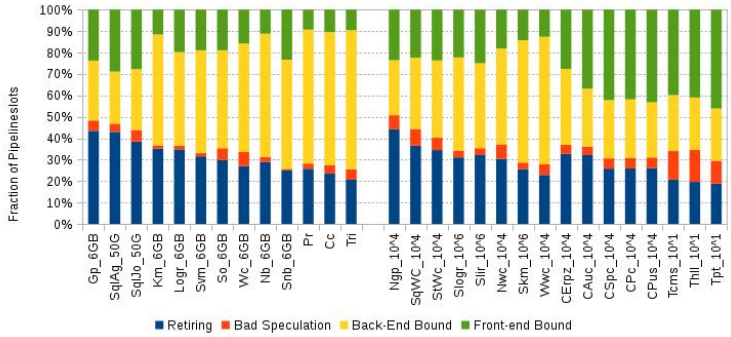
How does data velocity affect micro-architectural performance of in-memory data analytics with Spark?

In order to answer the question, we compare the micro-architectural characteristics of stream processing workloads at input data rates of 10, 100, 1000 and 10,000 events per second. Figure 5.2a shows that CPU utilization increases only modestly up to 1000 events/s after which it increases up to 20%. Likewise IPC in figure 5.2b increases by 42% in CSpc and 83% in CAuc when input rate is increased from 10 to 10,000 events per second.

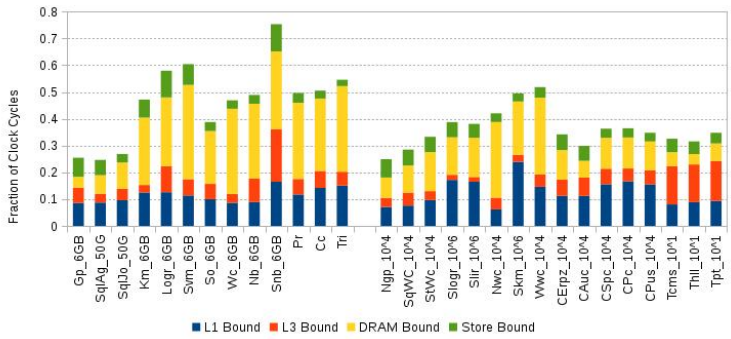
The pipeline slots breakdown in Figure 5.2c shows that when the input data rates are increased from 10 to 10,000 events/s, fraction of pipeline slots being retired increases by 14.9% in CAuc and 8.1% in CSpc because in CAuc, the fraction of front-end bound slots and bad speculation slots decrease by 9.3% and 8.1% respectively and the back-end bound slots increase by only 2.5%, whereas 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 5.2d 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 stalls and DRAM bound stalls remain roughly constant at 10, 100 and 1000 events/s because the working sets are still not large enough to create an impact but at 10,000 events/s, the working sets does not fit into the last level cache and thus DRAM bound stalls increase by approximately 20% while the L3 bound stalls decrease by the same amount. This is also evident from Figure 5.2f, 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.



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

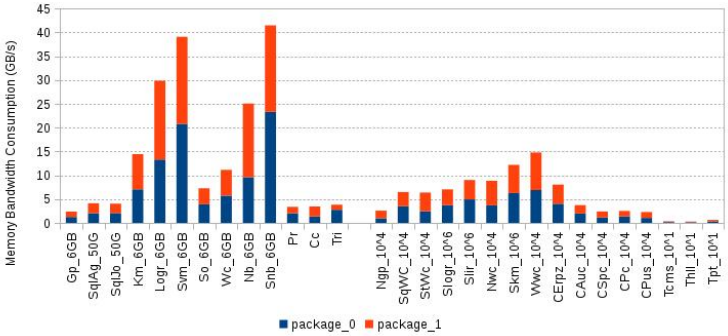


(b) Majority of stream processing workloads are back-end bound as that of batch processing workloads

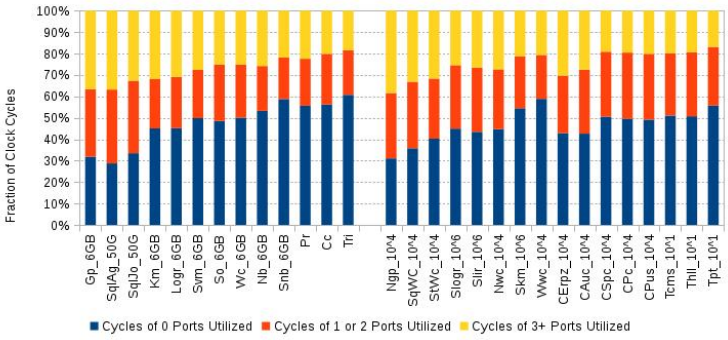


(c) Stream processing workloads are also DRAM bound but their fraction of DRAM bound stalled cycles is lower than that of batch processing workloads

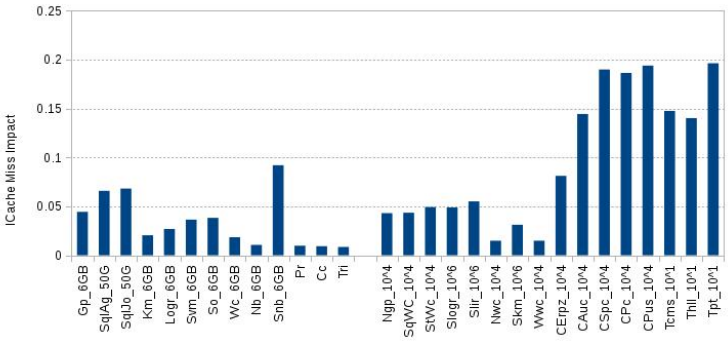
Figure 5.1: Comparison of micro-architectural characteristics of batch and stream processing workloads



(d) Memory bandwidth consumption of machine learning based batch processing workloads is higher than other Spark workloads

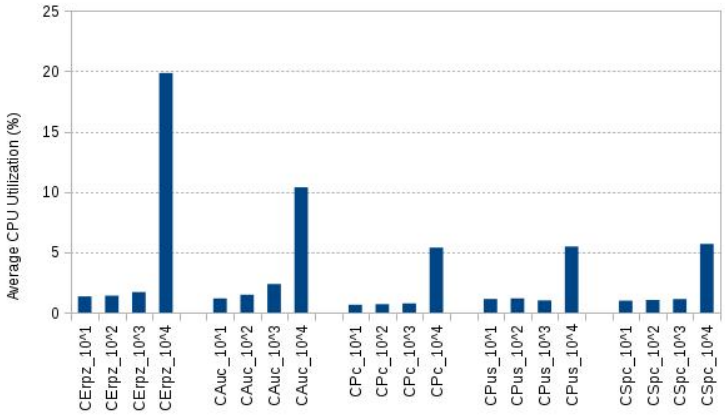


(e) Execution units starve both in batch in stream processing workloads

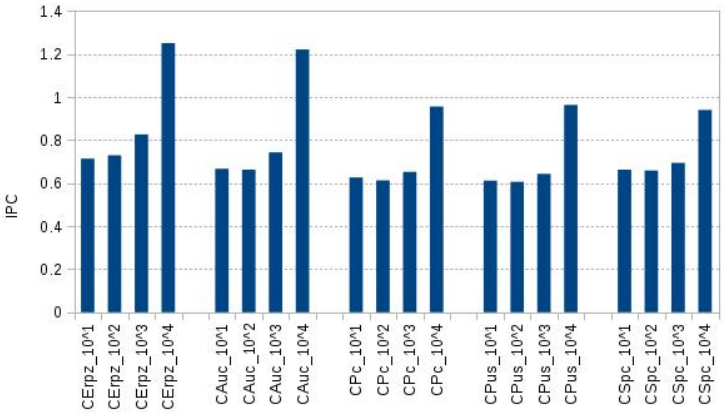


(f) ICache miss impact in majority of stream processing workloads is similar to batch processing workloads

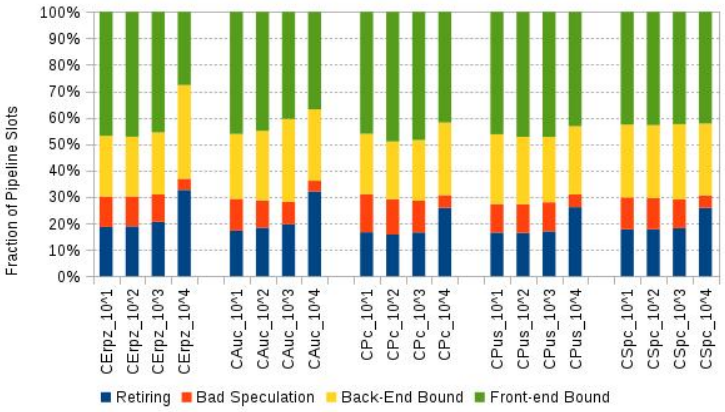
Figure 5.1: Comparison of micro-architectural characteristics of batch and stream processing workloads



(a) CPU utilization increases with data velocity

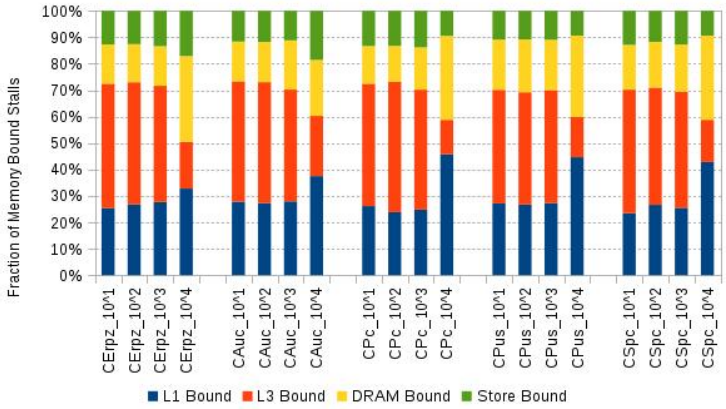


(b) Better IPC at higher data velocity

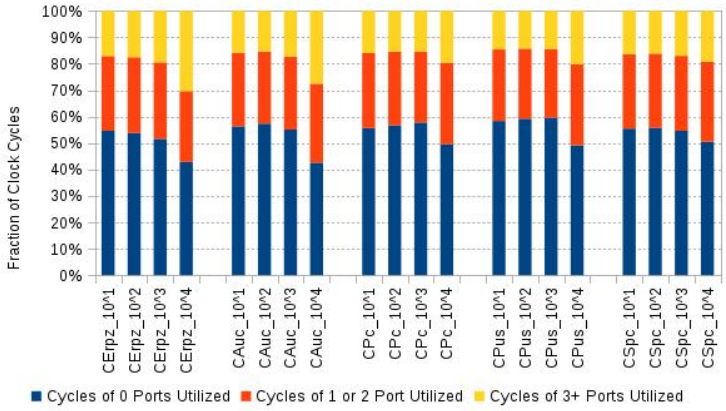


(c) Front-end bound stalls decrease and fraction of retiring slots increases with data velocity

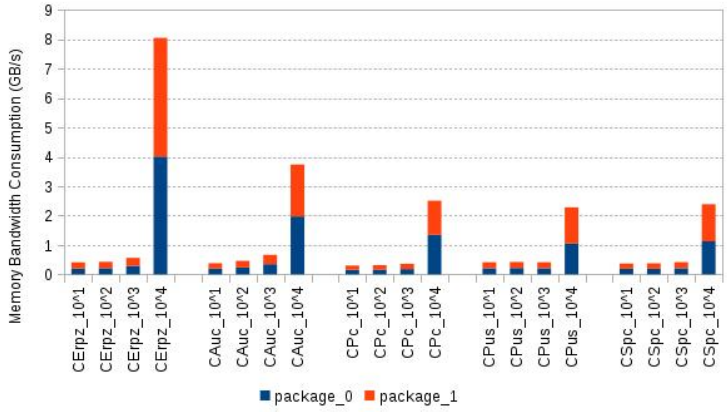
Figure 5.2: Impact of Data Velocity on Micro-architectural Performance of Spark Streaming Workloads



(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 5.2: Impact of Data Velocity on Micro-architectural Performance of Spark Streaming Workloads

5.5 Related Work

Several studies characterize the behaviour of big data workloads and identify the mismatch between the processor and the big data applications [64, 88–90, 99, 170, 186]. Ferdman et al. [64] 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. [196] 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 [88]. They also suffer from notable front-end stalls but L2 and L3 caches are effective for them. Wang et al. [170] conclude the same about L3 caches and L1 I-Cache miss rates despite using larger data sets. Deep dive analysis [186] reveal that big data analysis workload is bound on memory latency but the conclusion can not be generalized. None of the above-mentioned works consider frameworks that enable in-memory computing of data analysis workloads.

Ruirui et-al [119] have compared throughput, latency, data reception capability and performance penalty under a node failure of Apache Spark with Apache Storm. Miyuru et-al [49] have compared the performance of five streaming applications on System S and S4. Jagmon et-al [37] have analyzed the performance of S4 in terms of scalability, lost events, resource usage, and fault tolerance. Our work analyzes the micro-architectural performance of Spark Streaming.

5.6 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 behavior 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.

We recommend Spark users to prefer DataFrames over RDDs while developing Spark applications. Computer architects rely heavily on cycle accurate simulators to evaluate novel designs for processor and memory. Since simulators are quite slow, computer architects tend to pre-

fer smaller input data sets. Due to large inconsistencies in the micro-architectural behaviour with data velocity, computer architects need to simulate their proposals for stream processing workloads at large input data rates.

Chapter 6

Understanding the efficacy of architectural features in scale-up servers for In-Memory Data Analytics

6.1 Introduction

With a deluge in the volume and variety of data collecting, web enterprises (such as Yahoo, Facebook, and Google) run big data analytics applications using clusters of commodity servers. However, it has been recently reported that using clusters is a case of over-provisioning since a majority of analytics jobs do not process really big data sets and modern scale-up servers are adequate to run analytics jobs [21]. Additionally, commonly used predictive analytics such as machine learning algorithms, work on filtered data sets that easily fit into the memory of modern scale-up servers. Moreover, the today's scale-up servers can have CPU, memory, and persistent storage resources in abundance at affordable prices. Thus, we envision a small cluster of scale-up servers to be the preferable choice for processing data analytics. Choi et al. [44,45] define such clusters as scale-in clusters. They propose scale-in clusters with in-storage processing devices to reduce data movements towards CPUs. However, their proposal is based solely on the memory bandwidth characterization of in-memory data analytics with Spark and does not shed light on the specification of host CPU and memory.

While Phoenix [188], Ostrich [40] and Polymer [195] 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 [190], is getting popular in the industry because it enables in-memory processing, scales out to a large number of commodity machines and provides a unified framework for batch and stream processing of big data workloads. Like Choi et al. [44], we also favour Apache Spark to be the big data processing platform for scale-in clusters. By quantifying the architectural impact on the performance of in-memory data analytics with Spark on an Ivy Bridge server, we define the specifications of host CPU and memory and argue that a node with fixed function hardware accelerators near DRAM and NVRAM suits better for the processing of in-memory data analytics with Spark on scale-in clusters. Our contributions are:

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

6.2 Background

Spark

Spark is a cluster computing framework that uses Resilient Distributed Datasets (RDDs) [190] which are immutable collections of objects spread across a cluster. Spark programming model is based on higher-order functions that execute user-defined functions in parallel. These higher-order functions are of two types: "Transformations" and "Actions". Transformations are lazy operators that create new RDDs, whereas Actions launch a computation on RDDs and generate an output. When a user runs an action on an RDD, Spark first builds a DAG of stages from the RDD lineage graph. Next, it splits the DAG into stages that contain pipelined transformations with narrow dependencies. Further, it divides each stage into tasks, where a task is a combination of data and computation. Spark assigns tasks to the executor pool of threads and executes all tasks within a stage before moving on to the next stage. Finally, once all jobs are completed, it saves the results to file system.

Spark on Modern Scale-up Servers

Our recent efforts on identifying the bottlenecks in Spark [24,25] on scale-up server shows (i) Spark workloads exhibit poor multi-core scalability due to thread level load imbalance and work-time inflation, which is caused by frequent data accesses to DRAM and (ii) the performance of Spark workloads deteriorates severely as we enlarge the input data size due to significant garbage collection overhead and file I/O

We reproduce the multi-core scalability experiments from our previous work [24,25] to highlight the performance issues incurred by Spark workloads on scale-up servers. Each benchmark is run with 1, 6, 12, 18 and 24 executor pool threads. The size of input dataset is 6 GB. For each run, we set the CPU affinity of the Spark process to emulate hardware with the same number of cores as the worker threads. The cores are allocated from one socket first before switching to the second socket. Figure 6.1a plots speed-up as a function of the number of cores. It shows benchmarks scale linearly up to 4 cores within a socket. Beyond 4 cores, the workloads exhibit sub-linear speed-up, e.g., at 12 cores within a socket, average speed-up across workloads is 7.45. This average speed-up increases up to 8.74 when the Spark process is configured to use all 24 cores in the system. The performance gain of mere 17.3% over the 12 cores case suggest Spark applications gain less by using more than 12-core executors. Figure 6.1b shows pipeline-slots breakdown of Spark workloads.They are configured to run at 24 cores. The data show that most of the benchmarks are back-end bound because DRAM bound stalls

are the primary bottleneck (see Figure 6.1c) and remote DRAM accesses incur additional latency (see Figure 6.1d).

Simultaneous multi-threading and hardware prefetching are effective ways to hide data access latencies and additional latency overhead due to accesses to remote memory can be removed by co-locating the computations with data they access on the same socket. One reason for severe impact of garbage collection is that full generation garbage collections are triggered frequently at large volumes of input data and the size of JVM is directly related to Full GC time. Multiple smaller JVMs could be better than a single large JVM. In this chapter, we test the aforementioned techniques and study their implications on the architecture of node in scale-in cluster for in-memory data analytics with Spark.

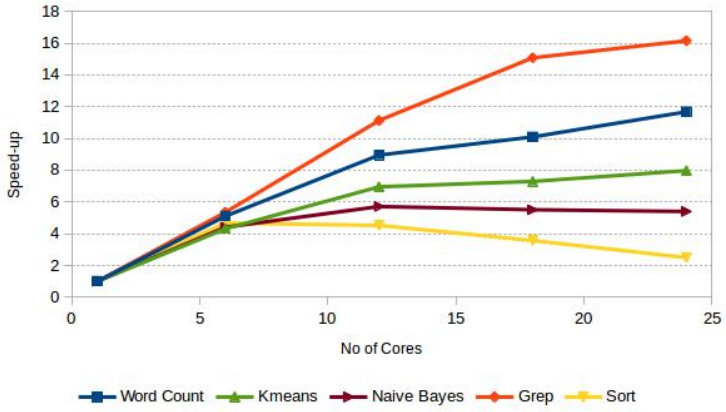
6.3 Methodology

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

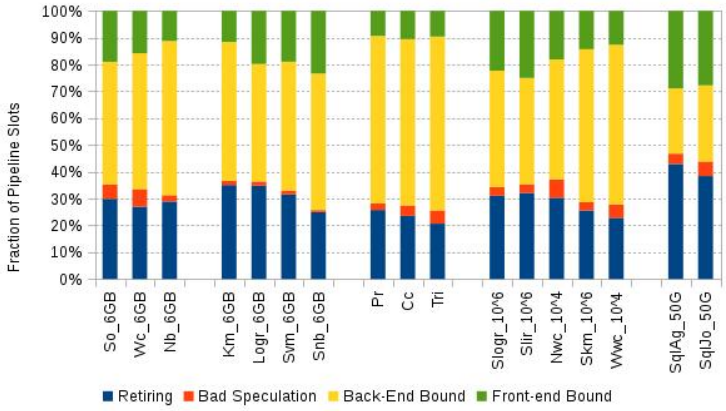
Workloads

We select the benchmarks based on following criteria;(a) workloads should cover a diverse set of Spark lazy transformations and actions, (b) workloads should be common among different big data benchmark suites available in the literature and (c) workloads have been used in the experimental evaluation of Map-Reduce frameworks. Table 7.1 shows the description of benchmarks. Batch processing workloads from Spark-core, Spark MLlib, Graph-X and Spark SQL are subset of BigdataBench [170] and HiBench [79] which are highly referenced benchmark suites in the big data domain. Stream processing workloads used in the chapter also partially cover the solution patterns for real-time streaming analytics [136].

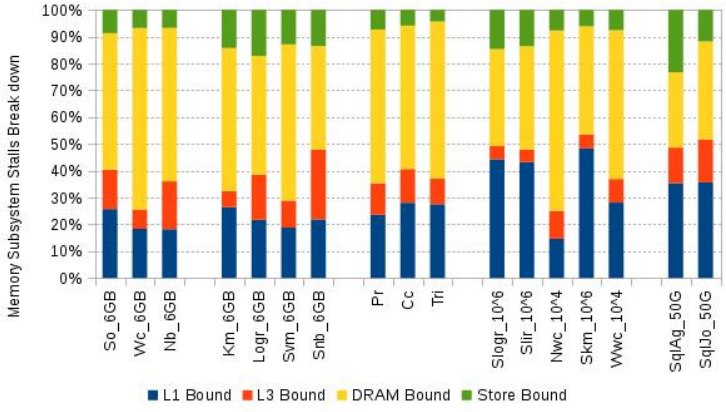
The source codes for Word Count, Grep, Sort, and NaiveBayes are taken from BigDataBench [170], whereas the source codes for K-Means, Gaussian, and Sparse NaiveBayes are taken from Spark MLlib (which is Spark's scalable machine learning library [122]) examples available along with Spark distribution. Likewise, the source codes for stream processing workloads and graph analytics are also available from Spark Streaming and GraphX examples respectively. Spark SQL queries from BigDataBench have been reprogrammed to use DataFrame API. Big



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

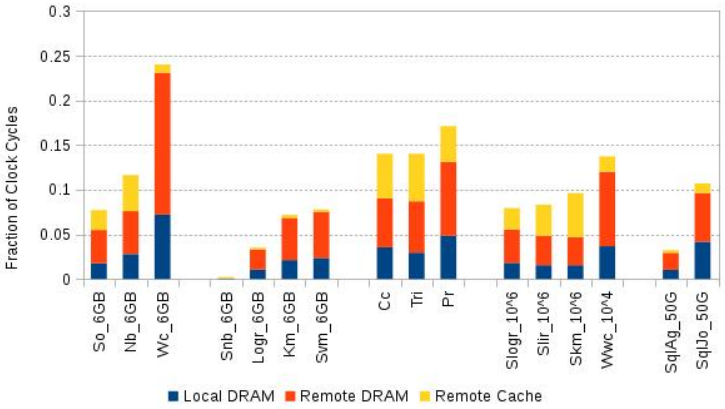


(b) Spark workloads are back-end bound



(c) Spark workloads are DRAM bound

Figure 6.1: Top Down Analysis of Spark Workloads



(d) Spark workloads have significant remote memory stalls

Figure 6.1: Top Down Analysis of Spark Workloads

Data Generator Suite (BDGS), an open source tool is used to generate synthetic data sets based on raw data sets [124].

System Configuration

To perform our measurements, we use a current dual-socket Intel Ivy Bridge server (IVB) with E5-2697 v2 processors, similar to what one would find in a datacenter. Table 7.3 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 [13]. With Hyper-Threading and Turbo-boost disabled, there are 24 cores in the system operating at the frequency of 2.7 GHz.

Table 7.4 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 [25]. The heap size is chosen such that the memory consumed is within the system. The details on Spark internal parameters are available [10].

Table 6.1: Spark Workloads

Spark Library	Workload	Description	Input data-sets
Spark Core	Word Count (Wc)	counts the number of occurrence of each word in a text file	Wikipedia Entries (Structured)
	Grep (Gp)	searches for the keyword The in a text file and filters out the lines with matching strings to the output file	
	Sort (So)	ranks records by their key	Numerical Records
	NaiveBayes (Nb)	runs sentiment classification	Amazon Movie Reviews
Spark Mllib	K-Means (Km)	uses K-Means clustering algorithm from Spark Mllib. The benchmark is run for 4 iterations with 8 desired clusters	Numerical Records (Structured)
	Gaussian (Gu)	uses Gaussian clustering algorithm from Spark Mllib. The benchmark is run for 10 iterations with 2 desired clusters	
	Sparse NaiveBayes (SNb)	uses NaiveBayes classification alogrithm from Spark Mllib	
	Support Vector Machines (Svm)	uses SVM classification alogrithm from Spark Mllib	
	Logistic Regression(Logr)	uses Logistic Regression alogrithm from Spark Mllib	
Graph X	Page Rank (Pr)	measures the importance of each vertex in a graph. The benchmark is run for 20 iterations	Live Journal Graph
	Connected Components (Cc)	labels each connected component of the graph with the ID of its lowest-numbered vertex	
	Triangles (Tr)	determines the number of triangles passing through each vertex	
Spark Streaming	Windowed Word Count (WWc)	generates every 10 seconds, word counts over the last 30 sec of,data received on a TCP socket every 2 sec.	Wikipedia Entries
	Streaming Kmeans (Skm)	uses streaming version of K-Means clustering algorithm from Spark Mllib. The benchmark is run for 4 iterations with 8 desired clusters	Numerical Records
	Streaming Logistic Regression (Slogr)	uses streaming version of Logistic Regression algorithm from Spark Mllib. The benchmark is run for 4 iterations with 8 desired clusters	
	Streaming Linear Regression (Slir)	uses streaming version of Logistic Regression algorithm from Spark Mllib. The benchmark is run for 4 iterations with 8 desired clusters	
Spark SQL	Aggregation (SqlAg)	implements aggregation query from BigdataBench using DataFrame API	Tables
	Join (SqlJo)	implements join query from BigdataBench using DataFrame API	

Table 6.2: Machine Details

Component	**Details**	
Processor	Intel Xeon E5-2697 V2, Ivy Bridge micro-architecture	
	Cores	12 @ 2.7GHz (Turbo up 3.5GHz)
	Threads	2 per Core (when Hyper-Threading is enabled)
	Sockets	2
	L1 Cache	32 KB for Instruction and 32 KB for Data per Core
	L2 Cache	256 KB per core
	L3 Cache (LLC)	30MB per Socket
Memory	2 x 32GB, 4 DDR3 channels, Max BW 60GB/s per Socket	
OS	Linux Kernel Version 2.6.32	
JVM	Oracle Hotspot JDK 7u71	
Spark	Version 1.5.0	

Table 6.3: Spark and JVM Parameters for Different Workloads

Parameters	**Batch Processing Workloads**		**Stream Processing Workloads**
	Spark-Core, Spark-SQL	**Spark Mllib, Graph X**	
spark.storage.memoryFraction	0.1	0.6	0.4
spark.shuffle.memoryFraction	0.7	0.4	0.6
spark.shuffle.consolidateFiles	true		
spark.shuffle.compress	true		
spark.shuffle.spill	true		
spark.shuffle.spill.compress	true		
spark.rdd.compress	true		
spark.broadcast.compress	true		
Heap Size (GB)	50		
Old Generation Garbage Collector	PS Mark Sweep		
Young Generation Garbage Collector	PS Scavenge		

Measurement Tools and Techniques

We configure Spark to collect GC logs which are then parsed to measure time (called real time in GC logs) spent in garbage collection. We rely on the log files generated by Spark to calculate the execution time of the benchmarks. We use Intel Vtune Amplifier [4] to perform general micro-architecture exploration and to collect hardware performance counters. We use numactl [8] to control the process and memory allocation affinity to a particular socket. We use hwloc [34] to get the CPU ID of hardware threads. We use msr-tools [7] to read and write model specific registers (MSRs). All measurement data are the average of three measure runs; Before each run, the buffer cache is cleared to avoid variation in the execution time of benchmarks. We find variance in measurements to be negligible and hence do not use box plots. Through concurrency analysis in Intel Vtune, we find executor pool threads in Spark start taking CPU time after 10 seconds. Hence, hardware performance counter values are collected after the ramp-up period of 10 seconds. For batch processing workloads, the measurements are taken for the entire run of the applications and for stream processing workloads, the measurements are taken for 180 seconds as the sliding interval and duration of windows in streaming workloads considered are much less than 180 seconds.

Top-Down Analysis Approach

We use top-down analysis method proposed by Yasin [185] to study the micro-architectural performance of the workloads because earlier studies on profiling on big data workloads shows the efficacy of this method in identifying the micro-architectural bottlenecks [24,97,186]. Super-scalar processors can be conceptually divided into the "front-end" where instructions are fetched and decoded into constituent operations, and the "back-end" where the required computation is performed. A pipeline slot represents the hardware resources needed to process one micro-operation. The top-down method assumes for each CPU core, there are four pipeline slots available per clock cycle. At issue point, each pipeline slot is classified into one of four base categories: Front-end Bound, Back-end Bound, Bad Speculation and Retiring. If a micro-operation is issued in a given cycle, it would eventually either get retired or cancelled. Thus, it can be attributed to either Retiring or Bad Speculation respectively. Pipeline slots that could not be filled with micro-operations due to problems in the front-end are attributed to Front-end Bound category whereas pipeline slot where no micro-operations are delivered due to a lack of required resources for accepting more micro-operations in the back-end of the pipeline are identified as Back-end Bound. The top-down method requires following the metrics described in Table 7.5, whose definition

Table 6.4: Metrics for Top-Down Analysis of Workloads

Metrics	**Description**
IPC	average number of retired instructions per clock cycle
DRAM Bound	how often CPU was stalled on the main memory
L1 Bound	how often machine was stalled without missing the L1 data cache
L2 Bound	how often machine was stalled on L2 cache
L3 Bound	how often CPU was stalled on L3 cache, or contended with a sibling Core
Store Bound	how often CPU was stalled on store operations
Front-End Bandwidth	fraction of slots during which CPU was stalled due to front-end bandwidth issues
Front-End Latency	fraction of slots during which CPU was stalled due to front-end latency issues
ICache Miss Impact	fraction of cycles spent on handling instruction cache misses
DTLB Overhead	fraction of cycles spent on handling first-level data TLB load misses
Cycles of 0 ports Utilized	the number of cycles during which no port was utilized.

are taken from Intel Vtune on-line help [4].

6.4 Evaluation

How much performance gain is achievable by co-locating the data and computations on NUMA nodes for in-memory data analytics with Spark?

Ivy Bridge Server is a NUMA multi-socket system. Each socket has 2 on-chip memory controllers and a part of the main memory is directly connected to each socket. This layout offers high bandwidth and low access latency to the directly connected part of the main memory. The sockets are connected by two QPI (Quick Path Interconnect) links, thus,

a socket can access the main memory of another socket. However, a memory access from one socket to memory from another socket (remote memory access) incurs additional latency overhead due to transferring the data by cross-chip interconnect. By co-locating the computations with the data they access, the NUMA overhead can be avoided.

To evaluate the impact of NUMA on Spark workloads, we run the benchmarks in two configurations: a) Local DRAM, where Spark process is bound to socket 0 and memory node 0, i.e. computations and data accesses are co-located, and b) Remote DRAM, where spark process is bound to socket 0 and memory node 1, i.e. all data accesses incur the additional latency. The input data size for the workloads is chosen as 6GB to ensure memory working set sizes fit socket memory. Spark parameters for the two configurations are given in Table 6.5.

Table 6.5: Machine and Spark Configuration for NUMA Evaluation

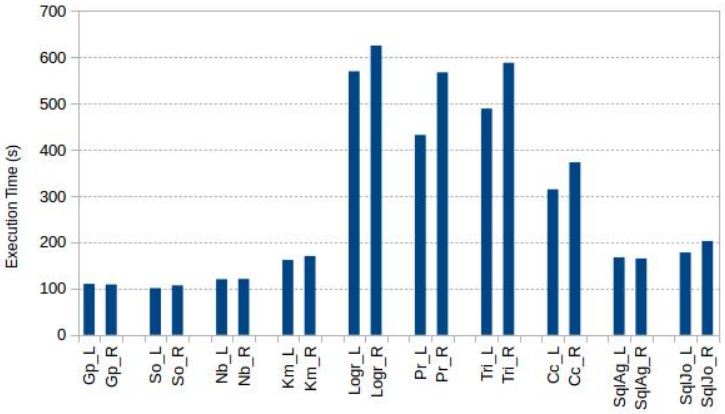
		Local DRAM (L)	**Remote DRAM (R)**
Hardware	Socket ID	0	0
	Memory Node ID	0	1
	No. of cores	12	12
	No. of threads	12	12
Spark	spark.driver.cores	12	12
	spark.default.parallelism	12	12
	spark.driver.memory (GB)	24	24

Figure 6.2a shows remote memory accesses can degrade the performance of Spark workloads by 10% on average. This is because despite the stalled cycles on remote memory accesses double (see Figure 6.2c), retiring category degrades by only 10.79%, Back-end bound stalls increases by 20.26%, bad speculation decreases by 13.08% and front-end bound stalls decreases by 12.66% on average as shown in Figure 6.2b. Furthermore, the total cross-chip bandwidth of 32 GB/sec (peak bandwidth of 16 GB/s per QPI link) satisfies the memory bandwidth requirements of Spark workloads (see Figure 6.2d).

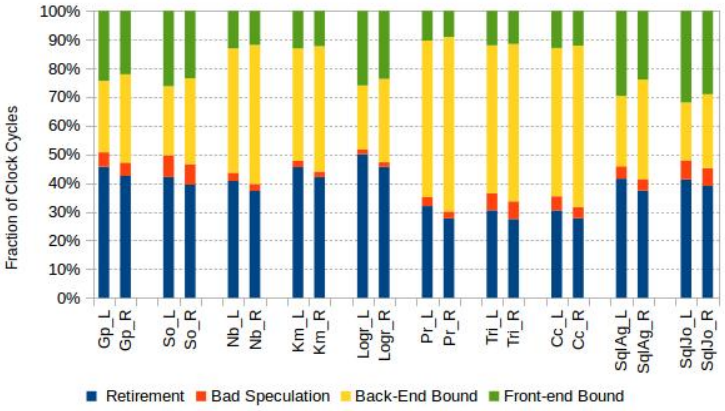
Implications: In-memory data analytics with Spark should use data from local memory on a multi-socket node of the scale-in cluster.

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

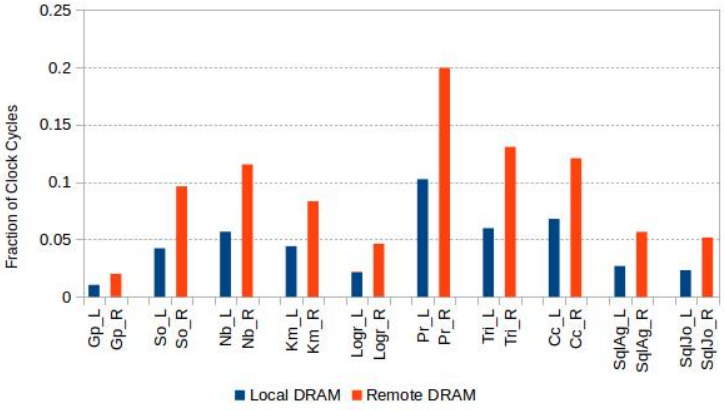
Ivy Bridge Machine uses Simultaneous Multi-threading(SMT), which enables one processor core to run two software threads simultaneously to hide data access latencies. To evaluate the effectiveness of Hyper-Threading, we run Spark process in the three different configurations a) ST:2x1, the baseline single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



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

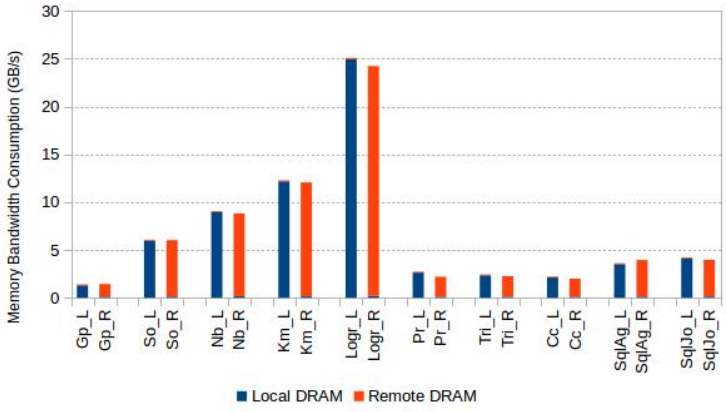


(b) Retiring decreases due to increased back-end bound in remote only mode.



(c) Stalled Cycles doubles in remote memory case

Figure 6.2: NUMA Characterization of Spark Benchmarks



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

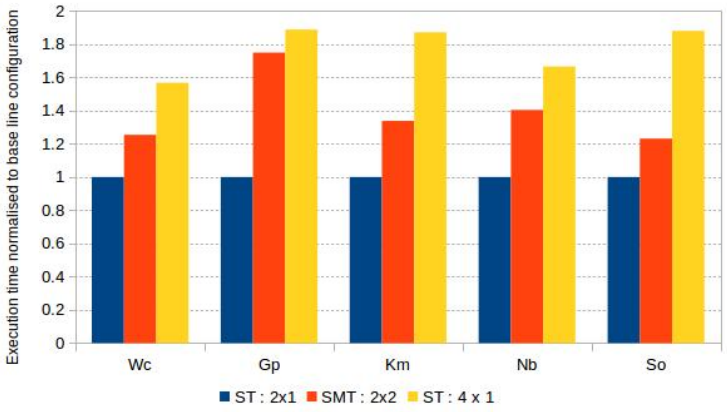
Figure 6.2: NUMA Characterization of Spark Benchmarks

bound to two physical cores b) SMT:2x2, a simultaneous multi-threaded configuration where Spark process is allowed to use 2 physical cores and their corresponding hyper threads and c) ST:4x1, the upper-bound single threaded configuration where Spark process is allowed to use 4 physical cores. Spark parameters for the aforementioned configurations are given in Table 6.6. We also experiment with baseline configurations, ST:1x1, ST:3x3, ST:4x4, ST:5x5 and ST:6x6. In all experiments socket 0 and memory node 0 is used to avoid NUMA effects and the size of input data for the workloads is 6GB.

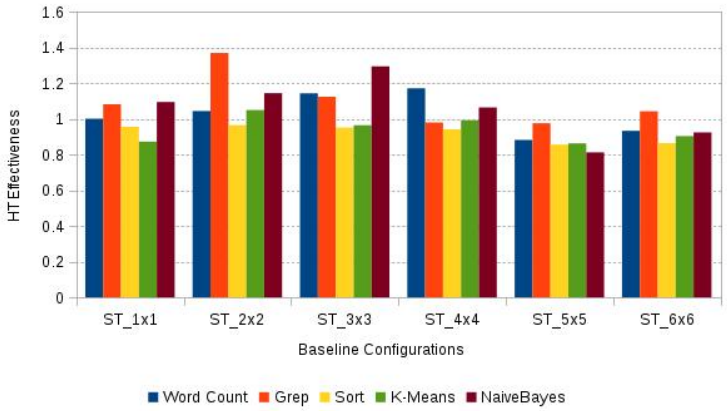
Table 6.6: Machine and Spark Configurations to evaluate Hyper Threading

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

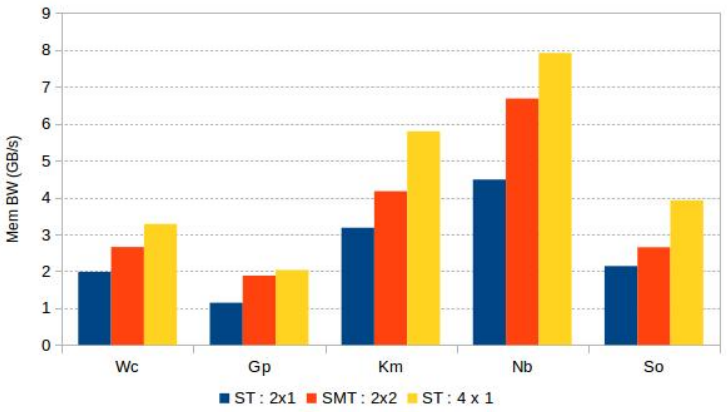
Figure 6.3a shows SMT provides 39.5% speedup on average across the workloads over baseline configuration, while the upper-bound configuration provided 77.45% on average across the workloads. The memory bandwidth in SMT case also keeps up with the multi-core case it is



(a) Multi-core vs Hyper-Threading

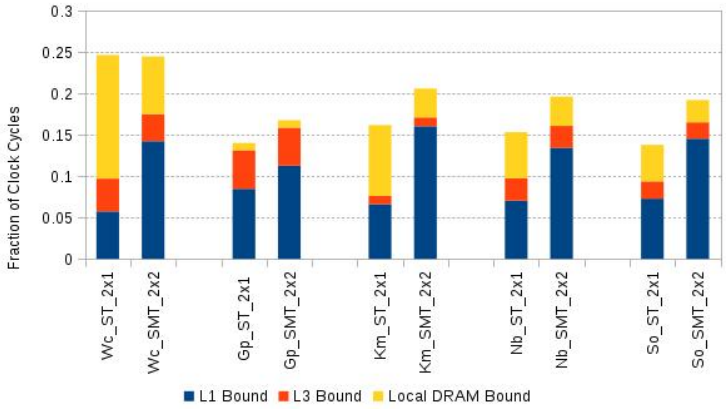


(b) HT Effectiveness is around 1



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

Figure 6.3: Hyper Threading is Effective



(d) DRAM Bound decreases and L1 Bound increases

Figure 6.3: Hyper Threading is Effective

20.54% less than that of the multi-core version on average across the workloads as shown in Figure 6.3c. Figure 6.3b presents HT Effectiveness at different baseline configurations. HT Effectiveness of 1 is desirable as it implies 30% performance improvement in Hyper-Threading case over the baseline single threaded configuration [2]. The data reveal HT effectiveness remains close to 1 on average across the workloads till 4 cores after that it drops. This is because of poor multi-core scalability of Spark workloads as shown in [24]

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 6.3d implying that Hyper-threading is effective in hiding the memory access latency for Spark workloads.

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

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

Prefetching is a promising approach to hide memory access latency by predicting the future memory accesses and fetching the corresponding memory blocks into the cache ahead of explicit accesses by the processor. Intel Ivy Bridge Server has two L1-D prefetchers and two L2 prefetchers.The description about prefetchers is given in Table 6.7. This information is taken from Intel software forum [1].

Table 6.7: Hardware Prefetchers Description

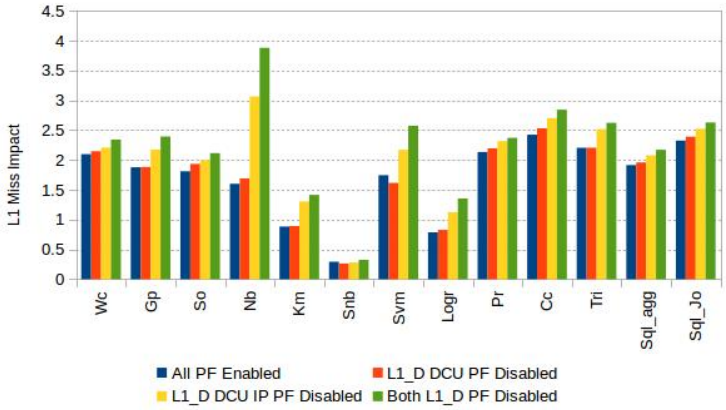
Prefetcher	**Bit No. in MSR (0x1A4)**	**Description**
L2 hardware prefetcher	0	Fetches additional lines of code or data into the L2 cache
L2 adjacent cache line prefetcher	1	Fetches the cache line that comprises a cache line pair(128 bytes)
DCU prefetcher	2	Fetches the next cache line into L1-D cache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

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

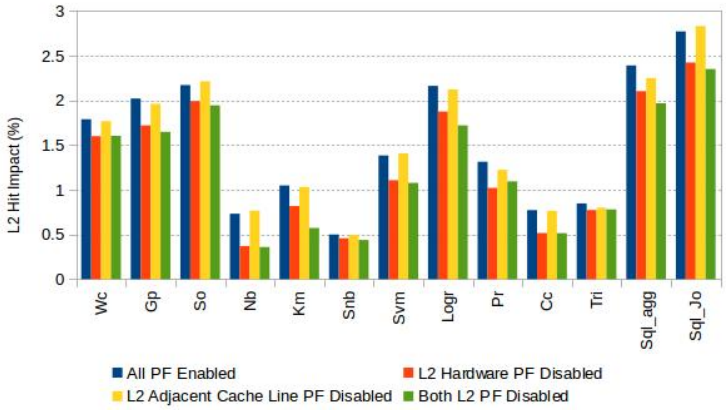
Figure 6.4a shows L1-D miss impact increases by only 3.17% on average across the workloads when DCU prefetcher disabled, whereas the same metric increases by 34.13% when DCU IP prefetcher is disabled in comparison with the case when all processor prefetchers are enabled. It implies DCU prefetcher is ineffective.

Figure 6.4b shows L2 miss rate increases by at most 5% in Grep when L2 adjacent cache line prefetcher disabled. In some cases for example sort and naivebayes, disabling L2 adjacent line prefetcher reduces the L2 miss rate. This implies L2 adjacent cache line prefetcher is ineffective. It also shows L2 miss rate increases by 14.31% on average across the workloads when L2 hardware prefetcher is disabled.

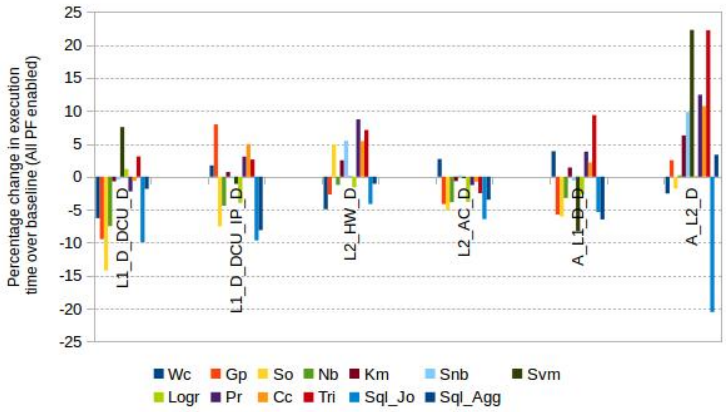
Figure 6.4c shows percentage change in execution time of Spark workloads over baseline configuration (all prefetchers are enabled). The data show L1-D next-line and adjacent cache line L2 prefetchers have a negative impact on Spark workloads and disabling them improves the performance of Spark workloads on average by 7.9% and 2.31% respectively. This implies simple next-line hardware prefetchers in modern scale-up servers are ineffective for in-memory data analytics.



(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 6.4: Evaluation of Hardware Prefetchers

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

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. [85], the bandwidth vs latency response curve for a system has three regions. For the first 40% of the sustained bandwidth, the latency response is nearly constant. The average memory latency equals idle latency in the system and the system performance is unbounded by the memory bandwidth in the constant region. In between 40% to 80% of the sustained bandwidth, the average memory latency increases almost linearly due to contention overhead by numerous memory requests. The performance degradation of the system starts in this linear region. Between 80% to 100% of the sustained bandwidth, the memory latency can increase exponentially over the idle latency of DRAM system and the applications performance is limited by available memory bandwidth in this exponential region. Note that maximum sustained bandwidth is 65% to 75% of the theoretical maximum for server workloads.

Using the formula taken from Intel's document [13], we calculate maximum theoretical bandwidth, per socket, for a processor with DDR3-1866 and 4 channels is 59.7GB/s and the total system bandwidth is 119.4 GB/s. To find sustained maximum bandwidth, we compile the OpenMP version of STREAM [11] using Intel's ICC compiler. On running the benchmark, we find the maximum sustained bandwidth to be 92 GB/s.

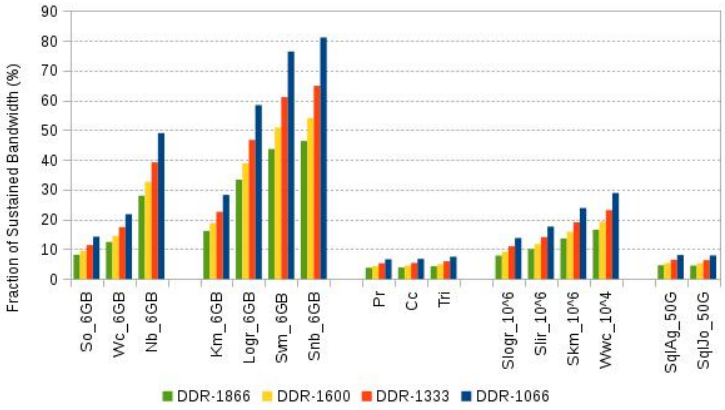


Figure 6.5: Spark workloads do not experience loaded latencies

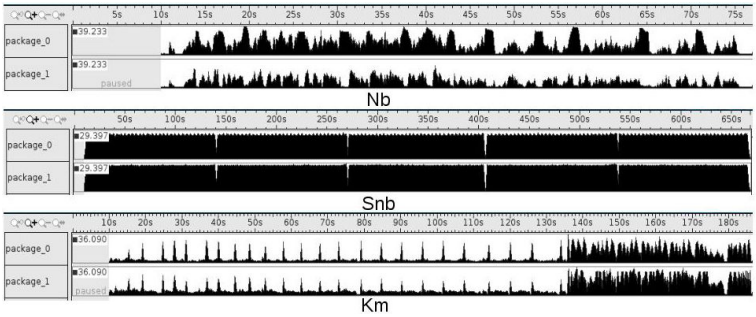
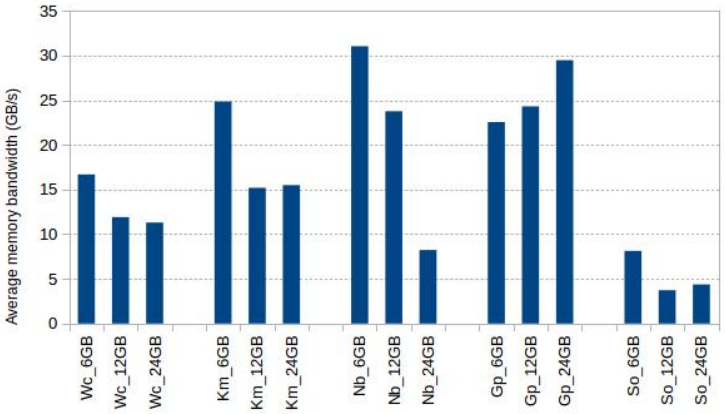


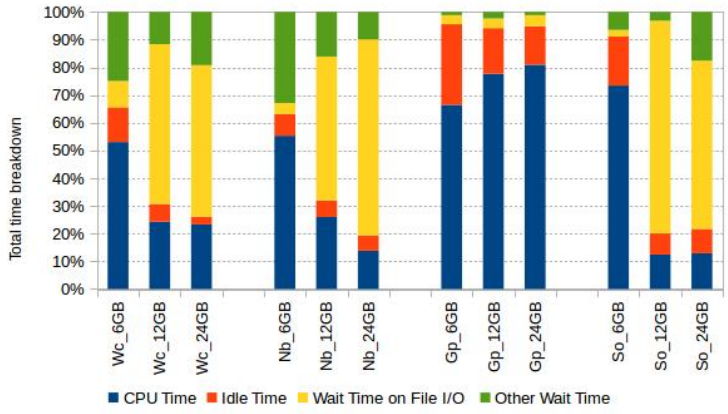
Figure 6.6: Bandwidth Consumption over time

Figure 7.1d shows the average bandwidth consumption as a fraction of sustained maximum bandwidth for different BIOS configurable data transfer rates of DDR3 memory. The data reveal Spark workloads consume less than 40% of sustained maximum bandwidth at 1866 data transfer rate and thus operate in the constant region. By lowering the data transfer rates to 1066, the majority of workloads from Spark core, all workloads from Spark SQL, Spark Streaming, and Graph-X still operate on the boundary of linear region whereas workloads from Spark MLlib shift to the linear region and mostly operate at the boundary of linear and exponential region. However at 1333, Spark MLlib workloads operate roughly in the middle of the linear region. From the bandwidth consumption over time curves of the Km, Snb and Nb in Figure 6.6, it can be seen even when the peak bandwidth utilization goes into the exponential region, it lasts only for a short period of time and thus, have a negligible impact on the performance. As we enlarge the input data set, Figure 6.7a shows average memory bandwidth consumption decreases from 20.7 GB/s in the 6 GB case to 13.7 GB/s in the 24 GB case on average across the workloads. Moreover, wait time on file I/O becomes dominant at large input data sets as shown in Figure 6.7b.

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



(a) Memory traffic decreases with data size.



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

Figure 6.7: Effect of Data Volume on Spark workloads

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

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

We deploy Spark in standalone mode on a single machine, i.e. master and worker daemons run on the same machine. We run applications with 1, 2, 4 and 6 executors. Beyond 6, we hit the operating system limit of a maximum number of threads in the system. Table 6.8 lists down the configuration details. In all configurations, the total number of cores and the total memory used by the applications are constant at 24 cores and 50GB respectively.

Table 6.8: Multiple Executors Configuration

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

Figure 6.8 data shows 2 executors configuration are better than 1 executor configuration, e.g. for K-Means and Gaussian, 2E configuration provides 29.31% and 30.43% performance improvement over the baseline 1E configuration, however, 6E configuration only increases the performance gain to 36.48% and 35.47% respectively. For the same workloads, GC time in 6E case is 4.08x and 4.60x less than the 1E case. A small performance gain from 2E to 6E despite the reduction in GC time can be attributed to increased communication overhead among the executors and master.

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

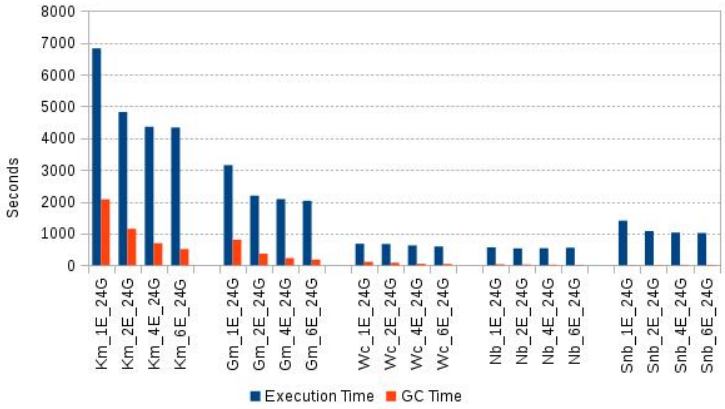


Figure 6.8: Multiple small executors are better than single large executor due to reduction in GC time

6.5 The case of Near Data Computing both in DRAM and in Storage

Since DRAM scaling is lagging behind the Moore's law, increasing DRAM capacity will be a challenge. NVRAM, on the other hand, shows a promising trend in terms of capacity scaling. Since Spark based workloads are I/O intensive when the input datasets don't fit in memory and are bound on latency when they do fit in-memory, In-Memory processing, and In-storage processing can be combined together into a hybrid architecture where the host is connected to DRAM with custom accelerators and flash based NVRAM with integrated hardware units to reduce the data movement. We envision a single node with fixed function hardware accelerators both in DRAM and also in-Storage. Figure 9.1 shows the architecture.

Let's consider an example. Many transformations in Spark such as groupByKey, reduceByKey, sortByKey, join etc involve shuffling of data between the tasks. To organize the data for shuffle, spark generates set of tasks; map tasks to organize the data and a set of reduce tasks to aggregate it. Map output records from each task are kept in memory until they can't fit. At that point records are sorted by reduce tasks for which they are destined and then spilled to a single file. Since the records are dispersed throughout the memory, they results in poor cache locality and sorting them on CPU will experience a significant amount of cache misses and using near DRAM hardware accelerators for sort function, this phase can be accelerated. If this process occurs multiple times, the spilled segments are merged later. On the reduce side, tasks read

the relevant sorted blocks. A single reduce task can receive blocks from thousands of map tasks. To make this many-way merge efficient, especially in the case where the data exceeds the memory size, It is better to use hardware accelerators for merge function near the faster persistent storage device like NVRAM.

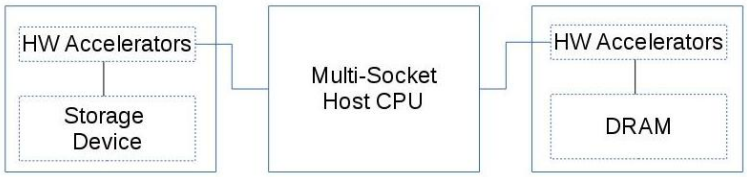


Figure 6.9: NDC Supported Single Node in Scale-in Clusters for in-Memory Data Analytics with Spark

6.6 Related Work

Several studies characterize the behaviour of big data workloads and identify the mismatch between the processor and the big data applications [30, 64, 88–90, 97, 99, 170, 186]. None of the above-mentioned works analyze the impact of NUMA, SMT and hardware prefetchers on the performance of in-memory data analytics with Apache Spark.

Chiba et al. [41] also study the impact of NUMA, SMT and multiple executors on the performance of TPC-H queries with Apache Spark on IBM Power 8 server. However, their work is limited to Spark SQL library only. They only explore thread affinity, i.e. bind JVMs to sockets but allow the cross socket accesses. Our study covers the workloads not only from Spark SQL but also from Spark-core, Spark MLlib, Graph X and Spark Streaming. We use Intel Ivy bridge server. By using a diverse category of Spark workloads and a different hardware platform, our findings build upon Chiba's work. We give in-depth insights into the limited potential of NUMA affinity for Spark SQL workloads, e.g. Spark SQL queries exhibit 2-3% performance improvement by considering NUMA locality whereas Graph-X workloads show more than 20% speed-up because CPU stalled cycles on remote accesses are much less in Spark SQL queries compared to Graph-X workloads. We show the effectiveness of hyper-threading is due to the reduction in DRAM bound stalls and also show that HT is effective for Spark workloads only up to 6 cores. Besides that, we also quantify the impact of existing hardware prefetchers in scale-up servers on Spark workloads and quantify the DRAM speed sufficient for Spark workloads. Moreover, we derive insights about the

architecture of a node in scale-in cluster for in-memory data analytics based on their performance characterization.

6.7 Conclusion

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

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

We advise using executors with memory size less than or equal to 32GB and restrict each executor to use NUMA-local memory. We recommend enabling hyper-threading, disable next-line L1-D and adjacent cache line L2 prefetchers and lower the DDR3 speed to 1333.

We also envision processors with 6 hyper-threaded cores without L1-D next line and adjacent cache line L2 prefetchers. The die area saved can be used to increase the LLC capacity and the use of high bandwidth memories like Hybrid memory cubes [3] is not justified for in-memory data analytics with Spark.

Chapter 7

The Case of Near Data Processing Servers for In-Memory Data Analytics

7.1 Introduction

With a deluge in the volume and variety of data collecting, web enterprises (such as Yahoo, Facebook, and Google) run big data analytics applications using clusters of commodity servers. While cluster computing frameworks are continuously evolving to provide real-time data analysis capabilities, Apache Spark [190] has managed to be at the forefront of big data analytics for being a unified framework for SQL queries, machine learning algorithms, graph analysis and stream data processing. Recent studies on characterizing in-memory data analytics with Spark show that (i) in-memory data analytics are bound by the latency of frequent data accesses to DRAM [24] and (ii) their performance deteriorates severely as we enlarge the input data size due to significant wait time on I/O [25].

The concept of near-data processing (NDP) is regaining the attention of researchers partially because of technological advancement and partially because moving the compute closer to the data where it resides, can remove the performance bottlenecks due to data movement. The umbrella of NDP covers 2D-integrated Processing-In-Memory, 3D-stacked Processing-In-Memory (PIM) and In-Storage Processing (ISP). Existing studies show efficacy of processing-in-memory (PIM) approach for simple map-reduce applications [83, 137], graph analytics [15, 127], machine learning applications [31, 107] and SQL queries [125, 177]. Researchers also show the potential of processing in non-volatile memories for I/O bound big data applications [36,143,173]. However, it is not clear which aspect of NDP (high bandwidth, improved latency, reduction in data movement, etc..) will benefit state-of-art big data frameworks like Apache Spark. Before quantifying the performance gain achievable by NDP for Spark, it is pertinent to answer which form of NDP (PIM, ISP) would better suit Spark workloads?

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

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

ing compute demands of Apache Spark based workloads.

7.2 Background and Related Work

Spark

Spark is a cluster computing framework that uses Resilient Distributed Datasets (RDDs), which are immutable collections of objects spread across a cluster. Spark programming model is based on higher-order functions that execute user-defined functions in parallel. These higher-order functions are of two types: "Transformations" and "Actions". Transformations are lazy operators that create new RDDs, whereas Actions launch a computation on RDDs and generate an output. When a user runs an action on an RDD, Spark first builds a DAG of stages from the RDD lineage graph. Next, it splits the DAG into stages that contain pipe-lined transformations. Further, it divides each stage into tasks, where a task is a combination of data and computation. Tasks are assigned to executor pool of threads. Spark executes all tasks within a stage before moving on to the next stage. Finally, once all jobs are completed, the results are saved to file systems.

Spark MLlib is a scalable machine learning library [122] on top of Spark Core. GraphX enables graph-parallel computation in Spark. Spark SQL is a Spark module for structured data processing with data schema information. This schema information is used to perform extra optimization. Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. Internally, a DStream is represented as a sequence of RDDs. Spark streaming can receive input data streams from sources such as Apache Kafka [103]. It then divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

Near Data Processing

The umbrella of near-data processing covers both processing in memory and in-storage processing. A survey [157] highlights historical achievements in technology that enables Processing-In-Memory (PIM) and various PIM architectures. It depicts PIM's advantages and challenges. Challenges of PIM architecture design are the cost-effective integration of logic and memory, unconventional programming models and lack of inter-operability with caches and virtual memory.

PIM approach can reduce the latency and energy consumption associated with moving data back-and-forth through the cache and memory hierarchy, as well as greatly increase memory bandwidth by sidestepping the conventional memory-package pin-count limitations. There exists a

continuum of processing that can be embedded "in memory" [115]. This includes i) software transparent applications of logic in memory, ii) fixed function accelerators, iii) bounded operand PIM operations, which can be specified in a manner that is consistent with existing instruction-level memory operand formats, directly encoded in the opcode in the instruction set architecture, iv) compound PIM operations, which may access an arbitrary number of memory locations and perform number of different operations and v) fully programmable logic in memory, either a processor or re-configurable logic device.

Related work for NDP

Applications of PIM

PIM for Map-Reduce: For Map-Reduce applications, prior studies [83, 138] propose simple processing cores in the logic layer of 3D-stacked memory devices to perform Map operations with efficient data access and without hitting the memory bandwidth wall. The reduce operations despite having random memory access patterns are performed on the central host processor.

PIM for Graph Analytics: The performance of graph analytics is bound by the inability of conventional processing systems to fully utilize the memory bandwidth and Ahn et al. [15] propose in-order cores with graph processing specific prefetchers in the logic layer of 3D-stacked DRAM to fully utilize the memory bandwidth. Graph traversals are bounded by irregular memory access patterns of graph property and a study [127] proposes to offload the graph property to hybrid memory cube [3] (HMC) by utilizing the atomic requests described in HMC 2.0 specification (that is limited to only integer operations and one memory operand).

PIM for Machine Learning: Lee et al. [107] use State Synchronous Parallel (SSP) model to evaluate asynchronous parallel machine learning workloads and observe that atomic operations are the hotspots and propose to offload them onto logic layers in 3D stacked memories. These atomic operations are overlapped with main computation to increase the execution efficiency. K-means, a popular machine learning algorithm, is shown to benefit from higher bandwidth achieved by physically bonding the memory to the package containing processing elements [31]. Another proposal [52] is to use content addressable memories with hamming distance units in the logic layer to minimize the impact of significant data movement in k-nearest neighbours.

PIM for SQL queries: Researchers also exploit PIM for SQL queries. The motivation for pushing select query down to memory is reduce data movement by pushing only relevant data up the memory hierarchy [177].

Join query can exploit 3D stacked PIM as it is characterized by irregular access patterns, but near-memory algorithms are required that consider data placement and communication cost and exploit locality with in one stack as much as possible [125]

PIM for Data Re-organization operations: Another application of PIM is to accelerate data access and to help CPU cores to compute on complex linked data structures by efficiently packing them into the cache. Using strided DMA units, gather/scatter hardware and in-memory scratchpad buffers, the programmable near memory data rearrangement engines proposed in [69] perform fill and drain operations to gather the blocks of application data structures.

In-Storage Processing

Ranganathan et al. [143] propose nano-stores that co-locates processors and non-volatile memory on the same chip and connect to one another to form a large cluster for data-centric workloads that operate on more diverse data with I/O intensive, often random data access patterns and limited locality. Chang et al. [36] examine the potential and limit of designs that move compute in close proximity of NVM based data stores. The limit study demonstrates significant potential of this approach (3-162x improvement in energy-delay product) particularly for I/O intensive workloads. Wang et al. [173] 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.

7.3 Big Data Frameworks and NDP

Motivation

Even though NDP seems promising for applications like map-reduce, machine learning algorithms, SQL queries and graph analytics, but the existing literature lacks a study that identifies the potential of NDP for big data processing frameworks like Apache Spark, which run on top of Java Virtual Machine and use map-reduce programming model to enable machine learning, graph analysis and SQL processing on batched and streaming data. One can argue that previous NDP proposals made only by studying the algorithms can be extrapolated to the big data frameworks but we refute the argument by stating that earlier proposal of using 3D-Stacked PIM for map reduce applications [83, 138] was motivated by the fact that the performance of map phase is limited by the memory bandwidth. Our experiments show that Apache Spark based

map-reduce workloads don't fully utilize the available memory bandwidth. Prior work [27] also shows that high bandwidth memories are not needed for Apache Spark based workloads.

Methodology

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

Workloads

Our selection of benchmarks is inspired by [27]. We select the benchmarks based on following criteria;(a) workloads should cover a diverse set of Spark lazy transformations and actions, (b) workloads should be common among different big data benchmark suites available in the literature and (c) workloads have been used in the experimental evaluation of Map-Reduce frameworks. Table 7.1 shows the description of benchmarks and the breakdown of each benchmark into transformations and actions are given in Table 7.2. Batch processing workloads from Spark-core, Spark MLlib, Graph-X and Spark SQL are subset of BigdataBench [170] and HiBench [79] which are highly referenced benchmark suites in the big data domain. Stream processing workloads used in the chapter also partially cover the solution patterns for real-time streaming analytics [136].

The source codes for Word Count, Grep, Sort, and NaiveBayes are taken from BigDataBench [170], whereas the source codes for K-Means, Gaussian, and Sparse NaiveBayes are taken from Spark MLlib (which is Spark's scalable machine learning library [**?**]) examples available along with Spark distribution. Likewise, the source codes for stream processing workloads and graph analytics are also available from Spark Streaming and GraphX examples respectively. Spark SQL queries from BigDataBench have been reprogrammed to use DataFrame API. Big Data Generator Suite (BDGS), an open source tool is used to generate synthetic data sets based on raw data sets [124].

System Configuration

To perform our measurements, we use a current dual-socket Intel Ivy Bridge server (IVB) with E5-2697 v2 processors, similar to what one would find in a datacenter. Table 7.3 shows details about our test machine. Hyper-Threading and Turbo-boost are disabled through BIOS during the experiments as per Intel Vtune guidelines to tune software on the Intel Xeon processor E5/E7 v2 family [13]. With Hyper-Threading

Table 7.1: Spark Workloads

Spark Library	**Workload**	**Description**	**Input data-sets**
Spark Core	Word Count (Wc)	counts the number of occurrence of each word in a text file	Wikipedia Entries
	Grep (Gp)	searches for the keyword The in a text file and filters out the lines with matching strings to the output file	
	Sort (So)	ranks records by their key	Numerical Records
	NaiveBayes (Nb)	runs sentiment classification	Amazon Movie Reviews
Spark MLlib	K-Means (Km)	uses K-Means clustering algorithm from Spark MLlib. The benchmark is run for 4 iterations with 8 desired clusters	Numerical Records
	Sparse NaiveBayes (Snb)	uses NaiveBayes classification algorithm from Spark MLlib	
	Support Vector Machines (Svm)	uses SVM classification algorithm from Spark MLlib	
	Logistic Regression(Logr)	uses Logistic Regression algorithm from Spark MLlib	
Graph X	Page Rank (Pr)	measures the importance of each vertex in a graph. The benchmark is run for 20 iterations	Live Journal Graph
	Connected Components (Cc)	labels each connected component of the graph with the ID of its lowest-numbered vertex	
	Triangles (Tr)	determines the number of triangles passing through each vertex	
Spark SQL	Aggregation (Sql_Agg)	implements aggregation query from BigdataBench using DataFrame API	Tables
	Join (Sql_Jo)	implements join query from BigdataBench using DataFrame API	
	Difference (Sql_Diff)	implements difference query from BigdataBench using DataFrame API	
	Cross Product (Sql_Cro)	implements cross product query from BigdataBench using DataFrame API	
	Order By (Sql_Ord)	implements order by query from BigdataBench using DataFrame API	
Spark Streaming	Windowed Word Count (WWc)	generates every 10 seconds, word counts over the last 30 sec of data received on a TCP socket every 2 sec.	Wikipedia Entries
	Stateful Word Count (StWc)	counts words cumulatively in text received from the network every sec starting with initial value of word count.	
	Network Word Count (NWc)	counts the number of words in the text, received from a data server listening on a TCP socket every 2 sec and print the counts on the screen. A data server is created by running Netcat (a networking utility in Unix systems for creating TCP/UDP connections)	

Table 7.2: Converted Spark Operations in Workloads

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

and Turbo-boost disabled, there are 24 cores in the system operating at the frequency of 2.7 GHz.

Table 7.4 lists the parameters of JVM and Spark after tuning. For our experiments, we configure Spark in local mode in which driver and executor run inside a single JVM. We use HotSpot JDK version 7u71 configured in server mode (64 bit) and use Parallel Scavenge (PS) and Parallel Mark Sweep for young and old generations respectively as recommended in [25]. The heap size is chosen such that the memory consumed is within the system.

Measurement Tools and Techniques

We use linux iotop command to measure the total disk bandwidth. To find sustained maximum bandwidth, we compile the OpenMP version of STREAM [11] using Intel's ICC compiler. We use linux top com-

Table 7.3: Machine Details.

Component	**Details**	
Processor	Intel Xeon E5-2697 V2, Ivy Bridge micro-architecture	
	Cores	12 @ 2.7GHz
	Threads	1 per Core
	Sockets	2
	L1 Cache	32 KB for Instruction and 32 KB for Data per Core
	L2 Cache	256 KB per core
	L3 Cache (LLC)	30MB per Socket
Memory	2 x 32GB, 4 DDR3 channels, Max BW 60GB/s per Socket	
OS	Linux Kernel Version 2.6.32	
JVM	Oracle Hotspot JDK 7u71	
Spark	Version 1.5.0	

Table 7.4: 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		

mand in batch mode and monitor only java process of Spark to measure %usr (percentage CPU used by user process) and %io (percentage CPU waiting for I/O)

We use Intel Vtune Amplifier [4] to perform general micro-architecture exploration and to collect hardware performance counters. All measurement data are the average of three measure runs; Before each run, the file buffer cache is cleared to avoid variation in the execution time of benchmarks. Through concurrency analysis in Intel Vtune, we found that executor pool threads in Spark start taking CPU time after 10 seconds. Hence, hardware performance counter values are collected after the ramp-up period of 10 seconds. For batch processing workloads, the measurements are taken for the entire run of the applications and for stream processing workloads, the measurements are taken for 180 seconds as the sliding interval and duration of windows in streaming workloads considered are much less than 180 seconds.

We use top-down analysis method proposed by Yasin [185] to study the micro-architectural performance of the workloads. Earlier studies on profiling of big data workloads shows the efficacy of this method in identifying the micro-architectural bottlenecks [24, 97, 186]. The top-down method requires following metrics described in Table 7.5, whose definition are taken from Intel Vtune on-line help [4].

7.4 Evaluation

The case of ISP for Spark

Figure 7.1b shows the average amount of data read from and written to the disk per second for different Spark workloads. The data reveal that on average across the workloads, total disk bandwidth consumption is 56 MB/s. The SATA HDD installed in the machine under test can support up to 164.5 MB/s of 128 KB sequential reads and writes. However, the average response time for 4 KB reads and writes are 1803.41ms and 1305.66ms respectively [12]. This implies that Spark workloads do not saturate the bandwidth of SATA HDD, Earlier work [25] shows severe degradation in the performance of Spark workloads using large datasets due to significant wait time on I/O. Hence, it is the latency of I/O operations that are detrimental to the performance of Spark workloads.

Figure 7.1a shows average percentage CPU, a) used by Spark java process, b) in system mode c) waiting for I/O and d) in idle state during the execution of different Spark workloads. Even though the number of Spark worker threads are equal to the number of CPUs available in the system, during the execution of Spark SQL queries, only 8.97% CPUs

Table 7.5: 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.

are in user mode, 22.93% CPUs are waiting for I/O and 63.52% CPUs are in idle state. We see similar characteristics for Grep and Sort.

Grep, WordCount, Sort, NaiveBayes, Join, Aggregation, Cross Product, Difference and Orderby queries are all non iterative workloads, the data is read from and written to disk through out the execution period of workloads (see Figure 7.2a, 7.2b, 7.2c, 7.2d, 7.2f, 7.2g) and compute intensity varies from low to medium and the amount of data written to the disk also varies. For all these disk based workloads, we recommend in-storage processing. Since these workloads differ in the compute intensity, putting simple in-order cores would be less effective as compared to programmable logic, which can be programmed with workload specific hardware accelerators. Moreover, using hardware accelerators inside the NAND flash can free up the resources at the host CPU, which in turn can be used for other compute-intensive tasks.

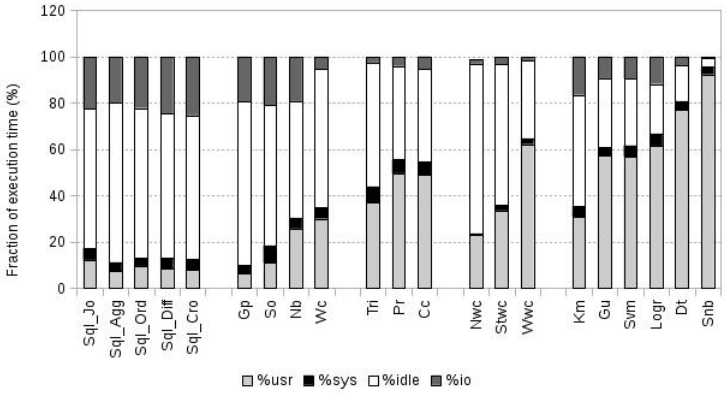
The case of PIM for Apache Spark

When Graph-X workloads are run, 45.15% CPUs are in the user mode, 3.98% CPUs wait for I/O and 44.63% CPUs are in the idle state. Pagerank, Connected Components and Triangle counting are iterative applications on graph data. All these workloads have a phase of heavy I/O with moderate CPU utilization followed by the phase of high CPU utilization and negligible I/O (see Figure 7.2l, 7.2m, 7.2n). These workloads are dominant by the second phase.

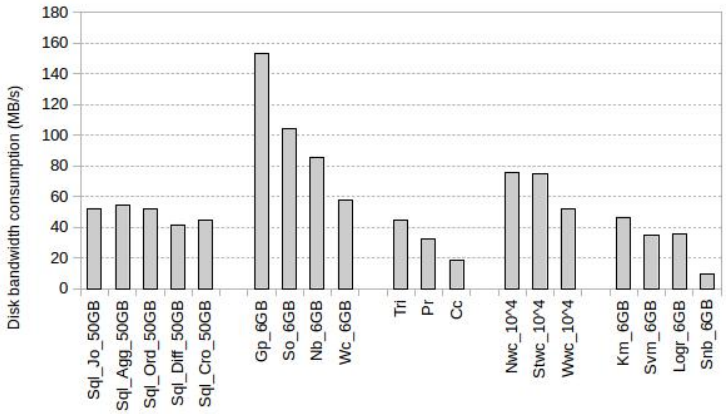
During the execution of stream processing workloads, 39.52% CPUs are in the user mode, 2.29% CPUs wait for I/O and 55.78% CPUs are in the idle state. The wait time on I/O for stream processing workloads is negligible (see Figure 7.2i, 7.2j, 7.2k) due to the streaming nature of the workloads but the CPU utilization also varies from low to high.

For Spark MLlib workloads, the percentage of CPUs in user mode, waiting for I/O and in idle state are 60.27%, 9.56% and 25.48%. SVM and Logistic Regression are phasic in terms of I/O (see Figure 7.2p, 7.2q). The training phase has significant I/O and also high CPU utilization, whereas the testing phase has negligible I/O and high CPU utilization because before the training starts, the input data is split into training and testing data and are cached in the memory.

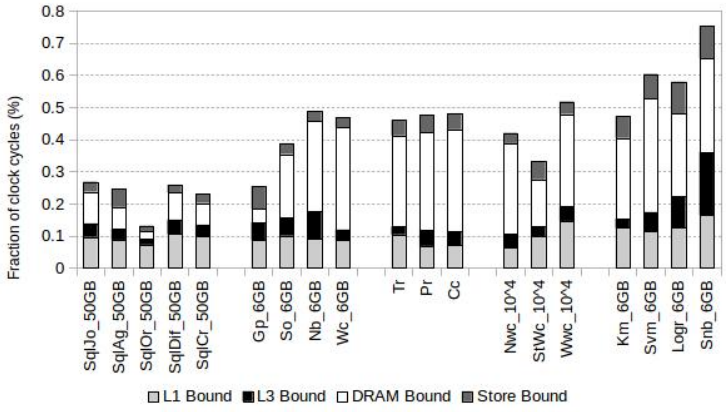
Since DRAM bound stalls are higher than L3 bound stalls and L1 bound stalls for most of the Graph-X, Spark Spark Streaming and Spark MLlib workloads (see Figure 7.1c), it means that CPUs are stalled waiting for the data to be fetched from the main memory and not by the caches (for detailed analysis see [24–26]). So, instead of moving the data back and forth through the cache hierarchy in between the iterations, it would be beneficial to use programmable logic based processing-in-memory. As a



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

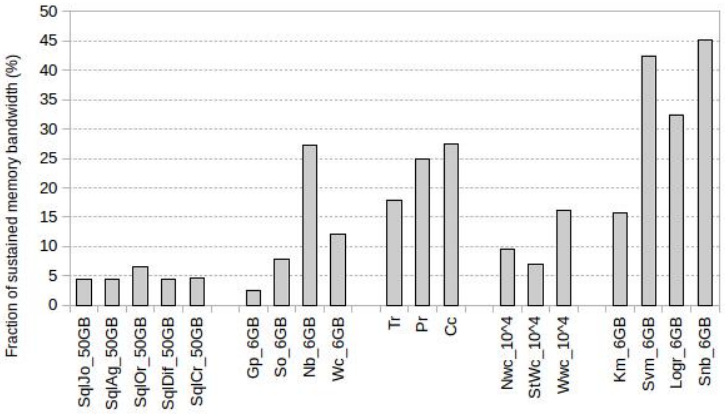


(b) Spark workloads do not saturate the disk bandwidth



(c) Spark workloads are DRAM bound

Figure 7.1: Characterization of Spark workloads from NDP perspective



(d) Spark workloads do not experience loaded latencies

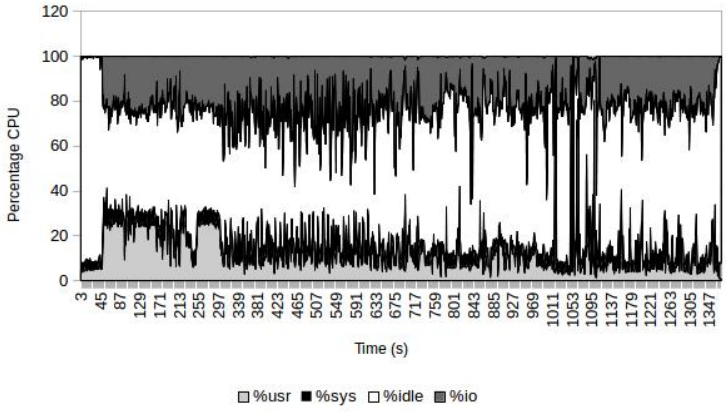
Figure 7.1: Characterization of Spark workloads from NDP perspective

result, application specific hardware accelerators are brought closer to the data, which will reduce the data movement and improve the performance of Spark workloads.

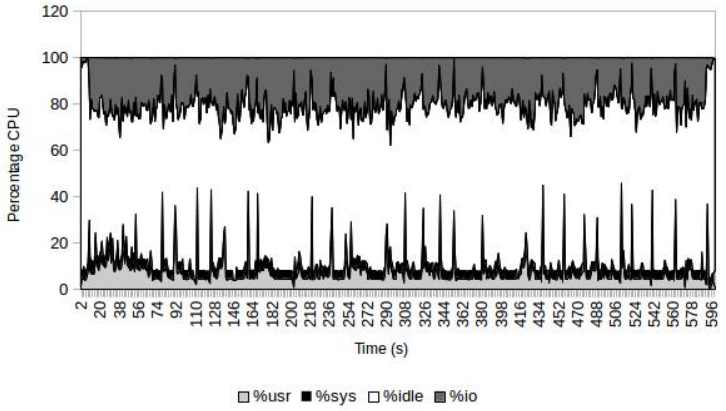
The case of 2D integrated PIM instead of 3D stacked PIM for Apache Spark

According to Jacob et al. [85], the bandwidth vs latency response curve for a system has three regions. For the first 40% of the sustained bandwidth, the latency response is nearly constant. The average memory latency equals idle latency in the system and the system performance is unbounded by the memory bandwidth in the constant region. In between 40% to 80% of the sustained bandwidth, the average memory latency increases almost linearly due to contention overhead by numerous memory requests. The performance degradation of the system starts in this linear region. Between 80% to 100% of the sustained bandwidth, the memory latency can increase exponentially over the idle latency of DRAM system and the applications performance is limited by available memory bandwidth in this exponential region.

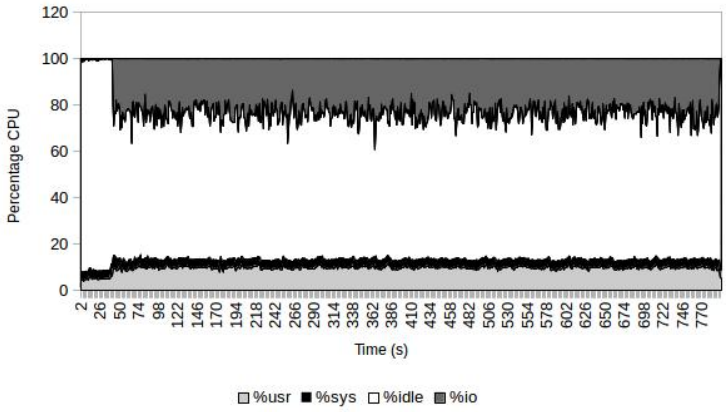
3D-Stacked PIM based on Hybrid Memory Cube (HMC) enables significantly more bandwidth between the memory banks and the compute units as compared to 2D integrated PIM, e.g. maximum theoretical bandwidth of 4 DDR3-1066 is 68.2 GB/s where as 4 HMC links provide 480 GB/s [142]. If the workload is operating in the exponential region on bandwidth vs latency curve of DDR3 based system, using HMC will move the workload to operate again in the constant region and average



(a) Sql_Jo

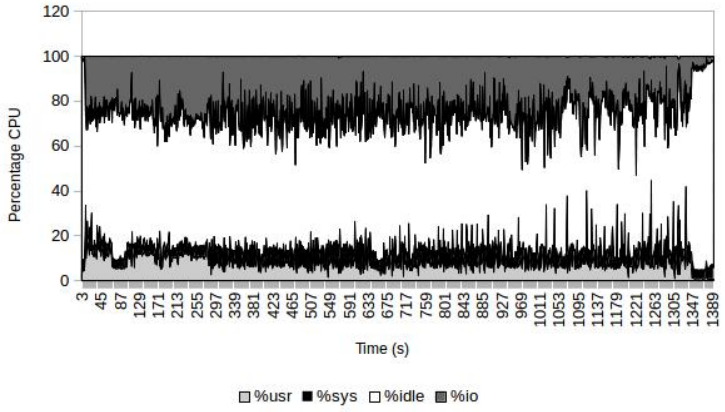


(b) Sql_Agg

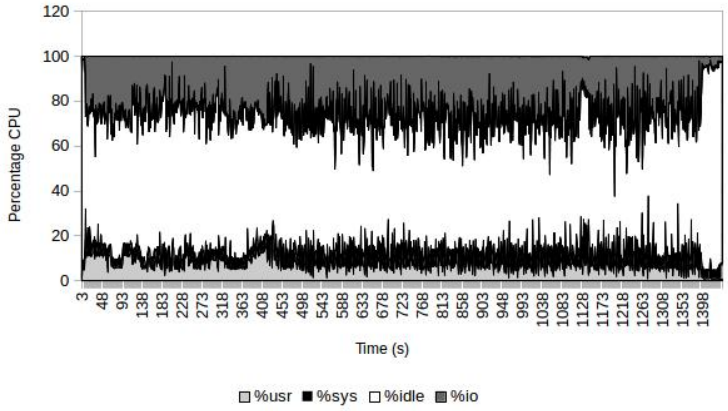


(c) Sql_Ord

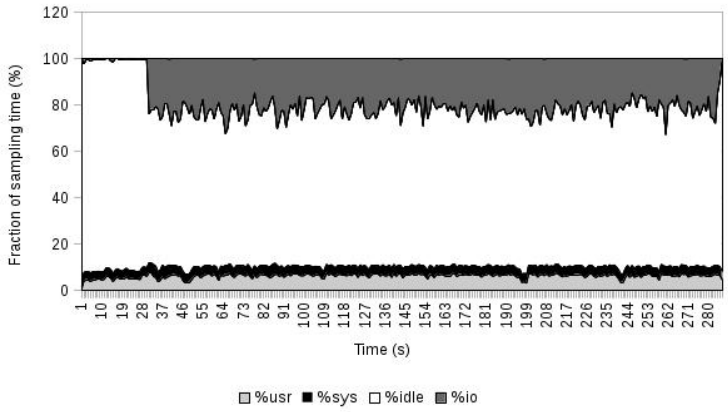
Figure 7.2: Execution time breakdown for Spark workloads



(d) Sql_Diff

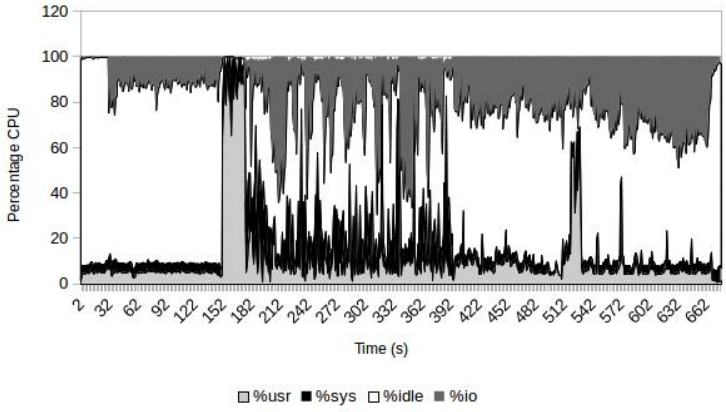


(e) Sql_Cro

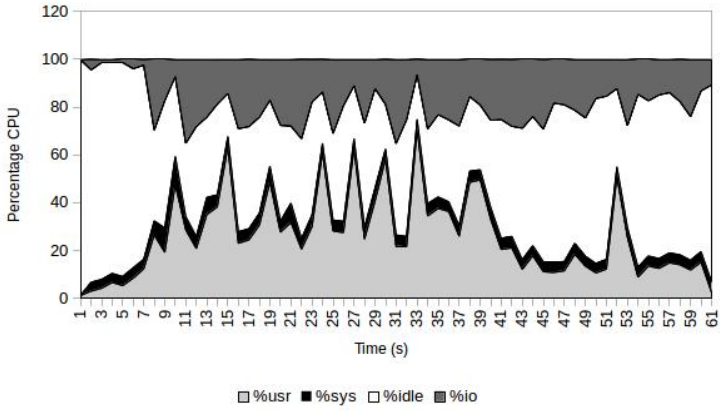


(f) Gp

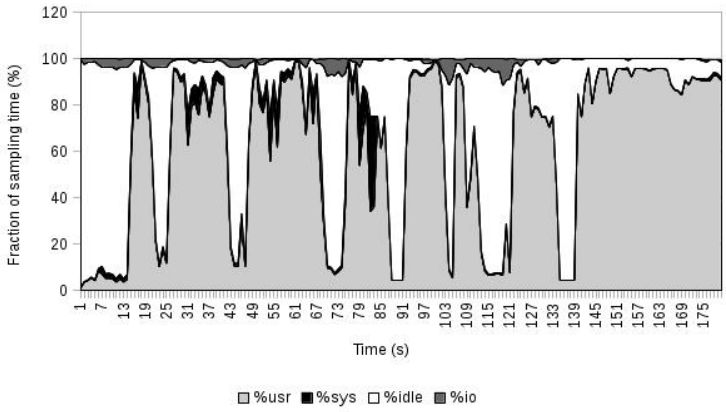
Figure 7.2: Execution time breakdown for Spark workloads



(g) So

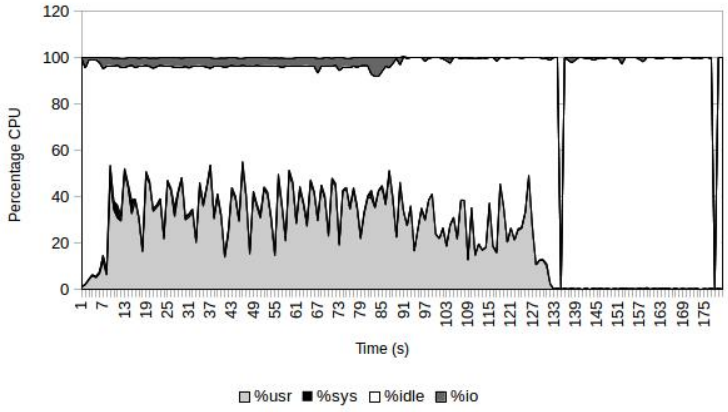


(h) Nb

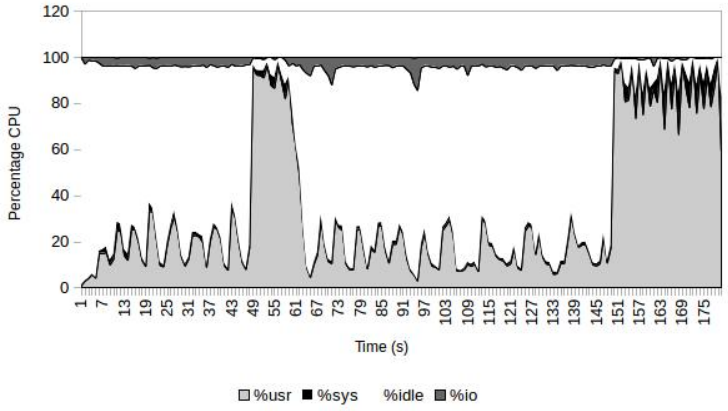


(i) WWc

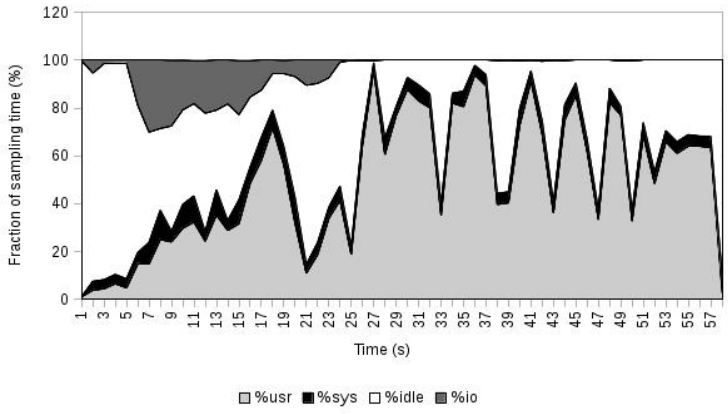
Figure 7.2: Execution time breakdown for Spark workloads



(j) NWc

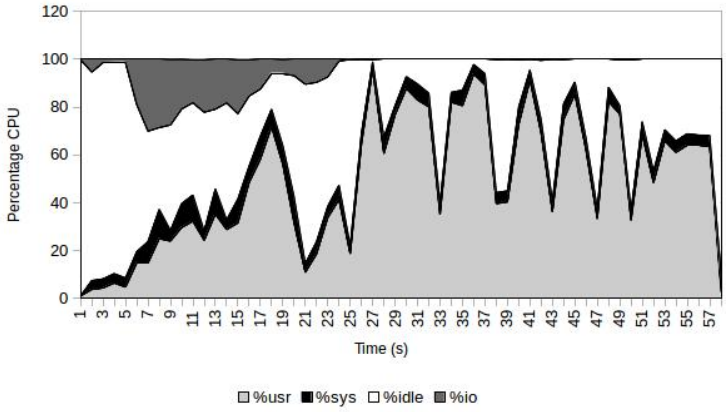


(k) Stwc

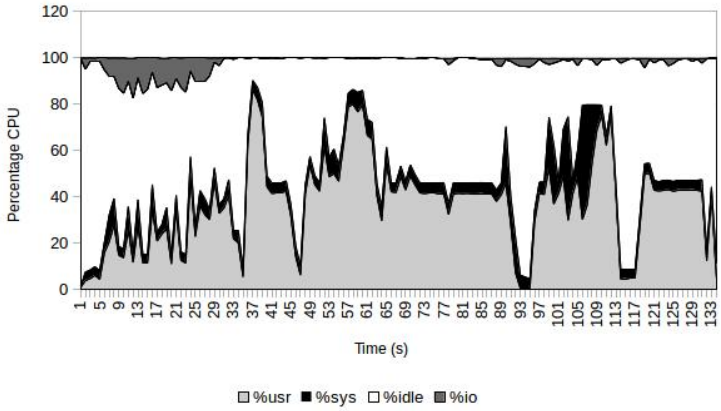


(l) Pr

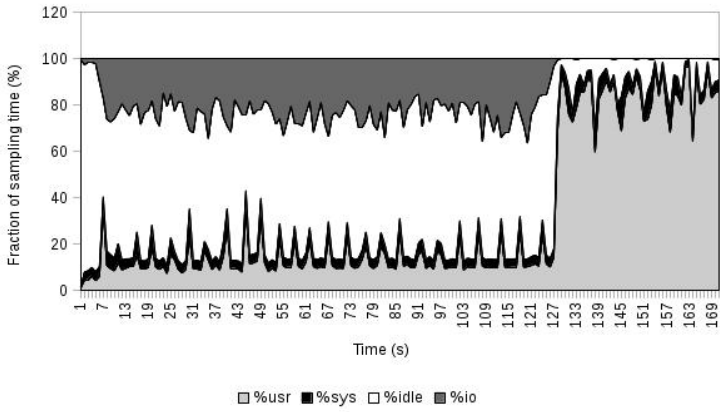
Figure 7.2: Execution time breakdown for Spark workloads



(m) Cc

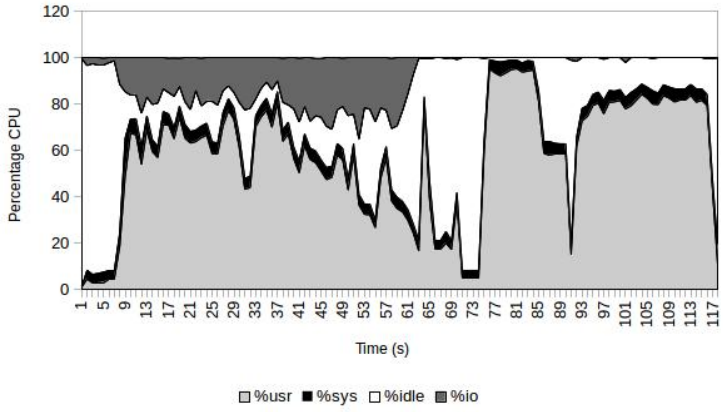


(n) Tr

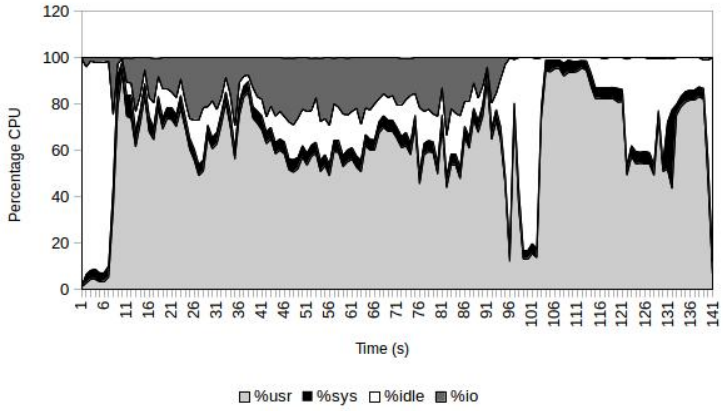


(o) Km

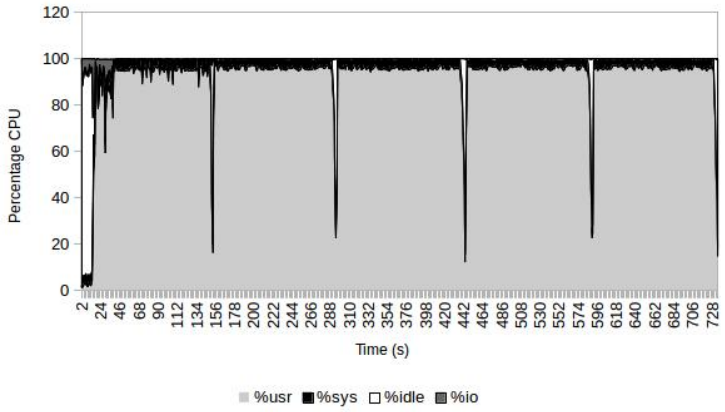
Figure 7.2: Execution time breakdown for Spark workloads



(p) Svm



(q) Logr



(r) Snb

Figure 7.2: Execution time breakdown for Spark workloads

memory latency equals idle latency of the system. On the other hand, if the workloads are not bounded by the memory bandwidth, NDP architecture based on 3D-stacked PIM would not be able to fully utilize the excessive bandwidth and goal of reducing the data movement can be achieved instead by 2D integrated PIM.

Figure 7.1d shows the average bandwidth consumption as a fraction of sustained maximum bandwidth. The data reveal Spark workloads consume less than 40% of sustained maximum bandwidth at 1866 MT/s data transfer rate and thus operate in the constant region. Awan et al. [27] study the bandwidth consumption of Spark workloads during the whole execution time of the workloads and show that even when the peak bandwidth utilization goes into the exponential region, it lasts only for a short period of time and thus, have a negligible impact on the performance. Thus we envision 2D integrated PIM instead of 3D stacked PIM for Apache Spark.

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

K-means is also an iterative algorithm. It has two distinct phases (see Figure 7.2o), heavy I/O phase followed by negligible I/O phase. The heavy IO phase has low cpu utilization. This phase implements kmeans|| initialization method to assign initial values to the clusters. This phase can be mapped to hardware accelerators in the programmable logic inside the storage, where as the main clustering algorithm can be mapped to 2D integrated PIM.

7.5 Conclusion

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

* Spark workloads, which are not iterative and have high ratio of % cpu waiting for I/O to % cpu in user mode like SQL queries, filter, word count and sort are ideal candidates for ISP.
* Spark workloads, which have low ratio of % cpu waiting for I/O to % cpu in user mode like stream processing and iterative graph processing workloads are bound by latency of frequent accesses to DRAM and are ideal candidates for 2D integrated PIM.
* Spark workloads, which are iterative and have moderate ratio of % cpu waiting for I/O to %cpu in user mode like K-means, have both I/O bound and memory bound phases and hence will benefit from the combination of 2D integrated PIM and ISP.

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

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

Chapter 8

The practicalities of Near Data Accelerators augmented Scale-up servers for In-Memory Data Analytics

8.1 Introduction

Traditionally, cluster computing frameworks like Apache Flink [35], Apache Spark [190], Apache Storm [164] etc, are being increasingly used to run real-time streaming analytics. These frameworks have been designed to use the cluster of commodity machines. Keeping in view the poor multi-core scalability of such frameworks [27], we hypothesize that coherently attached processor interface (CAPI) [161] based scale-up machines can deliver enhanced performance for in-memory big data analytics.

Our contributions are

* We propose system design for FPGA acceleration of big data processing frameworks on CAPI based scale-up servers.
* We estimate 4x speedup in the scale-up performance of Apache Spark on CAPI based scale-up machines using roof-line model.

8.2 System Design

This section describes the main challenges involved, the decisions we made, and the scheduling scheme we developed.

Challenges

The work addresses following challenges.

* How to efficiently utilize both CPU and FPGA for a single application?
  We propose to run the map and reduce tasks on both CPU and FPGA and balance the load dynamically between CPU and FPGA.
* How to attain peak performance on the CPU side?
  We propose multi-threading and vectorization.
* How to attain peak CAPI bandwidth consumption?
  We propose to overlap read/write requests on the FPGA.
* How to attain peak performance on the FPGA side?
  We propose to employ map side partial reductions to fit the intermediate data inside the FPGA and if it does not fit inside the FPGA use device external memory. Also, employ double buffering technique between the AFU and device memory and multiple FIFO buffers between PSL and AFU side to hide the PCIe overhead.
* How to make accelerators easily programmable?
  We propose to use pragmas in SDSoC to guide Vivado HLS to generate the map and reduce accelerators.
* How to hide JVM to FPGA communication?
  We propose to offload algorithm instead of specific kernels only.

High Level Design

Figure 8.1 shows our high level solution. The naive approach of offloading the hotspot functions identified by profiler like Intel Vtune does not work here as our profiling experience with Apache Spark and Apache Flink reveals, there is no single hot-spot function that contributes to more than 50% of the total execution time, and instead there are different hotspot functions, each contributing up to 10-15 % of the total execution time. Other ways of accelerating big data processing frameworks like Apache Spark are offloading the tasks or offloading the algorithm. By Comparing previous studies [67, 78], we find that offloading the whole algorithm incurs less JVM-FPGA communication overhead than offloading the individual tasks. Thus, we choose offloading the algorithm outside the Spark-framework, even though the algorithm is still written following the MapReduce programming model. The mapping decisions between CPU and FPGA are taken outside the JVM.

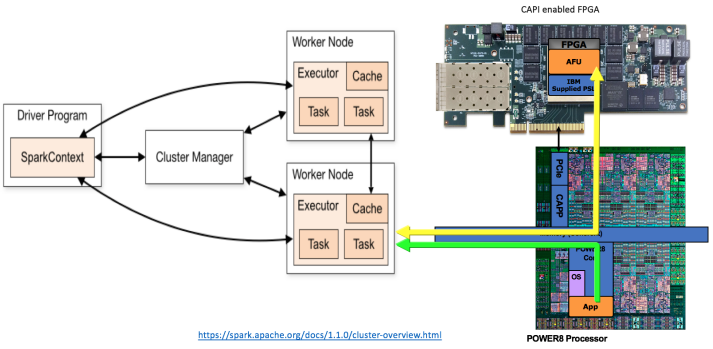


Figure 8.1: Our High Level Solution

Compared to existing literature, our work contrasts as follows

* We focus on hiding the data communication overhead by offloading the whole algorithm (reducing the no of accelerator function calls) and data-reuse on the FPGA side (amortizing the data transfer overhead).
* We use the integration approach proposed by [58] and apply the diverse set of optimization both on CPU side and FPGA side.
* We also offload the entire computation instead of key computation kernels from Spark to our optimized hardware/software co-designed framework similar to [20]. Contrasting their approach, in our work, data is read from the Java heap for optimized C++ processing on the CPUs and hardware acceleration of the FPGAs and final results are copied back into Spark using memory mapped byte buffers.

* We exploit CAPI to further reduce the communication cost.
* We use co-processing on the CPUs as well as FPGA to finish all the map tasks as quickly as possible.

CAPI Specific Optimization

CAPI allows to couple the hardware and software threads in a very fine-grained manner. Shared virtual memory is the key innovation of the OpenCL standard and allow host and device platforms to operate on shared data-structures using the same virtual address space. We pass the pointers to the CAPI accelerators to read the data directly from the Java Heap, which removes the overhead of pinned buffers on host memory. Due to CAPI, the accelerators have access to the whole system memory of TB scale and thus accelerators can work on big data sets.

HDL vs. HLL

The main obstacle for the adoption of FPGAs in big data analytics frameworks is the high programming complexity of hardware description languages (HDL). In last years, there are several efforts from the main FPGA and system vendors to allow users to program FPGA using high-level synthesis (HLS), like OpenCL or specific-domain languages like OpenSPL. Although HDLs can provide the higher speedup, the low programming complexity of HLL makes them very attractive in the big data community [67, 78, 129, 149]. We use SDSoC to generate the hardware accelerators. We exploit following pragmas whose description is taken from Xilinx SDSoC user guide [180].

Task Pipelining

If there are multiple calls in the application, you can structure the application such that you can pipeline these calls and overlap the setup and data transfer with the accelerator computation. To enable the pipelining, we need to provide extra local memory to store the second set of arguments while the accelerator is computing with the first set of arguments.The SDSoC generate these memories, called multi-buffers, under the guidance of the user. Specifying the task level, pipelining requires rewriting the calling code using the pragmas async (id) and wait (id).

Function Inlining

It replaces a function call by substituting a copy of the function body after resolving the actual and formal arguments. After that, the inlined function is dissolved and no longer appears as a separate level of the

hierarchy. Function inlining allows operations within the inlined function being optimized more effectively with surrounding operations, thus improves the overall latency or the initiation interval for a loop. SDSoC provides the pragma HLS inline.

Loop Pipelining

In sequential languages, the operations in a loop are executed sequentially and the next iteration of the loop can only begin when the last operation in the current loop iteration is complete. Loop pipelining allows the operations in a loop to be implemented in a concurrent manner. An important term for loop pipelining is called Initiation Interval, which is the number of the clock cycles between the start times of consecutive loop iterations. To pipeline a loop, use the pragma pipeline.

Loop Unrolling

It is another technique to exploit parallelism between loop iterations. It creates multiple copies of the loop body and adjusts the loop iteration counter accordingly. It generates more operations in each loop iteration, thus Vivado HLS can exploit more parallelism among these operations. If the factor N is less than the total number of loop iterations, it is called a partial unroll and if the factor N is the same number of loop iterations, it is called a full unroll.

Performance Limiting Factors and Remedies

Both, loop pipelining and loop unrolling exploit parallelism between iterations. However, parallelism between loop iterations is limited by two main factors: one is the data dependencies between loop iterations, the other is the number of available hardware resources. A data dependence from an operation in one iteration to another operation in a subsequent iteration is called loop-carried dependence. It implies that the operation in the subsequent iteration cannot start the operation in the current iteration has finished computing the data input for the operation in the subsequent iteration. Loop-carried dependencies fundamentally limit the initiation interval that can be achieved using loop pipelining and the parallelism that can be exploited using loop unrolling.

Another performance limiting factor for loop pipelining and loop unrolling is the number of available hardware resources, e.g. if the loop is pipelined with an initiation interval and if the memory has only a single port, then the two read operations cannot be executed simultaneously and must be executed in two cycles. The same can happen with other hardware resources, e,g. if the op_compute is implemented with a DSP

core which cannot accept new inputs every cycle, and there is one such DSP core. Then op_compute cannot be issued to the DSP core each cycle, and an initiation interval of one is not possible.

Increasing Local Memory Bandwidth: If the loop pipelining and loop unrolling are limited by insufficient memory ports, local memory bandwidth needs to be increased.

Array Partitioning

Arrays can be partitioned into smaller arrays. The physical implementation of memories have only a limited number of read ports and write ports, which can limit the throughput of a load/store intensive algorithm. The memory bandwidth can sometimes be improved by splitting the original array (implemented as a single memory source) into multiple smaller arrays (implemented as multiple memories), effectively increasing the number of load/store ports.

Three types of array partitioning are block, cyclic a complete. Block-split the original array into equally sized blocks of consecutive elements of the original. Cyclic-split the original array into equally sized blocks interleaving the elements of the original array. Complete-split the array into individual elements. This corresponds to implementing an array as a collection of registers rather than as a memory.

Array Reshaping

Arrays can be reshaped to increase the memory bandwidth. Reshaping takes different elements from a dimension in the original array and combines them into a single wider element. Array reshaping is similar to array partitioning, but instead of partitioning into multiple arrays, it widens array elements.

Data Flow Pipelining

The previously discussed optimization techniques are all "fine grain" parallelizing optimizations at the level of operators, such as multiplier, adder, and memory load/store operations. These techniques optimize the parallelism between these operators. Data flow pipelining, on the other hand, exploits the "coarse grain" parallelism at the level of functions and loops. Data flow pipelining can increase the concurrency between functions and loops.

Function Data Flow Pipelining

The default behavior for a series of function calls in Vivado HLS is to complete a function before starting the next function. Vivado HLS

implements function data flow pipelining by inserting "channels" between functions. These functions are implemented as either ping-pong buffers or FIFOs, depending on the access patterns of the producer and the consumer of the data. If a function parameter (producer or consumer) is an array, the corresponding channel is implemented as a multi-buffer using standard memory accesses (with associated address and control signals). For a scalar, pointer and reference parameters, as well as the function return, the channel is implemented as a FIFO, which uses fewer hardware resources (no address generation) but required that the data is accessed sequentially.

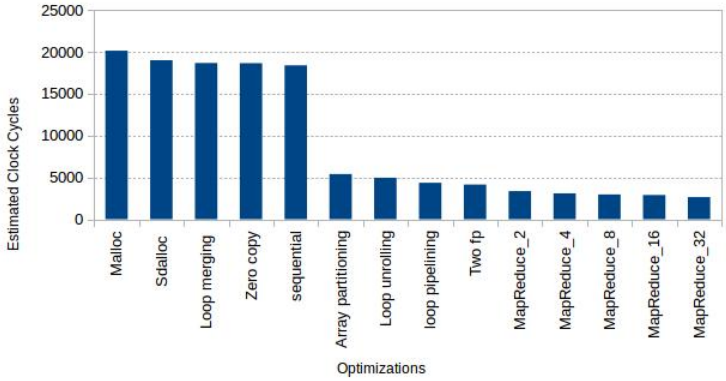
Loop Data Flow Pipelining

Data flow pipelining can also be applied to loops in a similar manner as it can be applied to functions. It enables the sequence of loops, normally executed sequentially, to execute concurrently. Data flow pipelining should be applied to a function, loop or region, which contains either all functions or all loops: do not apply on a scope containing a mixture of loops and functions. Vivado HLS automatically inserts channels between the loops to ensure data can flow asynchronously from one loop to the next.

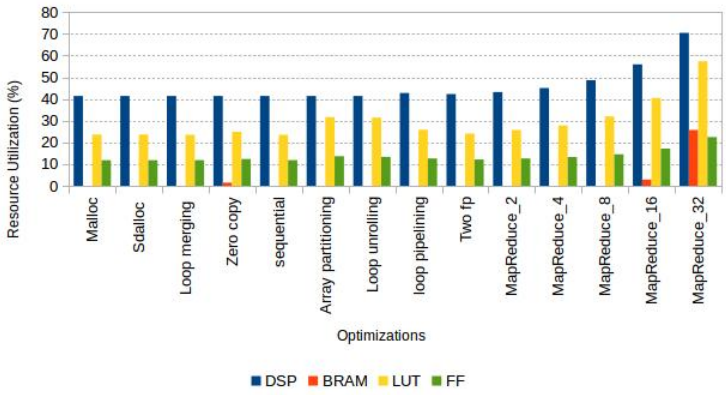
Opportunities and Limitations of High-Level Synthesis for Big Data Workloads

We apply the optimizations described in the previous section to Stream processing algorithms, i) Bloom Filters [109], ii) Count-Min Sketch [48] and iii) HyperLogLog [65]. We discuss the implementation of Hyperloglog in detail and walk through different optimization to show how these tools enable quick design space exploration (see Fig 8.2). For Count-Min Sketch and Bloom Filters, we only show the numbers before and after applying the series of optimizations (see Fig 8.3).

Figure 8.2a shows that impact of different optimizations applied to the starting code for HyperLogLog written in C/C++. The optimized version after introducing a series of pragmas is 4.4x faster than the baseline implementation. We then rewrite the optimized version using map-reduce programming model which improves further the estimated clock cycles by 1.7x. However, instantiating multiple number of mappers and reducers strain the internal FPGA resources especially block rams and dsp units as shown in Fig 8.2b. The resource utilization of dsp units increases 42% to 70% when 32 mappers are instantiated instead of 2.



(a) Estimated hardware clock cycles for different optimization

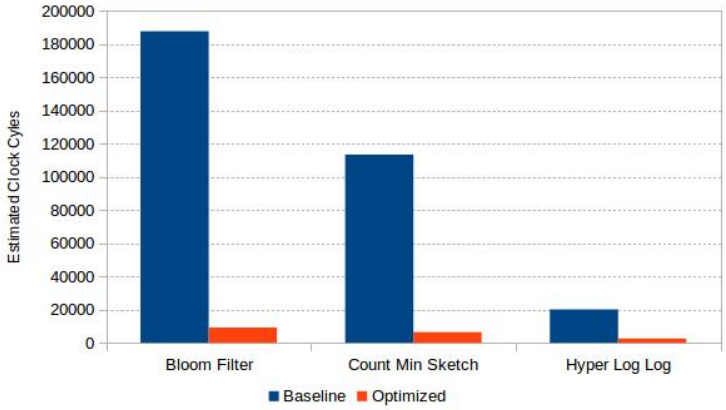


(b) Resource utilization for different optimization

Figure 8.2: Design Space Exploration of HyperLogLog Using SDSoC

Programmable Accelerators for iterative map-reduce programming model

In one iteration, Mapper accelerators read (K, V) pairs from system memory at slow speed, process them in parallel and generate the output (K, V) pairs. The amount of intermediate data depends on the number of mappers. This number, which can be generated, depends on the FPGA resources consumed by a single mapper. The smaller the resources consumed by one single map accelerator, the larger the number of map accelerators can be instantiated. The resources consumed by a single map accelerator depends on the compute intensity of the map function and the inherent parallelism available within the map function.



(a) Estimated hardware clock cycles

Figure 8.3: Comparison of Baseline and Optimized implementation of Stream Processing Applications

By using fully parallel design, a single map accelerator consumes a lot of FPGA resources and thus the number of parallel mappers that can be instantiated is reduced accordingly, as well as the amount of the intermediate data. The intermediate data can be stored inside the FPGA and the reducers operate on this data to generate the final results which are also stored inside the FPGA.

Current designs in the literature fit to one of the following assumptions.

* Assumption 01: Training data, model, and intermediate data fit in the FPGA internal memory and is kept across the iterations.
* Assumption 02: Model and intermediate data fit in the FPGA internal memory, but training data does not fit inside the FPGA and is kept on FPGA external DDR3 memory.
* Assumption 03: Training data does not fit on the FPGA external memory but model fits inside the FPGA.
* Assumption 04: Training data does not fit on the FPGA external memory but fits on the System memory and model does not fit inside the FPGA memory.

Our focus is on designing accelerators that fit Assumption 03 and 04.

Scheduling scheme for Assumption 03 (Big Data and small model): Send the model once over the CAPI and store inside the FPGA block rams. Stream the training data from the CAPI into FPGA, update the predictive model using map-reduce accelerators every iteration and once the convergence is reached, the model is output to the system memory.

Scheduling scheme for Assumption 04 (Big Data and big model): Stream the model once from system memory to the FPGA device external memory and then, at the start of each iteration, stream the model in from FPGA device external memory and write the model out at the end of each iteration, whereas training data is still streamed in over CAPI. Another design option is that training data is streamed in from CAPI to map accelerator, the output of which is written to FPGA device external memory, which is further streamed in by the reducers.

Device Service Layer: Vivado can create the IP using Xilinx Memory Interface Generator [179] for DDR3 memory.

Word-Lengths: Our design supports 32-bit floating point numbers. The yellow lines are the data-path and blue lines are the control signals (see Fig 8.4)

General Sequencer: It is a finite state machine with variable no of states. The number of states can be varied to adjust to different configurations of mappers and reducers and also to different scheduling schemes. Awan [23] uses a similar idea to generate accelerators for norm optimal iterative learning control algorithm.

On-Chip Distributed Memory Architecture: It comprises of double buffers to hide the access latencies of PCIe, FIFO buffers at the input and output of mappers and reducers. The length of buffers is also configurable to adjust to the amount of block-ram available in the FPGA card.

Parametric Design: Our design has different parameters to configure for each workload. A constraint solver will be required to generate an optimized number of parameters for the number of mappers and reducers.

Advantages of our design

Our solution has following advantages,

* Template-based design to support generality.
* No of mappers and reducers can be instantiated based on the FPGA card.
* General Sequencer is a Finite State Machine whose states can be varied to meet the diverse set of workloads
* Mappers and Reducers can be programmed in C/C++ and can be synthesized using Vivado High-Level Synthesis.
* Support hardware acceleration of diverse set of workloads

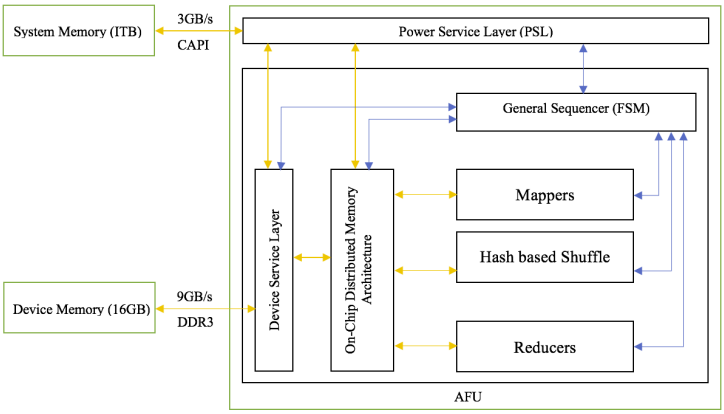


Figure 8.4: System Architecture on FPGA-side

Table 8.1: Spark MLlib Workloads

Spark Workloads	Time Complexity per iteration
K-Means	O(k*n*d)
Linear Regression	O(n*d2 + d3)
Gradient Descent	O(n*d)
SVM using SMO (libsvm)	O(d*n3)
Decision Tree Training	O(n*d*log(n))
Least Squares using SVD	O(n* d2)
Ridge Regression	O(n*d2)
Least angle regression	O(n*d2)
Alternating Least Squares	O(k2 + n*k3)
Cholesky Factorization	O(n3)
Multi Layer Perceptron	O(n*m*d)
Stochastic Gradient Descent	O(n*d + k*n)

8.3 Evaluation Technique and Results

We develop roof-line model for the target machine whose specifications are given in Table 8.2. The peak performance (Gflops) for the CPU is calculated by multiplying cpu-speed, number of sockets, number of core per socket and instruction retirement per socket. The peak bandwidth for CPU is obtained by multiplying the number of centaur chips with the summation of read and write bandwidth per centaur chip. In order to calculate peak GFlops for the FPGA card, we adopt approach described by Intel [123]. According to this approach, the best choice is to use

Table 8.2: Machine Details

CPU Specifications	**Power 8 based Server**
CPU Speed (GHz)	3.325
Sockets	4
Cores per socket	6
Threads per core	8
Instruction retirement per core	8
Centaur chips	24
Read bandwidth per Centaur (GB/s)	19.2
Write bandwidth per Centaur (GB/s)	9.6
FPGA Specifications	**ADM-PCIE-KU3**
CAPI Complaint	Yes
Host I/F	PCIe Gen 3x8
Target Device	Xilinx® Kintex® Ultrascale™ : XCKU060 - FFVA1156
On-board memory (GB)	16
BRAM (Mb)	38
Max Distributed RAM (Mb)	9.1
Block RAM blocks	1,080
DSP slices	2,760
System Logic Cells	725,550
CLB Flip-Flops	663,360
CLB LUTs	331,680

the add/subtract function to maximize floating-point rating. The best strategy is to build as many adders as possible until the DSP48E slices are exhausted, and build the remaining adders with pure logic. The DSP resourced adders require 2 DSP slices and 354 LUT-FF pairs (or LCs) and a single instantiation can operate at 519 MHz. The logic-based adder uses 578 LCs, and a single instantiation can operate at 616 MHz. Assuming 100% logic and 100% DSP slices are used (this requires enough routing to be available to utilize all of the logic), the Kintex Ultra-scale XCKU060 FPGA (see Table 8.2) can deliver 968 GFLOPS of single-precision floating-point performance. The sustained bandwidth for DMAs to the system memory over CAPI interface is 3GB/s and DMAs to the device memory attains 9GB/s of sustained bandwidth [47]. Fig 8.6 shows the combined roof-line model.

We combine modeling and partial emulation to estimate the bounds on the speedup achieved by our solution. The based-line is obtained by reproducing the scalability experiments from Chapter 03. Fig 8.5 shows that speed up for K-Means application saturates at 9 when 24

threads are configured in the executor pool and each worker thread is bound to the separate core. In other words, the peak Gflops attained by Spark based K-Means on a 24-core machine is equivalent to the peak Gflops of a machine with 9 cores. Thus in terms of peak Gflops, the baseline performance for Spark based K-Means is 30 Gflops. Based on the time and space complexity of K-Means algorithm (see table 8.1), the arithmetic intensity is estimated to be 32. By mapping, both arithmetic intensity of K-Means algorithm and peak attainable Gflops for Spark based K-Means, on the roofline model of CPU + CAPI based FPGA machine (see Fig 8.6) we see that offloading the whole algorithm to the FPGA with coherent accesses to TB scale system memory can deliver 120 Gflops. This implies the potential of 4x speedup. If the accelerators use device memory, which has 3x higher bandwidth than CAPI interface, the upper limit for speedup is 288/30 = 9.6x. Enabling vectorization on CPU side can deliver upto 8x speedup. The arithmetic intensities of other machine learning workloads in Table 8.1 are much higher than that of K-means and thus the potential of performance improvement for those workloads is even better.

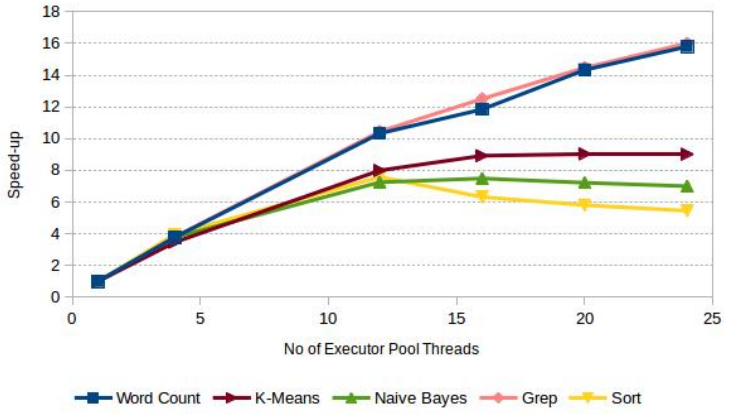


Figure 8.5: Scalability Study of Spark applications

8.4 Conclusion

Using the roof-line model of coherently attached FPGA based scale-up server, we estimate the speedup achievable by in-memory big data analytics on coherently attached FPGA based scale-up servers.

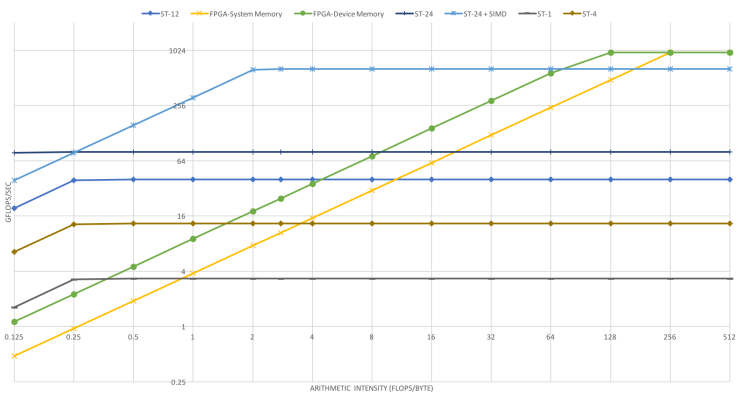


Figure 8.6: Roofline Model of Power 8 + CAPI enabled FPGA Server

Chapter 9

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 MT/s.

Secondly, we envision processors with 6 hyperthreaded 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 Moore's law, increasing DRAM capacity will be a challenge. NVRAM, on the other hand, shows a promising trend in terms of capacity scaling. Since Spark based workloads are I/O intensive when the input datasets don't fit in memory and are bound on latency when they do fit in-memory, In-Memory processing, and In-storage processing can be combined to form 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 9.1 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 organize the data and a set of reduce tasks to aggregate it. Internally results are kept in

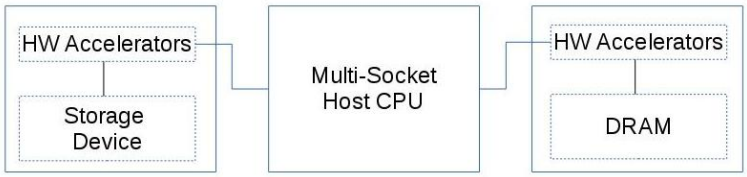


Figure 9.1: NDC Supported Single Node in Scale-in Clusters for in-Memory Data Analytics with Spark

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 worthwhile to investigate 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 database operations like Aggregations, Projections, Joins, Sorting, Indexing, and Compression can be researched further. Q100 [176] 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. Other future work involved addressing following research questions

- How to design the best hybrid CPU + FPGA ML workloads?
- How to attain peak performance on CPU side?
- How to attain peak performance on FPGA side?
- How to balance the load between CPU and FPGA?
- How hide communication between JVM and FPGA?
- How to attain peak CAPI bandwidth consumption?
- How to design the clever ML workload accelerators using HLS tools?

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

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

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

[6] Message Processing Interface. URL http://mpi-forum.org/.

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

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

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

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

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

[12] Toshiba SATA HDD Enterprise, Performance Review. URL http://www.storagereview.com/toshiba_sata_hdd_enterprise_35_review_mg03acax00.

[13] 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.

[14] Tarek S Abdelrahman. Accelerating k-means clustering on a tightly-coupled processor-fpga heterogeneous system. In *Application-specific Systems, Architectures and Processors (ASAP), 2016 IEEE 27th International Conference on*, pages 176–181. IEEE, 2016.

[15] 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.

[16] 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.

[17] 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.

[18] AMD. AMD's Aparapi. URL `https://github.com/aparapi/aparapi`.

[19] 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.

[20] Michael Anderson, Shaden Smith, Narayanan Sundaram, Mihai Capota, Zheguang Zhao, Subramanya Dulloor, Nadathur Satish, and Theodore L Willke. Bridging the gap between hpc and big data frameworks. *Proceedings of the VLDB Endowment*, 10(8), 2017.

[21] 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.

[22] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, *et al.* Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[23] Ahsan Javed Awan. *FPGA Based Implementation of Norm Optimal Iterative Learning Control*. Master thesis, University of Southampton, UK, June. 2012.

[24] 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.

[25] 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.

[26] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. Micro-architectural characterization of apache spark on batch and stream processing workloads. In *Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom), 2016 IEEE International Conferences on*, pages 59–66. IEEE, 2016.

[27] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. Node architecture implications for in-memory data analytics on scale-in clusters. In *Big Data Computing Applications and Technologies (BDCAT), 2016 IEEE/ACM 3rd International Conference on*, pages 237–246. IEEE, 2016.

[28] Ahsan Javed Awan, Moriyoshi Ohara, Eduard Ayguadé, Kazuaki Ishizaki, Mats Brorsson, and Vladimir Vlassov. Identifying the potential of near data processing for apache spark. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2017, Alexandria, VA, USA, October 02 - 05, 2017*, pages 60–67, 2017.

[29] 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.

[30] 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.

[31] 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.

[32] Dhruba Borthakur *et al.* Hdfs architecture guide. *Hadoop Apache Project*, 53, 2008.

[33] breeze. Breeze. URL `https://github.com/scalanlp/breeze`.

[34] 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.

[35] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[36] 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.

[37] 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.

[38] 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.

[39] Ren Chen and Viktor K Prasanna. Accelerating equi-join on a cpu-fpga heterogeneous platform. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*, pages 212–219. IEEE, 2016.

[40] 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.

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

[42] Benjamin Y Cho, Won Seob Jeong, Doohwan Oh, and Won Woo Ro. Xsd: Accelerating mapreduce by harnessing the gpu inside an ssd. In *Proceedings of the 1st Workshop on Near-Data Processing*, 2013.

[43] Hyeokjun Choe, Seil Lee, Hyunha Nam, Seongsik Park, Seijoon Kim, Eui-Young Chung, and Sungroh Yoon. Near-data processing for differentiable machine learning models. 2017.

[44] 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.

[45] 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.

[46] 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.

[47] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. A quantitative analysis on microarchitectures of modern cpu-fpga platforms. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2016.

[48] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[49] 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.

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

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

[52] 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.

[53] 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.

[54] 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.

[55] 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.

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

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

[58] Celestine Dünner, Thomas Parnell, Kubilay Atasu, Manolis Sifalakis, and Haralampos Pozidis. High-performance distributed machine learning using apache spark. *arXiv preprint arXiv:1612.01437*, 2016.

[59] 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.

[60] 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.

[61] 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.

[62] Babak Falsafi and Thomas F Wenisch. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture*, 9(1):1–67, 2014.

[63] 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.

[64] 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.

[65] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[66] 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.

[67] Ehsan Ghasemi and Paul Chow. Accelerating apache spark big data analysis with fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*, pages 94–94. IEEE, 2016.

[68] Heiner Giefers, Raphael Polig, and Christoph Hagleitner. Accelerating arithmetic kernels with coherent attached fpga coprocessors. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1072–1077. EDA Consortium, 2015.

[69] 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.

[70] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, 2014.

[71] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, *et al.* Biscuit: A framework for near-data processing of big data workloads. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 153–165. IEEE, 2016.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] Sam MH Ho, Maolin Wang, Ho-Cheung Ng, and Hayden Kwok-Hay So. Towards fpga-assisted spark: An svm training acceleration case study. In *ReConFigurable Computing and FPGAs (ReConFig), 2016 International Conference on*, pages 1–6. IEEE, 2016.

[77] 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.

[78] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. Programming and runtime support to blaze fpga accelerator deployment at datacenter scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 456–469, 2016.

[79] 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.

[80] Skand Hurkat, Jungwook Choi, Eriko Nurvitadhi, José F Martínez, and Rob A Rutenbar. Fast hierarchical implementation of sequential tree-reweighted belief propagation for probabilistic inference. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–8. IEEE, 2015.

[81] Intel Corporation. Intel® FPGAs. URL `https://www.altera.com/products/fpga/overview.html`.

[82] Intel Corporation. Xeon+FPGA Platform for the Data Center. URL `https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf`.

[83] 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.

[84] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: intelligent distributed storage. *Proceedings of the VLDB Endowment*, 10(11):1202–1213, 2017.

[85] 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.

[86] 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.

[87] Zhen Jia, Wanling Gao, Yingjie Shi, Sally A McKee, Jianfeng Zhan, Lei Wang, and Lixin Zhang. Understanding processors design decisions for data analytics in homogeneous data centers. *IEEE Transactions on Big Data*, 2017.

[88] 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.

[89] 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.

[90] 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.

[91] Yong-Yeon Jo, Sang-Wook Kim, Moonjun Chung, and Hyunok Oh. Data mining in intelligent ssd: Simulation-based evaluation. In *Big Data and Smart Computing (BigComp), 2016 International Conference on*, pages 123–128. IEEE, 2016.

[92] S. W. Jun, C. Chung, and Arvind. Large-scale high-dimensional nearest neighbor search using flash memory with in-store processing. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, Dec 2015.

[93] S. W. Jun, H. T. Nguyen, V. Gadepally, and Arvind. In-storage embedded accelerator for sparse pattern processing. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept 2016.

[94] Christoforos Kachris, Dionysios Diamantopoulos, Georgios Ch Sirakoulis, and Dimitrios Soudris. An fpga-based integrated mapreduce accelerator platform. *Journal of Signal Processing Systems*, pages 1–13, 2016.

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

[96] Christoforos Kachris and Dimitrios Soudris. A survey on reconfigurable accelerators for cloud computing. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–10. IEEE, 2016.

[97] 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.

[98] Yangwook Kang, Yang-suk Kee, Ethan L Miller, and Chanik Park. Enabling cost-effective data processing with smart ssd. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–12. IEEE, 2013.

[99] 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.

[100] 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.

[101] 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.

[102] Gunjae Koo, Kiran Kumar Matam, HV Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, Murali Annavaram, *et al.* Summarizer: trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 219–231. ACM, 2017.

[103] Jay Kreps, Neha Narkhede, Jun Rao, *et al.* Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[104] 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.

[105] 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.

[106] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H Peter Hofstee, Gi-Joon Nam, Mark R Nutter, and Damir Jamsek. Extrav: boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment*, 10(12):1706–1717, 2017.

[107] Joo Hwan Lee, Jaewoong Sim, and Hyesoon Kim. Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 241–252. IEEE, 2015.

[108] Young-Sik Lee, Luis Cavazos Quero, Youngjae Lee, Jin-Soo Kim, and Seungryoul Maeng. Accelerating external sorting via on-the-fly data merge in active ssds. In *HotStorage*, 2014.

[109] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

[110] 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.

[111] 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.

[112] Zhehao Li, Jifang Jin, and Lingli Wang. High-performance k-means implementation based on a coarse-grained map-reduce architecture. *arXiv preprint arXiv:1610.05601*, 2016.

[113] 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.

[114] 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.

[115] 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.

[116] 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.

[117] 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.

[118] 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.

[119] 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.

[120] 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.

[121] 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.

[122] 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.

[123] Michael Parker (Intel). Understanding Peak Floating-Point Performance Claims. `https://www.altera.com/en_US/pdfs/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf`.

[124] 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.

[125] 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.

[126] Raghid Morcel, Mazen Ezzeddine, and Haitham Akkary. Fpga-based accelerator for deep convolutional neural networks for the spark environment. In *Smart Cloud (SmartCloud), IEEE International Conference on*, pages 126–133. IEEE, 2016.

[127] 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.

[128] Kohei Nakamura, Ami Hayashi, and Hiroki Matsutani. An fpga-based low-latency network processing for spark streaming. In *Proceedings of the Workshop on Real-Time and Stream Analytics in Big Data (IEEE BigData 2016 Workshop)*, 2016.

[129] 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.

[130] 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.

[131] 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.

[132] David Ojika, Piotr Majcher, Wojciech Neubauer, Suchit Subhaschandra, and Darin Acosta. Swif: A simplified workload-centric framework for fpga-based computing. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pages 26–26. IEEE, 2017.

[133] 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.

[134] 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.

[135] Kwanghyun Park, Yang-Suk Kee, Jignesh M Patel, Jaeyoung Do, Chanik Park, and David J Dewitt. Query processing on smart ssds. *IEEE Data Eng. Bull.*, 37(2):19–26, 2014.

[136] 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.

[137] 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.

[138] Seth Hintze Pugsley. *Opportunities for near data computing in MapReduce workloads.* PhD thesis, The University of Utah, 2015.

[139] 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.

[140] 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.

[141] Luis Cavazos Quero, Young-Sik Lee, and Jin-Soo Kim. Self-sorting ssd: Producing sorted data inside active ssds. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–7. IEEE, 2015.

[142] Milan Radulovic, Darko Zivanovic, Daniel Ruiz, Bronis R de Supinski, Sally A McKee, Petar Radojković, and Eduard Ayguadé. Another trip to the wall: How much will stacked dram benefit hpc? In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 31–36. ACM, 2015.

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

[144] 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.

[145] 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.

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

[147] 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.

[148] 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.

[149] Oren Segal, Philip Colangelo, Nasibeh Nasiri, Zhuo Qian, and Martin Margala. Sparkcl: A unified programming framework for accelerators on heterogeneous clusters. *arXiv preprint arXiv:1505.01120*, 2015.

[150] Oren Segal, Martin Margala, Sai Rahul Chalamalasetti, and Mitch Wright. High level programming framework for fpgas in the data center. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–4. IEEE, 2014.

[151] Sudharsan Seshadri, Mark Gahagan, Meenakshi Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *OSDI*, pages 67–80, 2014.

[152] 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.

[153] 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.

[154] 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.

[155] 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.

[156] 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.

[157] Patrick Siegl, Rainer Buchty, and Mladen Berekovic. Data-centric computing frontiers: A survey on processing-in-memory. In *Proceedings of the Second International Symposium on Memory Systems*, pages 295–308. ACM, 2016.

[158] 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.

[159] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[160] 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.

[161] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.

[162] 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.

[163] 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.

[164] 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.

[165] Kuen Hung Tsoi and Wayne Luk. Axel: a heterogeneous cluster with fpgas and gpus. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 115–124. ACM, 2010.

[166] Yaman Umuroglu, Donn Morrison, and Magnus Jahre. Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–8. IEEE, 2015.

[167] Cornelis Jan van Leeuwen, Przemyslaw Pawelczak, CJ van Leeuwen, CJ van Leeuwen, AHR Halma, K Schutte, CJ van Leeuwen, J Sijs, and Z Papp. Cocoa: A non-iterative approach to a local search (a) dcop solver. In *AAAI*, pages 3944–3950, 2017.

[168] Erik Vermij, Leandro Fiorin, Christoph Hagleitner, and Koen Bertels. Sorting big data on heterogeneous near-data processing systems. In *Proceedings of the Computing Frontiers Conference*, pages 349–354. ACM, 2017.

[169] Jiajun Wang, Reena Panda, and Lizy Kurian John. Prefetching for cloud workloads: An analysis based on address patterns. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*, pages 163–172. IEEE, 2017.

[170] 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.

[171] 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.

[172] 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.

[173] 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.

[174] Zeke Wang, Shuhao Zhang, Bingsheng He, and Wei Zhang. Melia: A mapreduce framework on opencl-based fpgas. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3547–3560, 2016.

[175] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C Hoe. A study of pointer-chasing performance on shared-memory processor-fpga systems. In *Proceedings of the 2016*

ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 264–273. ACM, 2016.

[176] 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.

[177] 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.

[178] 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.

[179] Xilinx. Memory Interface Solutions User Guide UG086 (v3.6), September 21, 2010, howpublished = `https://www.xilinx.com/support/documentation/ip_documentation/ug086.pdf`.

[180] Xilinx. SDSoC Environment User Guide UG1027(v2016.3, november 30,2016. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_3/ug1027-sdsoc-user-guide.pdf`.

[181] Xilinx. Xilinx All Programmable SoCs. URL `https://www.xilinx.com/products/silicon-devices/soc.html`.

[182] 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.

[183] 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.

[184] 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.

[185] 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.

[186] 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.

[187] Jackson HC Yeung, CC Tsang, Kuen Hung Tsoi, Bill SH Kwan, Chris CC Cheung, Anthony PC Chan, and Philip HW Leong. Map-reduce as a programming model for custom computing machines. In *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*, pages 149–159. IEEE, 2008.

[188] 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.

[189] Zhibin Yu, Wen Xiong, Lieven Eeckhout, Zhendong Bei, Mendelson Avi, and Chengzhong Xu. Mia: Metric importance analysis for big data workload characterization. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[190] 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.

[191] 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.

[192] 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.

[193] Chi Zhang, Ren Chen, and Viktor Prasanna. High throughput large scale sorting on a cpu-fpga heterogeneous platform. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 148–155. IEEE, 2016.

[194] Chi Zhang and Viktor Prasanna. Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 35–44. ACM, 2017.

[195] 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.

[196] 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.

[197] 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.

[198] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 1–6. IEEE, 2010.