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 garge collection and remote socket access overheads by using an efficient partitining method for a NUMA-based manycore scale-up server. 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 1.2x through 2.2x on a 120 core scale-up system.

CCS Concepts

•**Computer systems organization** → **Embedded systems;** *Redundancy;* Robotics; •**Networks** → Network reliability;

Keywords

ACM proceedings; LaTeX; text tagging

1. INTRODUCTION

Popular big data analytics infrastructures(e.g, Spark[], Hadoop[]) 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[HPC]. Moreover, hardware trends are beginning to change to the scale-up server; a scale-up server can now have substantial CPU, memory, and storage I/O resources[rethink]. Therefore, the big data analytics infrastructure on scale-up server is also importance.

Apache Spark is one of thet widely used big data analytics framework. However, Apache Saprk has been reported that it does not scale on the single node scale-up server because of garbage collection(GC)[][][] and remote memory access latency[][][]. A.J. Awan et al. analysed GC time and compared state of the art garbage collectors, changing GC can not solve the scalability problem(section 2). Moreover, in order to solve the memory access latency problem, researchs have attempt to NUMA balancing[][], but these method also can not satisfactory compared to partitioning approch(see section 2).

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 NUMA latency overheads by applying docker container-based and by using efficieny partitioning to Spark on scale-up server. Our basic key idea is that shared-memory system is dealt with as the distributed-system using partitioning approch in order to eliminate GC and memory access overheads.

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 **??**. Small size cpu groups can mitigate the thread serialized problem cased by GC pause time, and these group may only accees to local NUMA memory. Moreover, partiting method can somewhat reduce the operating systems scalalbility problems(e.g, lock contention[Bonsai, Radix], cache communication overhead[Oplog,FC], single address space problem)

To evaluate our approach, we applied our paritioning method on 120 core scale-up server. A too small size partitioning may reduce GC and NUMA remote access, but the benefits do not come for free because it may cause straggler tasks problem[]. Thus, the 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 1.7x, 1.6x, and 2.2x for a big data workload- , 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 parlel 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 1.5x through 2.7x on 120 core.

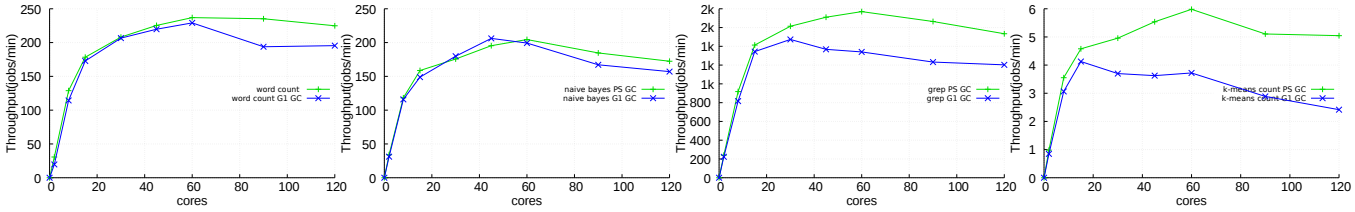
The rest of this paper is organized as follows. Section 2 describes the test-bed and spark scalability problem. Section 3 describes the our partitioning approach and Section 4 shows the results of the experimental evaluation. Section 5 describes related works. Finally, section 6 concludes the paper.

2. SCALE-UP SERVER SCALALBILITY

XXXXX 'XXXX', USA

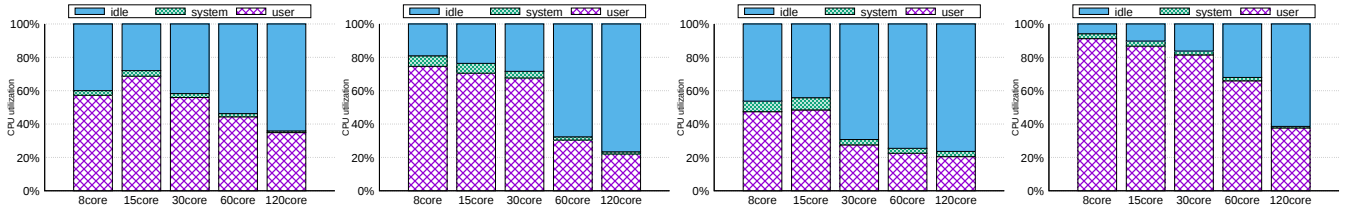
© 2016 ACM. ISBN XX...$15.00

DOI: XX.XXX/XXX_X



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

Figure 1: CPU utilization on 120 core.



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

Figure 2: CPU utilization on 120 core.

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 evits the least recently used RDD. Spark may has a substantial performance when dataset can fit in memory.

Test-bed. Our Intel platform is composed of Xeon E5 Each CPU exploits the Iby Brige technoloy and consists in a multi-core architecture. 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 9) and, to show that our conclusions generalize.

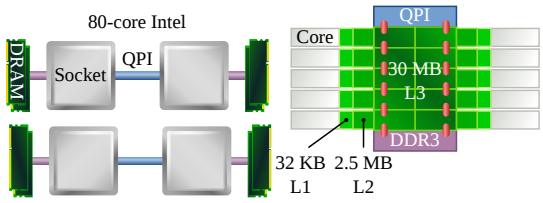


Figure 3: test-bed intel xeon archtecture.

Hyper-Threading is disabled, and Linux kernel 4.5-rc6.

Benchmark. 벤치마크와 워크로드에 대한 BigData Benchmark를 사용하였다.

- **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 G1 and Parallel Scavenge GC(PS). Up to 60 core, The five workloads scales linealy and then GC pause become bottlenecks. The word count flattens out after 60 core, but other benchmarks slightly go down because not only GC overhead but also NUMA effects. To evalute state of the art GC, we used G1 and PS GC. The result of impact of chainging the GC are PS GC results in 3.3x better performance than G1 GC on 120 core. However, although we used the scalable GC, the Spark performance scalabiliy is still limited by GC and NUMA effects.

Figure 2 shows the job's CPU utilization. Our goal is high cpu utilization, Th y-axis is the percentage of time spent in kernel-space code(sys), user-space code(user), and idle time(idle). When core counts increase, all benchmarks increase idle time due to GC pasue that means

2.3 Benefit of JVM Partitioning

Spark and Hadoop ramework use JAVA and it needs java virtual machine, so understanding the partitioning effect on JVM is importance. To preliminarily analysis the JVM partitioning effect, we conducted benchamrk using SPECjbb2013, which a state of the art benchmark for JVM performance. We used two different experiment settings. First, we used per-socket JVM partitioning using the numa control application(numactl). Second, we set maxium available memory to JVM heap 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 all core shows that partitioning approach outperforms than non-partitioning approach by Xx on 120 core. In manycore scale-up server, partiting approach has best performance scalability.

3. PARTITIONING FOR SPARK

The reason for requiring partitioning method is that spark library and runtime engine can be bottlenect by GC and memory

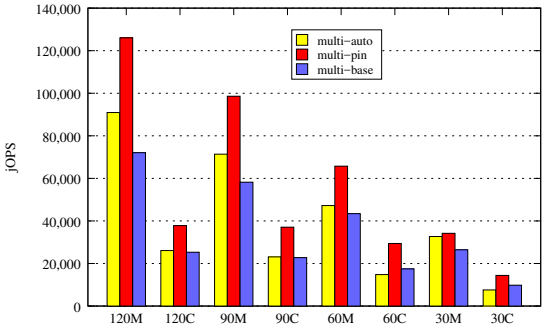


Figure 4: Test-bed Intel xeon archtecture.

latency because Spark have not been considers scale-up environment. Thus, we use the docker container-based partitioning method to eliminate GC and memory latency overheads. This section explains design aspects of our docker container-based partitioning method to solve GC and memory latency.

3.1 Design Consideration

The major problem of Spark scalability is GC, so partitioning approch is needed. Indeed, GC leads to many of the advantages of high-level languages becuase of an increase in productivity, while it is a double-edged sword because GC pauses can cause serilizaed operation and requests to take unacceptable long times. Thus, reducing GC pause time may lead to high performance scalability, and minialized CPU counts can mitigate GC pause time[]. Therefore, the first design consideration of scale partitioning is to minimize GC pause times.

The second design consideration is locality issues because of NUMA archtecture DRAM access latency. For example, threads are scheduled by the OS to execute on any core. When the thread is migrated to different memory area, the thread may accesse remote memory. Partitioning aprroach can prevent to migrate other socket's core. Indeed, the modern operating systems(Linux) has a NUMA balancing feature for enhancement of memory locality, but partitioning method can more superial performance regarding the large scale-up server(8 socket)[](see section 2).

In addition to GC and NUMA effect, operating systems noise can pose scalability bottlenct because it designed for shared-memory system;therefore, the next design consideration is to avoid operating system noise. For example, Single address sapce sharing problem[][][][] between multi-thread applications on JVM, scheduer bottlenct by load balancer[][], and cache comunication bottlenct[][] are major problems in manycore scale-up server operating system. This problem cause by sharing resource, so our approach can solve from these resource contention by using partitiong approch.

To satisfy these factors such as GC, NUMA and operating system bottlenct cause by shared-memory system, Spark on scale-up server should work as distributed system concept. Therefore, we use partitioning approch for shared-memory system and it is dealt with as the distributed-system big data analytics frameworks thereby eliminating GC and memory access overheads. We make shared resource to small size group as much as possible; as result of, the small size cpu groups can mitigate the thread serialized problem cased by GC pause time, and these group may only accees to local NUMA memory.

The final design consideration is straggler tasks problem. If 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[][]. Thus, in order to scale Spark performance scalability, depending on partitioning size.

3.2 Towards a Container-based Framework

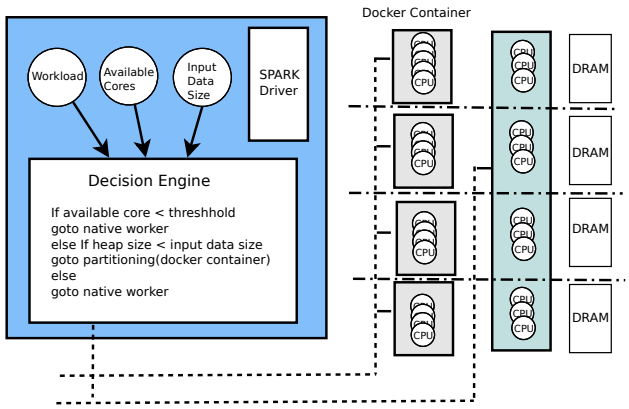


Figure 5: Overview of the docker contrainer-based partitioning

This section describes our vision that will accommodate the previous mentioned design consideration. Our proposed scaleble partitiong 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.

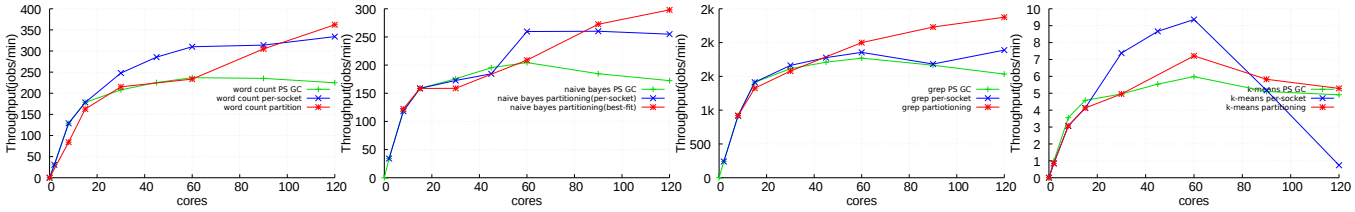
Decision engine is one of the most important features since every partitioning regarding Spark system workers are based on our decision engine compnent. Our simplified algorithm of the decision engine is that if available cores are less than pre-defined thrashhold, then a job works on the native CPUs because Spark system scales leanerly up to low cores(see section 2). If not, the decision engine compares heap size with input data size, then it decides whether or not running on the partitioned docker. If heap size is bigger than input data size, then a job works on the native CPUs since the Spark has a substantial performance scalability when the dataset can fit in memory(see section 2).

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. All experiments were performed on a 120 core machine with 8-socket, 15-core Intel E7-8870 chips equipped with 792 GB DDR3 DRAM. We used ram file system for HDFS due to the eliminating the HDFS bottlenect.

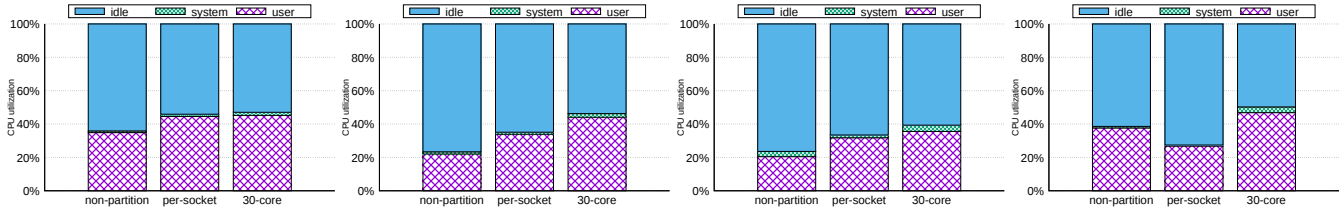
We used four different experiment settings. First, we used non-partitioning method as Figure section 2 graph and we set heap size(4G). Second we used fine-grained partitiong that is per-socket(15 core) partitiong because it can make maximize locality. We allocated heap size by modifing heap size is that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

The results for Word Count are shown in Figure 6(a), and the result shows the thoughput of BigDataBench with our four different settings. Up to 60 core, the PS GC version of non-partiting 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 corese-grained



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

Figure 6: CPU utilization on 120 core.



(a) Word Count (b) Naive Basian (c) Grep (d)

Figure 7: CPU utilization on 120 core.

partitiong outperforms non-partitioning by 1.5x and per-socket partiting by 1.5x on 120 core. The results(Figure **??**(b)) for Naive Basian is similar to Word Count workload. Our per-socket partitining outperform non-partitioning by 1.5x and per-socket partidfa by 1.5. on 120 core.

The results for Word Count are shown in Figure 6(a), and the result shows the thoughput of BigDataBench with our four different settings. Up to 60 core, the PS GC version of non-partiting approach scales linearly and then it flattens out.

The results for Word Count are shown in Figure 6(a), and the result shows the thoughput of BigDataBench with our four different settings. Up to 60 core, the PS GC version of non-partiting 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 corese-grained partitiong outperforms non-partitioning by 1.5x and per-socket partiting by 1.5x on 120 core.

5. RELATED WORK

Apache Spark Scalability. To improve the scalability, researchers have attempted to create new operating systems [2] [20] or have attempted to optimize existing operating systems [3] [6] [7] [4] 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 [2] [20] or have attempted to optimize existing operating systems [3] [6] [7] [4] 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 containner. However, many of sicentiy or big data acaldld may reuse simillar workload, so training phase for finding the best-fit partitiong can be superior solution than manulay partiion simillay way of compiler-based auto-tuner[].

7. REFERENCES

[1] Maya Arbel and Hagit Attiya. Concurrent Updates with RCU: Search Tree As an Example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, 2014.

[2] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, 2008.

[3] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, 2010.

[4] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. OpLog: a library for scaling

update-heavy data structures. In *Technical Report MIT-CSAIL-TR2014-019*, 2014.

[5] Milind Chabbi and John Mellor-Crummey. Contention-conscious, Locality-preserving Locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 22:1–22:14, 2016.

[6] Austin T. Clements, M. Frans Kaashoek, and Nickolai 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.

[7] Austin T. Clements, M. Frans Kaashoek, and Nickolai and 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.

[8] Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A Scalable, Correct Time-Stamped Stack. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 233–246, 2015.

[9] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the Combining Synchronization Technique. *SIGPLAN Not.*, 47(8):257–266, February 2012.

[10] Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, 2001.

[11] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. SPAA '10, pages 355–364, 2010.

[12] David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. Delegation Locking Libraries for Improved Performance of Multithreaded Program. In *Euro-Par 2014 Parallel Processing*, pages 572–583, 2014.

[13] Joohyun Kyong and Sung-Soo Lim. LDU: A Lightweight Concurrent Update Method with Deferred Processing for Linux Kernel Scalability. In *Proceedings of the IASTED International Conference, Parallel and Distributed Computing and Networks (PDCN 2016)*, 2016.

[14] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171. IEEE Computer Society, 1994.

[15] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A Lightweight Synchronization Mechanism for Concurrent Programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 168–183, 2015.

[16] Paul E. McKenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, October 1998.

[17] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[18] Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *DISC '13 Proceedings of the 27th International Conference on Distributed Computing, Jerusalem, Israel*. 2013.

[19] Zoran Radovic and Erik Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 241–. IEEE Computer Society, 2003.

[20] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 3–14, 2010.