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 the garge collection and remote socket access overheads by using efficiently partitining method for NUMA based manycore scale-up server. The proposed docker container-based partitioning method is evaluated using representative benchmark programs, and our evaluation revealed that the docker container-based scalable 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. conducted 이러한 방법 모두 GC가 가지고 있는 근본적인 성능 저하 때문에 여전히 scalability 문제가 있다(section 2).

XXXXX 'XXXX', USA

© 2016 ACM. ISBN XX...$15.00

DOI: XX.XXX/XXX_X

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).

In order to solve the GC problem and the memory access latency probelem that have been a major problem of spark scalability, we present a docker containers-based patitioning method. This method make cpu and memory run with small size of cpu and memory. Therefore, since small size of cpu groups work with shared global data, it can mitigate the thread serialized problem cased by GC. 또한 파티션된어 NUMA 소켓의 로컬 메모리만 접근하므로 memory access latency 줄일 수 있다. 뿐만아니라 이러한 방법은 Scale-up 서버의 shared-memory 시스템에서 사용되는 운영체제의 근본적인 문제인 공유 때문에 발생하는 문제들(e.g, lock contention[Bonsai, Radix], cache communication overhead[Oplog,FC], single address space problem)을 함께 제거할 수 있다. 우리는 docker container를 이용하여 파티션닝 기법을 구현하였고, 그 결과 shared-memory 시스템을 마치 distributed-system 처럼 구성하여 scalability를 향상 시켰다.

또한 문제를 해결하기 위해, 본 논문은 최고의 성능 확장서을 내기 위한 파티션 방법에 대해서 설명한다. 그 이유는 만약 너무 작게 파티션을 한다면 전형적인 scale-out 시스템의 문제점인 straggler tasks 때문에 job completion 시간이 연기가 되어 성능에 문제가 있고, 너무 크게 파티션을 한다면 GC와 memory latency 문제가 발생하기 때문이다. 우리의 효율적인 파티션 방법의 평가는 evaluation of the proposed apporch 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:

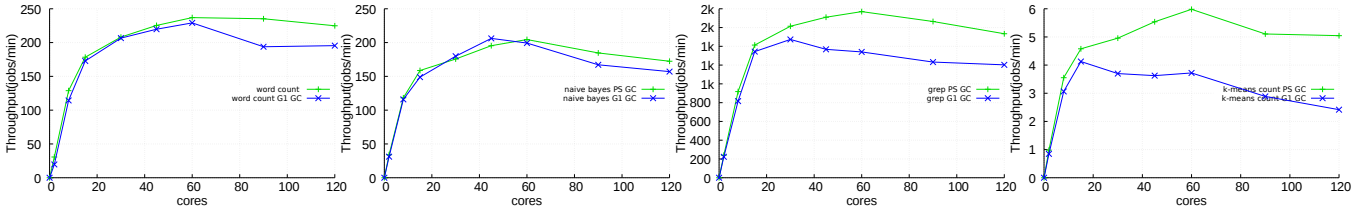
- We analyzed Apache Spark performance scalability on 120 core scale-up server. The results of scalability was that parlel GC can improve performance scalability up to 60 core, but then the GC flattens out after 60 core.
- We applied our docker container-based partitioning method to scale-up server thereby solving scalability problems in BigDataBench. Our design 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

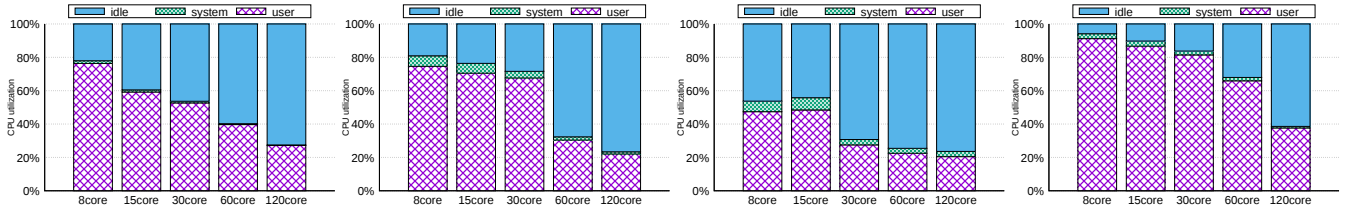
Figure 6(a)) since it waits to acquire

본 장에서는 single node로 구성된 scale-up 서버에서의 spark scalability에 분석한 내용에 대해 설명한다.



(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 open-source data analytics frameworks, designed to operate on datasets larger than can be process on a single node while automatically providing for scheduling and load-balancing.

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 two machines to evaluate conflict-free and conflicting operations on real hardware: an 80-core (8 sockets × 10 cores) Intel Xeon E7-8870 (the same machine used for evaluation in chapter 9) and, to show that our conclusions generalize, a 48-core (8 sockets × 6 cores) AMD the two manufacturers use different architectures, interconnects, and coherence protocols.

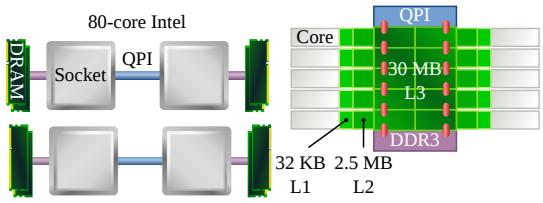


Figure 3: test-bed intel xeon archtecture.

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 workload. 실험 결과 60코어까지는, scalability를 가지다가 60코어 이후에는 scalability 문제가 생겼다. 그 이유는 GC에 의해 serialized 되는 부분과 NUMA에 대한 영향 때문에 scalability가 떨어진다. Word Count와 XXXX XXXX는 비슷한 결과를 가진다. 하지만 Pagerank는 다른 결과를 가지는데 그 이유는 XX Shared memory 시스템의 공유데이터 때문에 발생하는 scalability 저해 요소 때문에 필요하다. 첫째로 공유 데이터를 lock이 있다. 표 xxx 앞에서 실험한 spark의 wordcount에 대해서 .

우리는 scalable한 GC를 2가지를 대상으로 실험을 해보았다. 실험 결과 GC의 종류에 따른 성능차이는 크다. 먼저 GxXXX GC는 가장 높은 성능을 가졌으며, 다른 GC들은 이것보다는 낮은 성능을 보였다. 하지만 아무리 Pararall한 GC를 사용한다고 해도 여전히 scalability 문제는 존재한다. JVM 위에서 동작하는 thread간의 공유하는 single address space때문에 발생하는 공유 문제이다. 다음으로 scheduler가 아직

CPU utilization은 그림 XX-XX와 같다. node에 동작하는 시스템의 scalability 특성을 고려하지 않았기 때문이다. scale-up server를 위한 spark scalability의 근본적인 해결 방법은 spark library와 runtime엔진을 scale-up서버를 위해 scalable하게 만드는 것이다. 하지만 scale-out 시스템의 scalability를 위해 작성된 spark의 library와 runtime 엔진을 수정하는것은 쉽지않다.

2.3 Benefit of Partitioning

Spark의 모든 Thread들은 모두 JVM 위에서 동작하기 때문에, Spark는 JVM에 의존적이다. 따라서 우리는 JVM의 파티션닝에 대한 효과를 분석하기 위해, JVM에서 동작하는 응용프로그램의 성능을 가장 잘 practical하게 측정 할 수 있는 벤치마크인 SPECjbb2013 사용하여 사전에 파티션닝에 대한 효과를 분석을 하였다. 파티션닝된 실험은 SPECjbb2013 벤치마크의 JVM 수를 소캣의 갯수와 동일하게 설정하였고, 메모리 용량을 시스템 최대 가용 용량으로 설정하여 실험하였다. 실험은 다음과 같이 3가지 설정을 하여 수행하였다. 자동 NUMA 밸런싱 기능을 사용한 방법(multi-auto)과 사용하지 않은 방법(multi-base) 그리고 그룹을 노드에 최적화한 방법(multi-pin)을 사용하여 실험하였다.

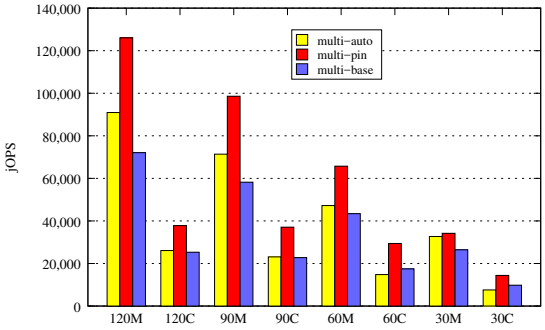


Figure 4: Test-bed Intel xeon archtecture.

실험 결과 대부분 응용프로그램에서 최적화된 파티션으로 설정한 방법이 가장 좋은 성능을 보였고, 다음으로 자동 NUMA 밸런싱이 좋은 성능을 나타낸다. 마지막으로 자동 NUMA 밸런싱을 사용하지 않으면 최악의 성능을 보였다. 이것은 아직까지는 리눅스 커널에 autoNUMA의 기술이 향상 되고 있어도, 잘 파티션되어 동작할 때와 많은 성능 차이를 보여준다[].

3. PARTITIONING FOR SPARK

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

파티션닝 방법이 필요한 가장 큰 이유는 GC의 영향 때문이다. GC가 발생하면 모든 Thread는 멈추고 동작하는데, pararrell GC를 사용해도, 여전히 GC가 완벽하게 parrall하게 동작하지는 않기 때문에, 여전히 serializaed 되는 부분이 존재한다. 하지만, JVM이 동작하는 CPU 수가 적을 수록 GC의 영향은 적게 받는다[]. 본 논문은 이러한 점을 이용하여 JVM을 나누어 동작시키도록 하여 GC의 영향을 최소화 하도록 하였다.

파티션닝 방법이 필요한 두번째 이유는 DRAM access latency 때문이다. 만약 scale-up server가 NUMA 아키텍쳐를 가진 경우일 경우, 원격 메모리 access에 따른 access latency가 떨어지기 때문에, 성능을 저하시킨다. 리눅스는 이러한 문제를 해결하기 위해 커널 내부에 automatic NUMA balancing이라는 기능이 있으나, 아직 파티션되어 수행하는 방법보다는 성능이 떨어진다[].

NUMA의 영향 뿐만 아니라, operating system의 저해 요소 때문에 파티션닝 방법은 필요하다. 즉 Shared memory 시스템의 공유데이터 때문에 발생하는 scalability 저해 요소 때문에 필요하다. 첫째로 공유 데이터에 대한 lock이 있다. JVM 위에서 동작하는 thread간의 공유하는 single address space때문에 발생하는 공유 문제[][][][]이다. 다음으로 scheduler에 대한 문제도 여전히 존재한다. 리눅스의 load balancer가 동작할 때는 serialized되기 때문에 lock이 걸릴 수 밖에 없다[][]. 마지막으로 공유 데이터 때문에 발생하는 cache cohearci traffic[][][]에 대한 문제도 제거할 수 있다.

이처럼 GC, NUMA 그리고 shared memory의 공유 데이터 때문에 발생하는 operating system의 scalability 저해 요소 때문에, scale-up 서버를 위한 스파크도 distributed system의 개념처럼 동작해야한다. 따라서 본 연구는 메모리와 CPU를 적은 수의 CPU와 메모리로 파티션닝을 하여 마치 shared memory 시스템을 distributed system 처럼 동작하도록 제안 한다. 스파크 워커들은 모두 독립적인 cpu와 memory를 할당받아 최대한 thread간의 공유메모리와 remote memory에 접근을 막도록 하였다.

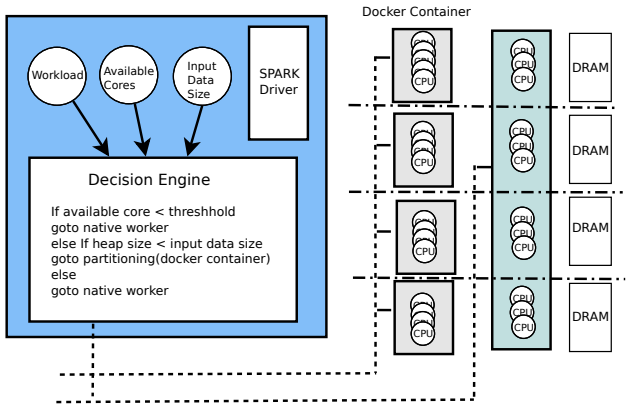


Figure 5: The example showing seven update operations(insert C, insert D , reader executes with lock.

하지만 너무 작은 파티션닝은 또 다른 문제를 낳는다.

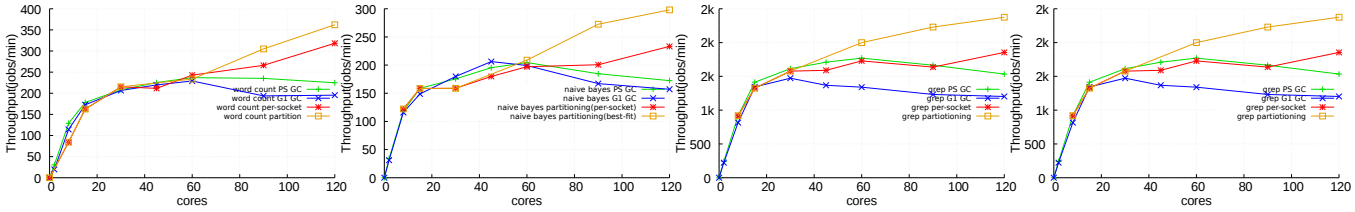
앞에서 설명한 디자인적인 요소를 고려하여 만든 본 연구에서 제안하는 파티션닝 기반의 구조는 그림 <x>와 같다. 그림의 좌측은 우리가 제안하는 방법에 대한 프레임 워크를 보여주며, 오른쪽은 파티션된 docker container들과 native로 동작하기 위한 구조를 보여주며, 마지막으로 소켓단위의 DRAM을 보여준다. 워크로드, 가용한 CPU 수 그리고 데이터 사이즈를 기반으로 파티션된 도커에서 수행할 지 아니면 native로 수행할 지 판단한다. 만약 사용 가능한 코어수가 작으면 SPARK는 적은 코어에서는 그림 <x-x>와 같이 Scalabe하므로 파티션된 도커 컨테이너에서 수행하지 않고, 그냥 Native로 수행한다. 그렇지 않다면, 데이터 사이즈가 설정된 heap 사이즈와 비교 후 Native로 실행 할 지 Docker에서 수행할지 판단한다. 만약 데이터 사이즈가 작은 경우는 native역시 scalable하므로 native 워커로 동작을 시킨다. 만약 데이터 사이즈가 큰 경우에는 파티션된 도커 워커에서 수행하는 것이 GC와 NUMA의 영향과 operating system의 scalabilby 저해요소를 제거 할 수 있으므로 파티션닝 기법으로 수행한다.

우리는 Worker들의 파티션닝을 위해 Docker container를 사용하였다. 그 이유는 virtual machine보다 훨씬 가벼운 구조로 되어 있으며, 최근 docker기반으로 시스템을 관리하는 구조로 변경되고 있기 때문이다. 예를 들어 google kubernet을 사용할 경우, 워크마다 가장 최적의 파티션닝 값을 구한 후 그에 맞는 도커 컨테이너를 실행시키는 방법으로 구성하면 된다. 따라서 본 연구의 파티셔닝을 위한 방법으로 Docker container를 사용하였다.

4. EVALUATION

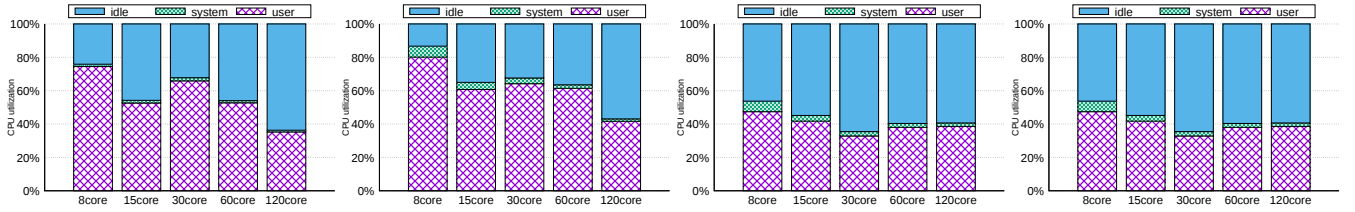
이번 장에서는 스파크의 Scalalbility에 대해서 설명한다. 실험 환경은 2장에서 수행한 방법과 같은 플랫폼에서 실험을 하였고, 아직 앞장에서 설명한 disition engine에 대해서는 구현하지 못하여, 수동으로 도커로 파티션하여 동작하도록 실험을 하였다. We ran the three benchmarks on Linux 3.19.rc4 with stock Linux. All experiments were performed on a 120 core machine with 8-socket, 15-core Intel E7-8870 chips equipped with 792 GB DDR3 DRAM.

이번 장에서는 스파크의 Scalalbility에 대해서 설명한다. 첫째로 fine-graind partitioning을 방법인 NUMA socket 단위로 파티션닝을 수행한 방법이다. 우리의 시스템은 한 소켓당 15코어를 가지는 NUMA machine이므로 15개의 코어를 가지는 도커를 8개를 생성하여 동작시켰다. 둘째로는 낙오자된 태스크



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

Figure 6: CPU utilization on 120 core.



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

Figure 7: CPU utilization on 120 core.

때문에 발생하는 문제를 제거하기 위해, 보다 corese-graind한 파티션 Heap 메모리 설정은 표와 같다. 기본으로 4G의 메모리를 heap 메모리 사이즈로 설정하였고, input 데이터는 10G 이상으로 설정하였다. 데이터를 사용하는 big data 영역에서는 heap 메모리 사이즈가 데이터 사이즈보다 일반적으로 크기 때문이다. 그리고 메모리 설정은 파티션을 수행한 경우는 각 도커에 512MB로 설정하여 메모리 사용을 동일하게 하여 동작시켰다.

그림 X-x는 실험 결과를 보여 준다. 대부분이 비슷한 양상을 보였고, 60코어 이하까지는 파티션한것과 Native와는 비슷한 결과를 가졌다. 하지만 60코어 이상이 되는 순간 도커 파티션닝 기법의 성능이 더 좋아지며, 120코어에서는 X배의 성능 향상을 이루었다. 그 이유는 파티션된 도커 워커에서 수행하는 것이 GC와 NUMA의 영향과 operating system의 scalabilby 저해요소를 제거 했기 때문에 Native로 수행한 것보다 높은 성능을 보였다.

AIM7 forks many processes, each of which concurrently runs. We used AIM7-multiuser, which is one of workload in AIM7. The multiuser workload is composed of various workloads such as disk-file operations, process creation, virtual memory operations, pipe I/O, and arithmetic operation. To minimize IO bottlenecks, the workload was executed with tmpfs filesystems, each of which is 10 GB.

AIM7 forks many processes, each of which concurrently runs. We used AIM7-multiuser, which is one of workload in AIM7. The multiuser workload is composed of various workloads such as disk-file operations, process creation, virtual memory operations, pipe I/O, and

AIM7 forks many processes, each of which concurrently runs. We used AIM7-multiuser, which is one of workload in AIM7. The multiuser workload is composed of various workloads such as disk-file operations, process creation, virtual memory operations, pipe I/O, and

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.

Spark Garbage Collect Problems. In order to improve Li 멀티커널.등등 공유메모리를 사용 X Many scalable data structures with scalable schemes show different performances depending on their update ratios. In low and middle update rate, researchers have attempted to create new scalable schemes [16] [15] [10] or have attempted to adapt these scheme to data structures [1] [8] [6]. In high update rate, the OpLog shows significant improvement in performance scalability for update-heavy data structures in many core systems, but suffers from limitation and overhead due to time-stamp counter management. We substantially extend our preliminary work [13] not only to support per-core algorithm but also to apply the to anonymous rmap due to improving the Linux kernel scalability.

Manycore Scalability. OS scalability.원천기술. Scalable locks have been designed by the queue-based locks [17] [14], hierarchical locks [19] [5] and delegation techniques [11] [9] [12]. Our research is similar to the delegation techniques because the s `synchronize` function runs as a combiner thread;it improves cache locality. However, our approach not only can improve cache locality but also can eliminate synchronization methods during updates due to using a lock-free manner.

6. CONCLUSION

We propose a concurrent update algorithm, , for update-heavy data structures scalable for many-core systems. To achieve the scalability during process spawning, we applied deferred log processing with global log queue and update-side absorbing to Linux reverse mapping. Evaluation results using the AIM7, Exim and lmbench reveal that LDU shows better performance up to 2.2 times compared to existing solutions. LDU is implemented on to Linux kernel 3.19 and available as open-source from

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.