# 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. To improve the efficiency and minimize the costs, we eliminate garbage collection and remote socket access overheads through 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 [25], Hadoop [22]) have been developed for a cluster scale-out environment, which adds nodes to a cluster system. On the other hand, for 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 use scale-up server environments, and they now need big data analytics frameworks [8]. 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 [4]. 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] [17] [15] and locality of memory accesses on Non-Uniform Memory Access(NUMA) architecture [7]. A.J. Awan *et al.* analysed GC time and compared state of the art garbage collectors by changing the GC. In order to avoid high costs of remote memory access, researchers have attempted to create a new NUMA balancing [12] [21], but these methods also can not satisfactory compared to partitioning approch(see section 2).

Our goals is to reduce the GC and the remote memory access overheads that have been major problems 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 a 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 overheads of Docker container are much smaller than a traditional virtual machine [16] and the container-based approach can easily combine the existing container management solutions such as Google Borg [23] and Kubernets [1].

Our method makes 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 [19]. Small size CPU groups can mitigate the thread serialized problem cased by GC pause time, and these group may only access the local NUMA memory. Moreover, partitioning method can somewhat reduce the operating systems scalability problems(e.g, address space problem [9] [11], cache communication overhead [6] [13])

To evaluate our approach, we manually applied our partitioning method on 120 core scale-up server. Though 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 [17] [20]. 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 were 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 large scale-up server thereby mitigating the scale-up server scalability problems in BigDataBench [24]. 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

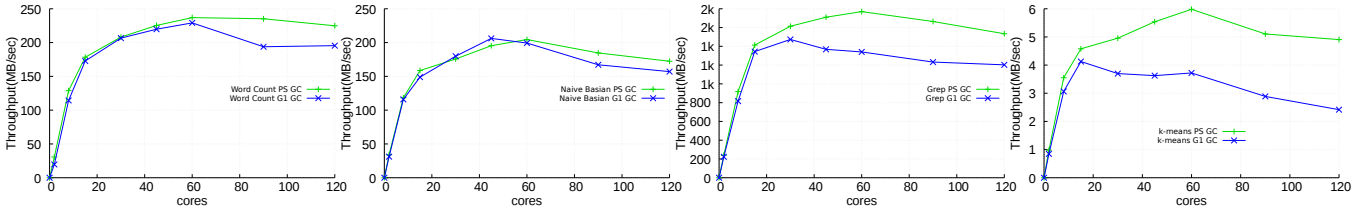



*XXXXX 'XXXX', USA*



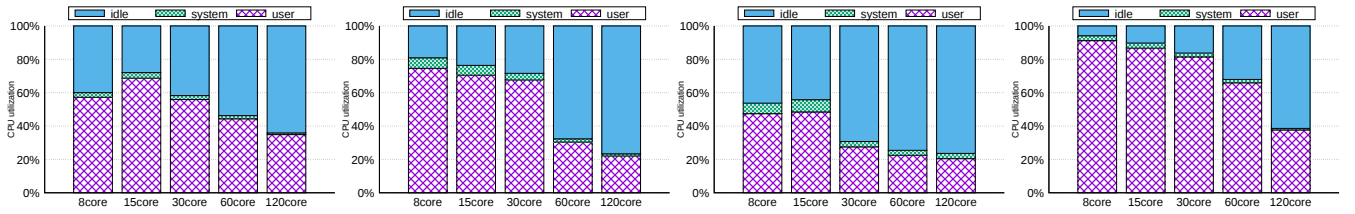


DOI: XX.XXX/XXX_X




(a) Word Count (b) Naive Basian (c) Grep (d) K-means

Figure 1: Performance scalability.




(a) Word Count (b) Naive Basian (c) Grep (d) K-means

Figure 2: CPU utilization.

## 2.1 Test-bed and Benchmark

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

**Test-bed.** We used 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 section 4) and, to show that our conclusions generalize. Hyper-Threading was disabled, and we used Linux kernel 4.5-rc6.

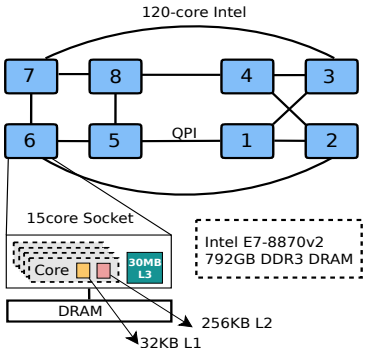



Figure 3: Test-bed intel xeon archtecture.

**Benchmark.** We used the BigDataBench.

| Workload | Input data size | Heap size | Configuration | Data type |
|---|---|---|---|---|
| Word Count | 10G | 4G | none | text |
| Naive Basian | 10G | 4G | none | text |
| Grep | 30G | 4G | "the" | text |
| K-means | 4G | 4G | k=8 | graph |

| JVM | Spark | Hadoop | OS | Distribution |
|---|---|---|---|---|
| Openjdk 1.8.0_91 | 1.3.1 | 1.2.1 | Linux 4.5-rc6 | Ubuntu 14.04 |

Table 1: System information and configuration values.

Table 1 shows our configurations. we used four workloads(Word Count, Naive Basian, Grep and K-means). For the simplicity of experiment, we used input data as the table 1. Of course, large Spark heap size can eliminate the GC overhead, but commonly the input data size is larger than the heap size in big data analytics area; we used the smaller heap size than the input data size.

## 2.2 Spark Scalability Problem

Figure 1 shows the Spark scalability of four workloads with two state of the art garbage collections, G1 and Parallel Scavenge(PS). Up to 60 core, the four workloads scale 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 overhead but also the remote memory access. To evaluate state of the art GC, we compared the G1 with PS GC. The effect of changing to the GC is the PS outperforms G1 up to 2.0x on 120 core. Although we used the state of the art scalable GC, the Spark performance scalability still suffers from GC and NUMA locality problems, and we could not see any significant differences when increasing the size of executors.

Our goal is to maximize CPU utilization, so we profiled the CPU utilization on the four workloads. Figure 2 shows the CPU utilizations. 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 they need java virtual machine(JVM), so understanding the JVM partitioning is important. To preliminarily analyse the JVM partitioning effect, we conducted benchmarking by using SPECjbb2013 [18], 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, an available system memory size, and threads are scheduled by the OS 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 1.4x on 120 core(figure 4). Therefore, in manycore scale-up server, partitioning approach has many ad-

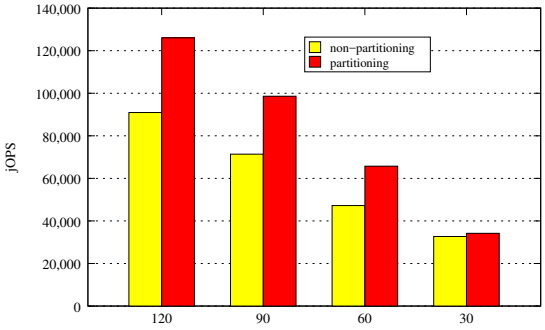



Figure 4: Effect on JVM partitioning.

vantages 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 remote 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 a 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 remote memory access on the NUMA architecture. Due to the fact that threads are scheduled by the OS to execute on any core, the thread is migrated to different memory area, and then the migrated thread may access remote memory. Partitioning approach can prevent to migrate other socket. Indeed, the modern operating systems have a NUMA balancing feature for enhancement of memory locality, but partitioning method can more superior performance regarding the large scale-up server(8 socket) [21].

In addition to GC and NUMA effects, operating systems noise can pose scalability bottlenecks 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 [9] [11] between multi-threaded applications, scheduler bottlenecks [14] and cache communication bottlenecks [6] [13] are major problems in manycore scale-up server operating systems. These problems are caused by sharing resource, so our approach can solve these resource contention problems by using partitioning approach.

To satisfy these factors such as GC, NUMA and operating system bottlenecks, Spark on scale-up server should run as distributed system concept. Therefore, we use the partitioning approach that treats the partitioned cores as a cluster node and moves shared-memory system workers to distributed system workers that communicate via message-passing thereby eliminating GC and remote memory access overheads. As result of, the small size CPU groups can mitigate the thread serialized problem caused by GC pause time, and these partitioned 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 [17] [20]. 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

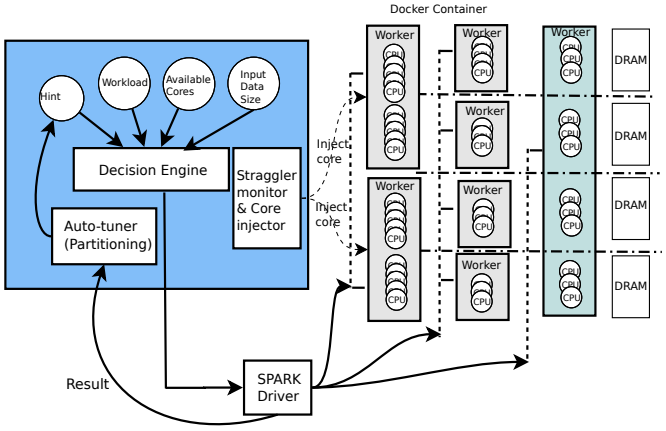



Figure 5: Overview of towards a docker contrainer-based partitioning

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 should inject to other Docker containers which contained the straggler tasks.

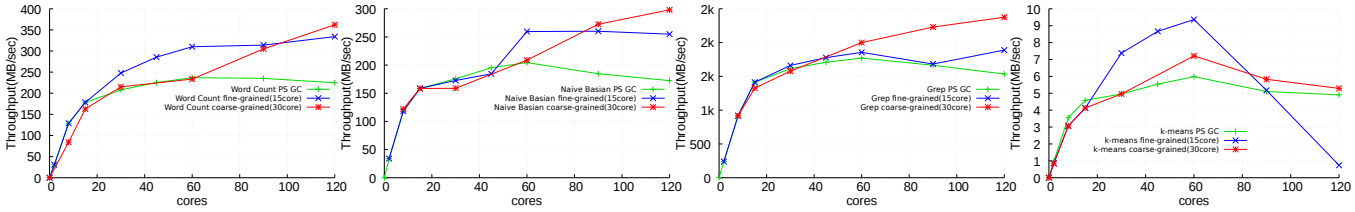
# 4. EVALUATION

In this section we discuss the docker container-based partitioning on the scale-up server described as section 2. We used ram file system for HDFS due to the eliminating the HDFS bottleneck.

| method | executor heap size | number of partitions |
|---|---|---|
| non-partition | 4G | 1 |
| coarse-grained(30 core) | 1G | 4 |
| fine-grained(15 core) | 512M | 8 |

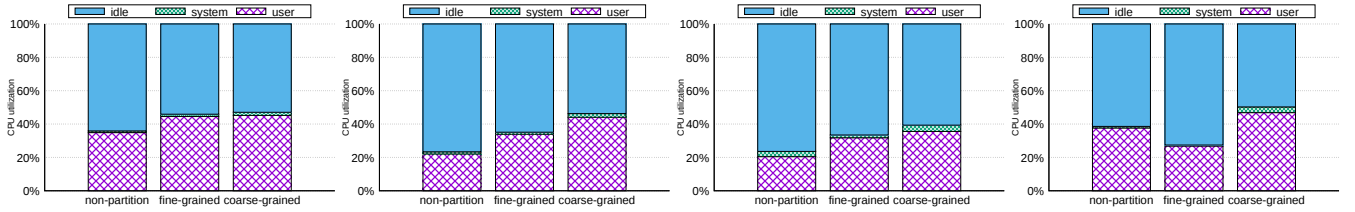
Table 2: Partitioning values.

We used three different experiment settings. First, we used the non-partitioning method as section 2 and we set heap size(4G). Sec-




(a) Word Count (b) Naive Basian (c) Grep (d) K-means

Figure 6: Performance scalability using docker container.




(a) Word Count (b) Naive Basian (c) Grep (d) K-means

Figure 7: CPU utilization on 120 core.

ond we used the fine-grained partitioning(15 core) partitioning because it can make maximize locality. Table 2 shows our partitioning values. The heap size of executor in the partitioned Docker is divided by number of partitions. Finally, we used the coarse-grained partitioning(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 with our three 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 fine-grained partitioning outperforms 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 fine-grained partitioning 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 6(b)) for Naive Bayesian is similar to Word Count workload. Our coarse-grained partitioning outperforms non-partitioning by 1.5x and fine-grained by 1.2x on 120 core.

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

The results for K-means are shown in Figure 6(d), The K-means workload suffers from GC [2];therefore, fine-grained partitioning approach has substantial 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 coarse-grained partitioning outperforms non-partitioning by 1.1x on 120 core. Thus, fine-grained partitioning approach has the lowest(72%) idle time. On the other hand, coarse-grained partitioning approach relatively less suffers from the straggler tasks problem.

# 5. RELATED WORK

**Apache Spark Scalability.** To improve the Spark scalability, researchers have attempted to optimize for scale-out server [17] [15] or to optimize scale-up server [2] [8]. Our research belongs to optimizing for scale-up server to eliminate GC overheads and to enhance the locality on NUMA architecture. However, previous studies did not considered Docker container-based partitioning, which can clearly reduce memory contention, and it can maximize locality of memory access. Furthermore, it can easily combine other container management solutions.

**Scale-up Server Scalability.** To improve the Spark scalability, researchers have attempted to apply distributed system concepts to shared memory systems [5] [6]. Barrelfish [5] creates a new operating system for efficient cache-coherent shared memory system by building an OS using message-based architecture. Our research also brings about distributed system concepts, but our approach applies to user level Spark framework instead of OS because OS can achieves performance scalability by commuting interface [10].

# 6. CONCLUSION AND FUTURE WORKS

We proposed a docker container-based partitioning method for Apache Spark scalability on scale-up server. To eliminate GC and remote memory access, we divided per-socket and best-fit partitioning using the docker container. Evaluation results(Word Count, Naive Basian, Grep and K-means) reveal that our method has substantial performance up to 1.7 times compared to existing solutions.

**Future Directions.** 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.

- **Implementing the auto-tuner.** In this paper, only support manually partitioning method. However, many of science applications or big data analytics may reuse similar workloads, so training phase for finding the best-fit partitioning can be a superior solution in a way similar to compiler-based auto-tuner [3].

## 7. REFERENCES


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

[2] V. V. E. A. Ahsan Javed Awan, Mats Brorsson. How Data Volume Affects Spark Based Data Analytics on a Scale-up Server. In *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, Springer, Lecture Notes in Computer Science, 2016.

[3] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, 2014.

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

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, 2009.

[6] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. OpLog: a library for scaling update-heavy data structures. In *Technical Report MIT-CSAIL-TR2014-019*, 2014.

[7] X. Cao, K. K. Panchputre, and D. H.-C. Du. Accelerating Data Shuffling in MapReduce Framework with a Scale-up NUMA Computing Architecture. In *Proceedings of the 24th High Performance Computing Symposium*, HPC '16, pages 17:1–17:8, 2016.

[8] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan. Scaling Spark on HPC Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 97–110, 2016.

[9] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, 2012.

[10] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13.

[11] A. T. Clements, M. F. Kaashoek, and N. a. Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, 2013.

[12] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. 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*, ASPLOS '13, 2013.

[13] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. SPAA '10, pages 355–364, 2010.

[14] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, 2016.

[15] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz. 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*, ASPLOS '16, pages 457–471, 2016.

[16] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014, 2014.

[17] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, 2015.

[18] C. Pogue, A. Kumar, D. Tollefson, and S. Realmuto. SPECjbb2013 1.0: An Overview. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14.

[19] M. K. Qureshi and Y. N. Patt. 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*, MICRO 39, pages 423–432, 2006.

[20] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu. Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 379–392, 2015.

[21] V. C. Rik van Riel. Automatic NUMA Balancing. https://www.linux-kvm.org/images/7/75/01x07b-NumaAutobalancing.pdf.

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, 2010.

[23] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[24] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499. IEEE, 2014.

[25] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. 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*, NSDI'12, 2012.