Docker Container-based Scalable Partitioning for Apache Spark Scale-up Server Scalability

ABSTRACT

We propose a Docker container-based scalable partitioning method to improve performance scalability for Apache Spark on a scale-up server through eliminating garbage collection and remote socket access overheads by using an efficient partitioning method. The proposed docker container-based partitioning method is evaluated using representative benchmark programs, and evaluation revealed that our partitioning method showed performance improvement by ranging from 1.1x through 1.7x on a 120 core scale-up system.

Keywords

Apache Spark; Scale-up; Docker; Scalability

1. INTRODUCTION

Popular big data analytics infrastructures(e.g, Spark [23], Hadoop [20]) have been developed for a cluster scale-out environment, which adds nodes to a cluster system. On the other hand, scale-up environment, which adds resources(e.g, cpu, memory) to a single node system, only special-purpose techniques exist. In science and machine learning fields, researchers commonly used scale-up server environments, and they now need big data analytics frameworks [9]. Moreover, hardware trends are beginning to change for the scale-up server; a scale-up server can now have substantial CPU, memory, and storage I/O resources [3]. Therefore, the big data analytics infrastructure on scale-up server is also important.

Spark is one of widely used big data analytics framework. However, Spark has been reported that it does not scale on the single node scale-up server because of garbage collection(GC) overhead [2] [16] [14] and locality of memory accesses on Non-Uniform Memory Access(NUMA) architecture [8]. Therefore, A.J. Awan *et al.* analysed GC time and compared state of the art garbage collectors by changing the GC. Moreover, in order to avoid high costs of remote memory access, researchers have attempted to create a new NUMA balancing [12] [19], but these methods also can not satisfactory compared to partitioning approch(see section **??**).

Our goals is to reduce the GC and the memory access latency overheads that have been a major problem of Spark scalability. To achieve our goal, this paper presents a new partitioning method that eliminates the GC and remote access overheads by applying Docker container-based efficient partitioning for Spark on scale-up server. Our basic key idea is that shared-memory system is dealt with as the distributed-system using partitioning approach in order to eliminate GC and memory access overheads. We use Docker containers since the overhead of a Docker container is much smaller than a traditional virtual machine [15] and the container-based approach can easily combine existing container management solutions such as Google Borg [21] and Kubernets [1].

Our method make shared resource to small size group as much as possible(minimal partition value is per-socket) because prior work have showed that shared resource contention should be minimized by partitioning shared resource accesses [17]. Small size cpu groups can mitigate the thread serialized problem cased by GC pause time, and these group may only access to local NUMA memory. Moreover, partitioning method can somewhat reduce the operating systems scalability problems(e.g, address space problem [10] [11], cache communication overhead [7] [13])

To evaluate our approach, we applied our partitioning method on 120 core scale-up server. A too small size partitioning may reduce GC overhead and remote memory access, but the benefits do not come for free because it may cause straggler tasks problem [16] [18]. Thus, this paper additionally describes performance scalability depending on partitioning size. Evaluation of the proposed best-fit partitioning on a 120 core system reveals that the execution times could be improved by 1.6x, 1.7x, 1.5x and 1.1x for Word Count, Naive Basian, Grep and K-means, respectively.

Contributions. Our research makes the following contributions:

- We analyzed Apache Spark performance scalability on 120 core scale-up server. The results of scalability was that parallel GC can improve performance scalability up to 60 core, but then the GC flattens out after 60 core.
- We evaluated proposal partitioning approach on a manycore scale-up server thereby mitigating scale-up server scalability problems in BigDataBench. Our approach improved throughput and execution time from 1.1x through 1.7x on 120 core.

The rest of this paper is organized as follows. Section 2 describes the test-bed and Spark scalability problem. Section 3 describes the our partitioning approach and Section 4 shows the results of the experimental evaluation. Section 5 describes related works. Finally, section 6 concludes the paper.

2. SCALE-UP SERVER SCALABILITY

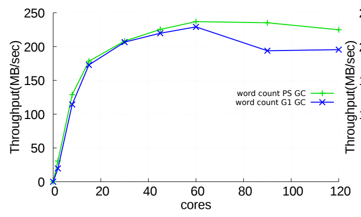
2.1 Test-bed and Benchmark

Apache Spark. Apache Spark is a framework for large scale distributed computation. RDD(Resilient Distributed Datasets) is

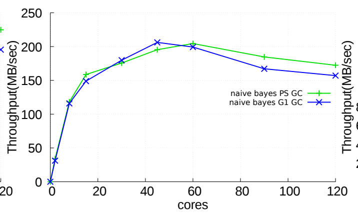
XXXXX 'XXXX', USA

© 2016 ACM. ISBN XX. . . $15.00

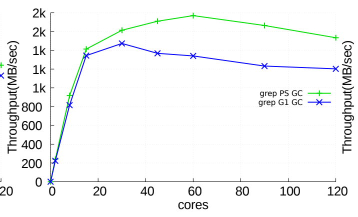
DOI: XX.XXX/XXX_X



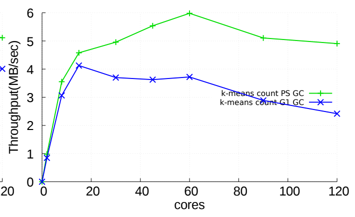
(a) Word Count



(b) Naive Basian

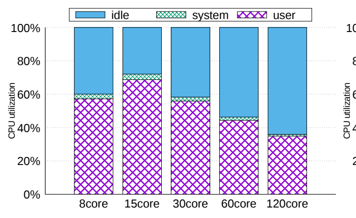


(c) Grep

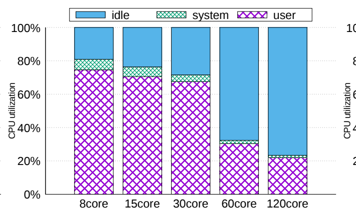


(d) K-means

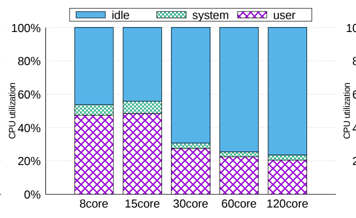
Figure 1: Performance scalability on 120 core.



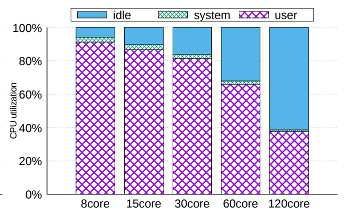
(a) Word Count



(b) Naive Basian



(c) Grep



(d) K-means

Figure 2: CPU utilization on 120 core.

a collection of partitions of records, and the RDD is managed as LRU(Least Recently Used), so when there is not enough memory, Spark evicts the least recently used RDD. Spark may has a substantial performance when data-set can fit in memory.

Test-bed. We use a machine to evaluate on real hardware: an 120-core (8 sockets × 15 cores) Intel Xeon E7-8870 (the same machine used for evaluation in chapter X) and, to show that our conclusions generalize. Hyper-Threading is disabled, and we used Linux kernel 4.5-rc6.

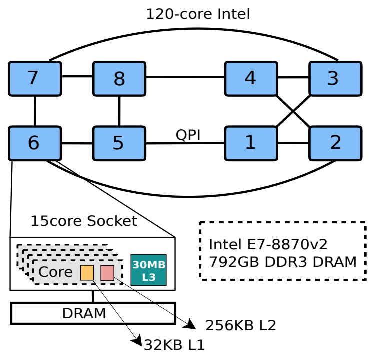


Figure 3: Test-bed intel xeon archtecture.

Benchmark. We used BigData Benchmark.

- **Word Count.** We have developed a novel lightweight log-based structures with efficient log management implementation.
- **Grep.** We applied the in Linux kernel to two reverse mapping(anonymous, file) on Our design improved throughput and execution time from 1.5x through 2.7x on 120 core.
- **Naive Basian.** We applied the in Linux kernel to two reverse mapping(anonymous, file) on Our design improved throughput and execution time from 1.5x through 2.7x on 120 core.
- **K-means.** We applied the in Linux kernel to two reverse mapping(anonymous, file) on Our design improved throughput and execution time from 1.5x through 2.7x on 120 core.

2.2 Spark Scalability Problem

Figure 1 shows the Spark scalability of five workloads with two state of the art garbage collection, G1 and Parallel Scavenge(PS). Up to 60 core, the five workloads scales lineally and then GC pause becomes bottlenecks. The Word Count workload flattens out after 60 core, and other benchmarks slightly go down because not only the GC but also the remote memory access overheads. To evaluate state of the art GC, we compared the G1 and PS GC. The effect of changing to the GC is the PS outperforms G1 by 3.3x on 120 core. However, although we used the state of the art scalable GC, the Spark performance scalability still suffers from GC and NUMA locality problem.

Our goal is to maximize CPU utilization, so we profiled the CPU utilization of five workloads. Figure 2 shows the CPU utilization. Th y-axis is the percentage of time spent in kernel-space code(sys), user-space code(user), and idle time(idle). All benchmarks increase the idle time due to the GC pause as core counts increase.

2.3 Benefit of JVM Partitioning

Spark and Hadoop frameworks use JAVA, and it needs java virtual machine(JVM), so understanding the partitioning is important. To preliminarily analyse the JVM partitioning effect, we conducted benchmarking by using SPECjbb2013 [**?**], which is a state of the art benchmark for JVM performance. We used two different experimental settings. First, we used per-socket JVM partitioning by using the NUMA control application(numactl). Second, we set maximum JVM heap size, which is system available memory size, and threads are scheduled by the OS in order to migrate any core, and we enable automatic NUMA balancing feature in the Linux kernel.

The results shows that partitioning approach outperforms non-partitioning approach by Xx on 120 core. Therefore, in manycore scale-up server, partitioning approach has many advantages over non-partitioning approach in terms of performance scalability.

3. PARTITIONING FOR SPARK

The reason for using partitioning method is that Spark library and run-time engine can be bottleneck by GC and remote memory access because Spark have not focused on scale-up environment. To achieve Spark performance scalability, we use the Docker container-based partitioning method to eliminate the GC and re-

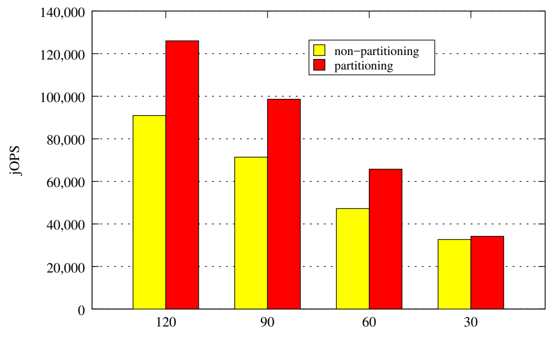


Figure 4: Test-bed Intel xeon architecture.

mote memory access overheads. This section explains design aspects of our Docker container-based partitioning method to solve GC and memory latency.

3.1 Design Consideration

As noted earlier, the major problem of Spark scalability is GC, so partitioning approach is needed. Indeed, GC leads to many of the advantages of high-level languages because of an increase in productivity, while it is a double-edged sword because GC pauses may lead a serialized operation and requests to take unacceptable long times. In order to reduce the GC pause time, a simple method is minimizing the CPU counts. Therefore, the first design consideration of scale partitioning is to minimize GC pause times.

The second design consideration is locality issues because of NUMA architecture DRAM access latency. Due to the fact that threads are scheduled by the OS to execute on any core, the thread is migrated to different memory area, so the migrated thread may access remote memory. Partitioning approach can prevent to migrate other socket. Indeed, the modern operating systems(Linux) has a NUMA balancing feature for enhancement of memory locality, but partitioning method can more superior performance regarding the large scale-up server(8 socket) [19].

In addition to GC and NUMA effect, operating systems noise can pose scalability bottleneck because modern operating systems have been designed for shared-memory systems;therefore, the next design consideration is to avoid operating systems noise. For example, Single address space sharing problem [10] [11] between multi-threaded applications, scheduler bottlenecks [**?**], and cache communication bottlenecks [7] [13] are major problems in manycore scale-up server operating systems. This problem cause by sharing resource, so our approach can solve these resource contention problem by using partitioning approach.

To satisfy these factors such as GC, NUMA and operating system bottlenecks cause by shared-memory system, Spark on scale-up server should work as distributed system concept. Therefore, we use partitioning approach that treats the partitioned cores as a cluster node and moves shared-memory system workers to a distributed system workers that communicate via message-passing thereby eliminating GC and remote memory access overheads. We identify two design principles: (1) make shared resource to small size group as much as possible, (2) make Spark infrastructure hardware-neutral. As result of, the small size CPU groups can mitigate the thread serialized problem cased by GC pause time, and these group may only access to local NUMA memory.

The final design consideration is straggler tasks(i.e, tasks take significantly longer than expected to complete) problem. Even though too small size partitioning may reduce GC and NUMA remote access, its benefit does not come for free because it may cause straggler tasks problem [16] [18]. Thus, in order to scale Spark performance scalability, a straggler monitor and a run-time core injector are needed.

3.2 Towards a Container-based Framework

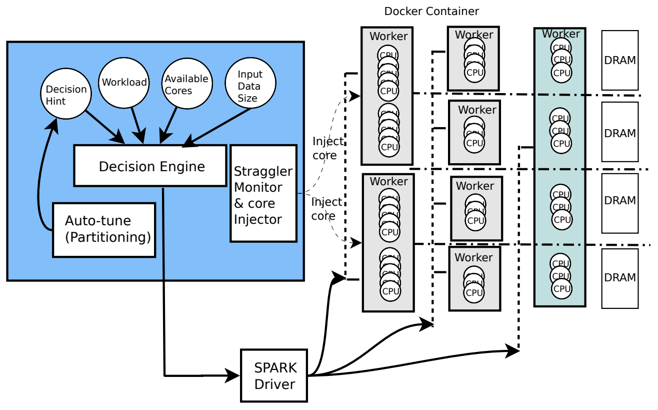


Figure 5: Overview of the docker contrainer-based partitioning

This section describes our vision that will accommodate the previous mentioned design consideration. Our proposed scalable partitioning framework is figure 5 with the necessary features. The left side of figure shows our proposed framework, and the right side of figure shows isolated Docker containers and per-socket CPU with memory.

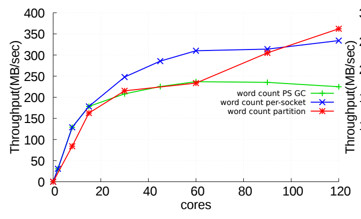
Decision engine is one of the most important features since every partitioning regarding Spark system workers are based on our decision engine component. The basic function of the decision engine chooses whether or not the job run on the Docker container. The necessity of the auto-tuner is that performance scalability depending on partitioning size commonly differs from each server architecture. To maximized CPU utilization, the straggler monitor and core injector are needed because straggler tasks prolong job completion times, so the early finished CPUs can inject to other Docker containers involved the straggler tasks.

4. EVALUATION

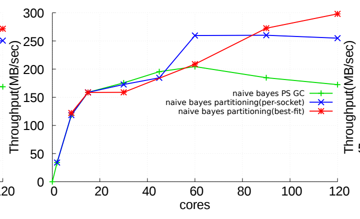
In this section we discuss the docker container-based partitioning on the scale-up server described in Section 3. We ran the four benchmarks on Linux 4.5-rc4 with stock Linux. We used ram file system for HDFS due to the eliminating the HDFS bottleneck.

We used four different experiment settings. First, we used non-partitioning method as Figure section 2 graph and we set heap size(4G). Second we used fine-grained partitioning that is per-socket(15 core) partitioning because it can make maximize locality. We allocated heap size by modifying heap size is that we divide 4G of number of partitioning. Finally, we used coarse-grained partitioning that is per-socket(30 core) partitioning since it can mitigate the straggler tasks problem.

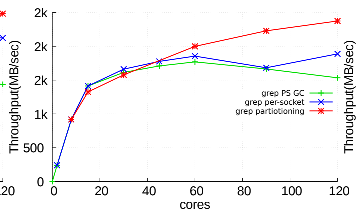
The results for Word Count are shown in Figure 6(a), and the result shows the throughput of BigDataBench with our four different settings. Up to 60 core, the PS GC version of non-partitioning approach scales linearly and then it flattens out. However, up to 60 core, our per-socket partitioning outperform non-partitioning since it can remove GC and NUMA latency overheads, and then the straggler tasks problem become bottlenecks. Our coarse-grained partitioning outperforms non-partitioning by 1.5x and per-socket



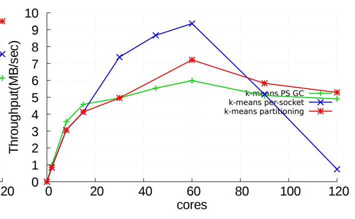
(a) Word Count



(b) Naive Basian

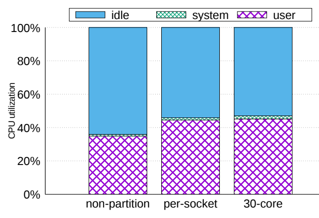


(c) Grep

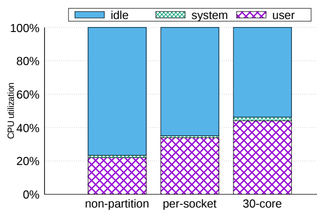


(d) K-means

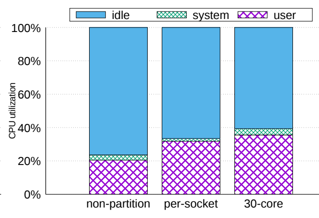
Figure 6: Performance scalability using docker container on 120 core.



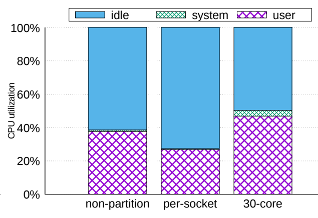
(a) Word Count



(b) Naive Basian



(c) Grep



(d) K-means

Figure 7: CPU utilization on 120 core.

partiting by 1.1x on 120 core. Furthermore, the non-partitioning approach has the highest idle time(64%) since GC becomes bottleneck(see figure 7). The results(Figure **??**(b)) for Naive Bayesian is similar to Word Count workload, so our best-fit partitioning outperforms non-partitioning by 1.5x and per-socket by 1.2x on 120 core.

The results for Grep are shown in Figure 6(c). After to 60 core, the 30 core partitioning approach scales linearly, but the others throughput goes down after to 60 core because non-partitioning version suffers from GC. The per-socket partitioning approach suffers from the straggler tasks problem. Therefore, although the per-socket partitioning approach eliminates the GC overhead and the remote memory access, its CPU utilization(23%) is low than 30 core partitioning(38%). Our best-fit partitioning outperforms non-partitioning by 1.5x and per-socket by 1.3x on 120 core.

The results for K-means are shown in Figure 6(d), The K-means workload extremely suffers from GC [2];therefore, per-socket partitioning approach has better performance scalability up to 60 core. However, then it collapses after 60 core since it extremely suffers from the straggler tasks problem that extends job completion times. Our best-fit partitioning outperforms non-partitioning by 1.1x on 120 core. Thus, per-socket partitioning approach has the lowest(72%) idle time. On the other hand, 30 core partitioning approach relatively less suffers from the straggler tasks problem.

5. RELATED WORK

Apache Spark Scalability. To improve the Spark scalability, Scale out Vs Scale Up researchers have attempted to create new operating or have attempted to optimize existing operating Our research belongs to optimizing existing operating systems in order to solve the Linux fork scalability problem. However, previous research did not deal with the anonymous reverse mapping, which is one of the fork scalability bottleneck.

Manycore Scale-up Server Scalability. To improve the scalability, researchers have attempted to create new operating systems [5] [4] or have attempted to optimize existing operating systems [6] [10] Our research belongs to optimizing existing operating systems in order to solve the Linux fork scalability problem. However, previous research did not deal with the anonymous reverse mapping, which is one of the fork scalability bottleneck.

6. CONCLUSION AND FUTURE WORKS

We propose a docker container-based partitioning method for Apache Spark scalability on scale-up server. To eliminate GC and NUMA effect, we divide per-socket and best-fit partitioning using the docker container. Evaluation results using the wordcount, navi-basian, grep and k-means reveal that our method better performance up to 1.5 times compared to existing solutions.

Future Directions. To achieve our goal, this paper only focused on the manual docker container-based partitioning, and our future directions are:

- **Solving the straggler tasks problem.** straggler tasks significantly extend job completion times. To mitigate this problem, we may use dynamic resource allocation solution in dockers to maximized cpu utilization for the straggler tasks by using our new resource hand-over solution.

- **Auto-tuned partitioning.** In this paper, only support manualy partition using docker container. However, many of sicent or big data acaldld may reuse similar workload, so training phase for finding the best-fit partitioning can be superior solution than manually partition similar way of compiler-based auto-tuner[].

7. REFERENCES

[1] Kubernetes. http://kubernetes.io/.

[2] Ahsan Javed Awan, Mats Brorsson, V. V. E. A. How data volume affects spark based data analytics on a scale-up server. In *Big Data Benchmarks, Performance Optimization, and Emerging Hardware* (2016), Springer, Lecture Notes in Computer Science.

[3] Appuswamy, R., Gkantsidis, C., Narayanan, D., Hodson, O., and Rowstron, A. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing* (2013), SOCC '13.

[4] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings*

of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (2009), SOSP '09, pp. 29–44.

[5] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), OSDI'08, pp. 43–57.

[6] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10, pp. 1–16.

[7] BOYD-WICKIZER, S., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. OpLog: a library for scaling update-heavy data structures. In *Technical Report MIT-CSAIL-TR2014-019* (2014).

[8] CAO, X., PANCHPUTRE, K. K., AND DU, D. H.-C. Accelerating data shuffling in mapreduce framework with a scale-up numa computing architecture. In *Proceedings of the 24th High Performance Computing Symposium* (2016), HPC '16, pp. 17:1–17:8.

[9] CHAIMOV, N., MALONY, A., CANON, S., IANCU, C., IBRAHIM, K. Z., AND SRINIVASAN, J. Scaling spark on hpc systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing* (2016), HPDC '16, pp. 97–110.

[10] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII, pp. 199–210.

[11] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. A. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, pp. 211–224.

[12] DASHTI, M., FEDOROVA, A., FUNSTON, J., GAUD, F., LACHAIZE, R., LEPERS, B., QUEMA, V., AND ROTH, M. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS '13.

[13] HENDLER, D., INCZE, I., SHAVIT, N., AND TZAFRIR, M. Flat Combining and the Synchronization-parallelism Tradeoff. SPAA '10, pp. 355–364.

[14] MAAS, M., ASANOVIĆ, K., HARRIS, T., AND KUBIATOWICZ, J. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ASPLOS '16, pp. 457–471.

[15] MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J. 2014* (2014).

[16] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (2015), NSDI'15.

[17] QURESHI, M. K., AND PATT, Y. N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (2006), MICRO 39, pp. 423–432.

[18] REN, X., ANANTHANARAYANAN, G., WIERMAN, A., AND YU, M. Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15, pp. 379–392.

[19] RIK VAN RIEL, V. C. Automatic numa balancing. https://www.linux-kvm.org/images/7/75/01x07b-NumaAutobalancing.pdf.

[20] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010), MSST '10.

[21] VERMA, A., PEDROSA, L., KORUPOLU, M. R., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Bordeaux, France, 2015).

[22] WANG, L., ZHAN, J., LUO, C., ZHU, Y., YANG, Q., HE, Y., GAO, W., JIA, Z., SHI, Y., ZHANG, S., ET AL. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (2014), IEEE, pp. 488–499.

[23] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (2012), NSDI'12.