Improving Scalability of Apache Spark-based Scale-up Server through Docker Container-based Partitioning

Joohyun Kyong
School of Computer Science,
Kookmin University,
Seoul, South Korea
joohyun0115@gmail.com

Jinwoo Jeon
School of Computer Science,
Kookmin University,
Seoul, South Korea
chichicapo00@gmail.com

Sung-Soo Lim
School of Computer Science,
Kookmin University,
Seoul, South Korea
sslim@kookmin.ac.kr

ABSTRACT
We propose an Apache Spark-based scale-up server architecture using Docker container-based partitioning method to improve performance scalability. The performance scalability problem of Apache Spark-based scale-up servers is due to garbage collection(GC) and remote memory access overheads when the servers are equipped with significant number of cores and Non-Uniform Memory Access(NUMA). The proposed method minimizes the problems using Docker container-based architecture effectively partitioning the original scale-up server into small logical servers. Our evaluation study based on benchmark programs revealed that the partitioning method showed performance improvement by ranging from 1.1x through 1.7x on a 120 core scale-up system. Our proof-of-concept scale-up server architecture provides the basis towards complete and practical design of partitioning-based scale-up servers showing performance scalability.

CCS Concepts
• **Computer systems organization →Multicore architectures**

Keywords
Apache Spark; scale-up; docker; salability

1. INTRODUCTION
Scale-out and scale-up configurations are two different representative methods to implement big data analytics infrastructures. In scale-out server clusters (e.g, Spark [1], Hadoop [2]), server upgrades are performed through adding nodes to the existing cluster system. On the other hand, in scale-up environment, server upgrades are performed through adding resources (e.g, CPU, memory) to the existing single node-based system. Scale-up servers are mostly used in scientific analytics areas [3], and big data analytics frameworks are being increasingly used. Another reason that the scale-up servers are becoming more popular is due to significantly increased resources even on single-node based server system [4]. This naturally requires substantial research on how to improve the performance scalability of scale-up servers.

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) overheads [5] [6] [7] and locality of memory accesses on Non-Uniform Memory Access(NUMA) architecture [8]. In order to minimize the remote memory access costs, researchers have attempted to create a new NUMA balancing [9] [10] and accomplished considerable level of performance improvement, but not satisfiable in scalability aspects.

In order to achieve the performance scalability, we need to devise a framework to avoid the major drawbacks of the Apache Spark regarding GC and remote memory access overheads. Our proposed architecture is based on the reasoning that logically partitioning the original servers into small servers could hide the Spark's performance scalability problems. Therefore, we propose a Docker container-based logical partitioning method for Spark-based scale-up servers. In this paper, we implemented a proof-of-concept architecture using Docker container-based scale-up server while leaving concrete and detailed complete design and implementation of necessary server components to future work.

To evaluate our approach, we manually applied our partitioning method to a 120 core scale-up server. While smaller sized partitioning may further reduce GC overhead and remote memory access, this may cause straggler tasks problem [7] [11]. Thus, this paper additionally addresses the trade-off relationship between the achieved performance scalability and partitioning sizes. Performance 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.

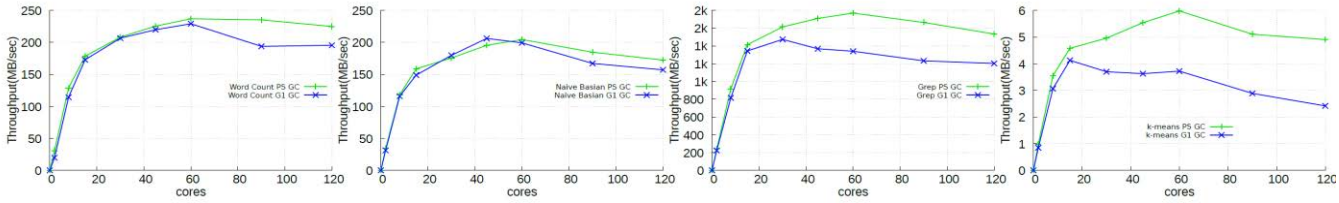
Contributions. Our research provides the following contributions:

- We measured Apache Spark performance scalability on a 120 core scale-up server. The results show that parallel GC only scales well up to 60 core systems while does not show performance improvement on the systems with more than 60 cores.
- We evaluated proposed partitioning approach on a large scale-up server using BigDataBench [12] and the results revealed that the proposed framework significantly mitigate the performance scalability problems of Apache Spark.
- We present a proof-of-concept architecture for Apache Spark-based scale-up servers based on Docker-based logical partitioning.

The rest of this paper is organized as follows. Section 2 describes the test-bed, Spark scalability problem and benefits of partitioning.

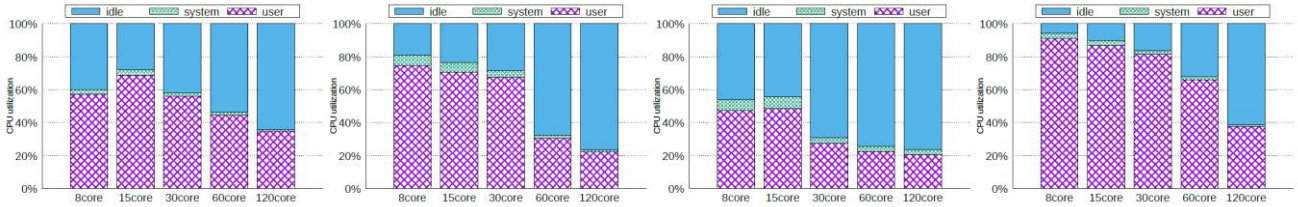
ICSCA 2017, February 26-28, 2017, Bangkok, Thailand

DOI: http://dx.doi.org/10.1145/3056662.3056686



(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.

Section 3 shows our proof-of-concept architecture. Section 4 describes related works. Finally, section 5 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. Resilient Distributed Datasets(RDD) is a collection of partitions of records, and the RDD is managed as Least Recently Used(LRU), 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: a 120-core (8 sockets × 15 cores) Intel Xeon E7-8870. Hyper-Threading was disabled.

Table 1 System information and configuration values.

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

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 size 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. Furthermore, we could not see any significant differences when increasing the size of Spark executors.

Our goal is to maximize CPU utilization, so we profiled the CPU utilization on the four workloads. Figure 2 shows the CPU utilizations. The 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.

2.3 Benefit of JVM Partitioning

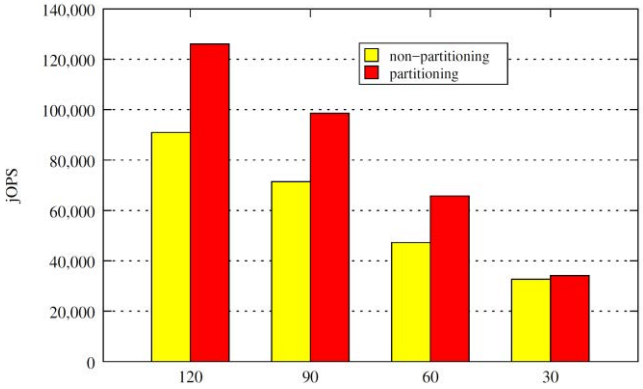
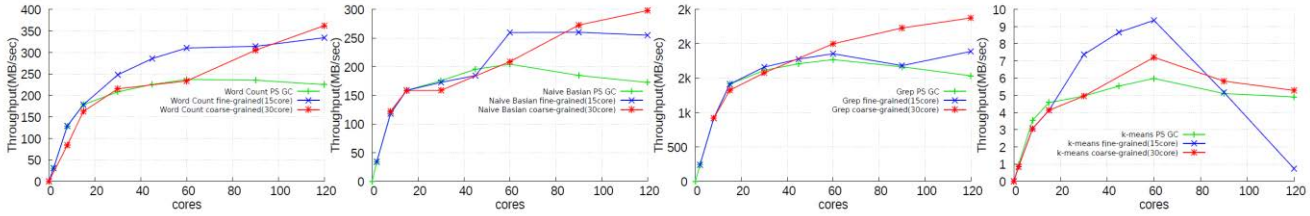


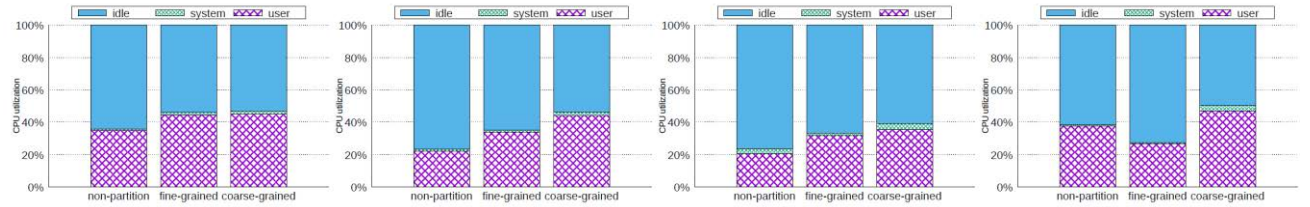
Figure 3 Effect on JVMspartitioning

Spark and Hadoop frameworks use JAVA, and they need java virtual machine(JVM), so understanding the JVM partitioning is important. To preliminary evaluate the JVM partitioning effect, we conducted experiments by using SPECjbb2013 [13], 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 all threads are scheduled by the OS to migrate any core, and we enable automatic NUMA balancing feature in the Linux kernel.



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

Figure 4 Performance scalability using Docker container



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

Figure 5 CPU utilization on 120 core

The results show that partitioning approach outperforms non-partitioning approach by 1.4x on 120 core (see figure 3). Therefore, in manycore scale-up server, partitioning approach has many advantages over non-partitioning approach in terms of performance scalability.

2.4 Benefit of Container-based Partitioning

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 eliminating the HDFS bottleneck.

Table 2 Partitioning values

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

We used three different experiment settings. First, we used the non-partitioning method as section 2(heap size is 4G). Second, we used a fine-grained partitioning (15 core) because it can maximize the NUMA 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) since it can mitigate a straggler tasks problem [7] [11].

The results for Word Count are shown in Figure 4(a). 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 a straggler tasks problem become bottlenecks. Our coarse-grained partitioning outperforms non-partitioning by 1.5x and fine-grained partitioning by1.1x on 120 core. Furthermore, the non-partitioning approach has the highest idle time (64%) since GC becomes bottleneck (see figure 5). The results (Figure 4(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 4(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. 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 4(d), The K-means workload suffers from GC [8]; therefore, fine-grained partitioning approach has substantial performance scalability up to 60 core. However, then it collapses 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. Fine-grained partitioning approach has the lowest (72%) idle time because the coarse-grained partitioning approach relatively less suffers from the straggler tasks problem.

3. PROOF-OF-CONCEPT ARCHITECTURE

As noted earlier, the major problems of Spark scalability are GC overhead and remote memory access, and it can be removed by the Docker container-based partitioning approach, which divides the original scale-up server into small logical servers that treats the partitioned cores as a cluster node and moves shared-memory system workers to distributed system workers that communicate via message-passing.

This section explains design consideration for architecture and our proposed proof-of-concept architecture to solve GC and memory latency.

3.1 Design Consideration

In order to reduce the GC pause time, minimizing CPU counts is a simple method, while it is a double-edged sword because small size of CPU counts may lead communicating bottlenecks. Therefore, the first design consideration is what is best-fit CPU counts for reducing the GC pause time.

The second design consideration is the straggler tasks (i.e., tasks take significantly longer than expected to complete) problem. Even though small size of partitioning may reduce GC, its benefit does not come for free because it may cause straggler tasks

problem. Thus, in order to scale Spark, a straggler monitor and a run-time core injector are needed.

In addition to finding the best-fit CPU counts and reducing the straggler tasks problem, the NUMA locality is next design consideration. Due to the fact that threads are scheduled by the OS to execute on any core, and partitioning approach can prevent to migrate other socket, so it is necessary to divide by per-socket.

Operating systems noise can pose scalability bottlenecks because modern operating systems have been designed for shared-memory systems; therefore, the final design consideration is to avoid operating systems noise. For example, single address space sharing problem [14] [15] between multi-threaded applications, scheduler bottlenecks [16] and cache communication bottlenecks [17] [18] are major problems in manycore scale-up server operating systems. These problems are caused by sharing resource, so the architecture should consider the operating systems noise.

3.2 Architecture

This section describes our vision that will accommodate the previous mentioned design consideration. Our proposed scalable partitioning architecture is Figure 6 with the necessary features. The left side of figure shows our proposed architecture, and the right side of figure shows isolated Docker containers and per-socket CPU with memory.

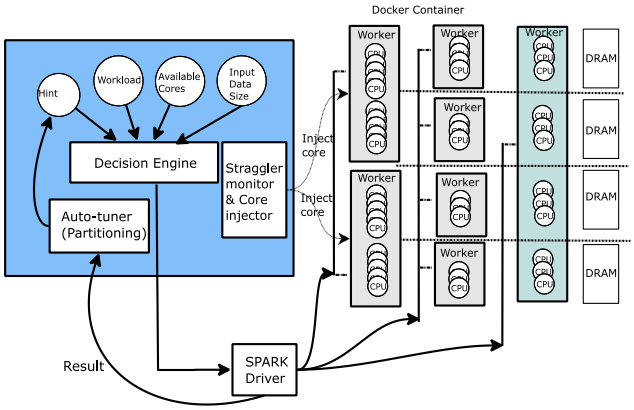


Figure 6 The proof-of-concept architecture

Decision engine is one of the most important features since all partitioning policy come from the 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 contains the straggler tasks.

4. RELATED WORK

Apache Spark Scalability. To improve the Spark scalability, researchers have attempted to optimize for scale-out server [6] [7] or to optimize scale-up server [3] [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 consider 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 [17] [19]. Barrelfish [19] 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 achieve performance scalability by commuting interface [20].

5. 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. Evaluation results (Word Count, Naive Basian, Grep and K-means) reveal that Docker-container method has substantial performance up to 1.7 times compared to existing solutions.

Future Directions. Our future directions are:

• Implementing the proof-of-concept architecture. This paper shows manually partitioning method, so we will implement the Docker-container-based partitioning method.

• Solving the straggler tasks problem. straggler tasks significantly extend job completion times. To mitigate this problem, we may use dynamic resource allocation solution in Docker to maximized CPU utilization.

6. ACKNOWLEDGMENTS

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. B010-16-0644, “Research Project on High Performance and Scalable Manycore Operating System”)

7. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[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] 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.

[6] M. Maas, K. Asanovic, 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.

[7] 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.

[8] 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.

[9] 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.

[10] V. C. Rik van Riel. Automatic NUMA Balancing. https://www.linux-kvm.org/images/7/75/01x07bNumaAutobalancing.pdf.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.