The Case for Tiny Tasks in Compute Clusters

Kay Ousterhout*, Aurojit Panda*, Joshua Rosen*, Shivaram Venkataraman*, Reynold Xin*,
Sylvia Ratnasamy*, Scott Shenker*†, Ion Stoica*
*UC Berkeley, † ICSI

To see the world in a grain of sand...
– William Blake

Abstract

We argue for breaking data-parallel jobs in compute clusters into *tiny tasks* that each complete in hundreds of milliseconds. Tiny tasks avoid the need for complex skew mitigation techniques: by breaking a large job into millions of tiny tasks, work will be evenly spread over available resources by the scheduler. Furthermore, tiny tasks alleviate long wait times seen in today's clusters for interactive jobs: even large batch jobs can be split into small tasks that finish quickly. We demonstrate a 5.2x improvement in response times due to the use of smaller tasks.

In current data-parallel computing frameworks, high task launch overheads and scalability limitations prevent users from running short tasks. Recent research has addressed many of these bottlenecks; we discuss remaining challenges and propose a task execution framework that can efficiently support tiny tasks.

1 Introduction

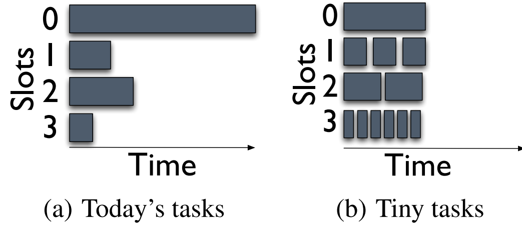
Cluster computing has become widespread, leading to a proliferation of research on improving performance for data-parallel computations. Researchers have attempted to tackle numerous problems that arise in this setting including unfairness [3, 18, 19, 36], stragglers [3, 4], and skew [1, 17, 20]. Reducing task granularity alleviates all of these problems, yet surprisingly has not been explored in this context. Historically, task launch overheads have prevented the use of smaller tasks, but recent improvements in distributed file systems and scheduling eliminate scaling bottlenecks. Thus, we argue for breaking all jobs into *tiny tasks*, which offers the following benefits:

Batch and interactive sharing: Current clusters are required to trade off utilization and responsiveness. If a cluster is highly utilized, an interactive job may need to wait for long-running batch tasks to complete before it can be serviced; reserving slots for interactive jobs avoids this problem but results in lower utilization. Tiny tasks allow a cluster to be *both* responsive and highly utilized, since small tasks ensure frequent opportunities for new, interactive jobs to be launched.

Straggler mitigation: Prior work has shown that job runtimes are largely determined by stragglers: tasks that take much longer to complete than other tasks in the job. Tiny tasks alleviate the straggler problem because work is allocated to machines at a fine granularity, so work is evenly spread over available resources by the task scheduler, and slower machines are assigned less work. Simulations based on a Facebook workload demonstrate that by mitigating stragglers, tiny tasks can improve response times by as much as a factor of 5.2.

Using smaller tasks offers performance improvements even in today's frameworks. However, we propose task durations of at most hundreds of milliseconds across all jobs, which cannot be supported without addressing numerous challenges. First, small tasks require a highly scalable scheduler that can make frequent scheduling decisions. Second, task launch overheads must be small enough so as not to counteract the benefits from using small tasks. Third, tiny tasks must operate on correspondingly tiny amounts of data, which requires a file system that can handle a large number of small file blocks. Finally, tiny tasks require modifications to current programming models to allow *all* jobs to be split into tiny tasks. We propose an architecture that addresses these challenges using a distributed scheduler, and an execution model that gives the execution framework control over I/O. Our proposed design supports 50 microsecond task launches, and allows most applications to be expressed in terms of a set of tiny tasks.

We begin by quantifying the benefits of tiny tasks in §2. We present a preliminary system design that addresses the challenges of supporting tiny tasks in §3, and explore design alternatives and related work in §4.



(a) Today's tasks (b) Tiny tasks

Figure 1: Tasks for a single job in a 4-slot cluster. With tiny tasks, work is allocated to machines at fine time-granularity, mitigating the effect of stragglers and allowing the job to complete more quickly.

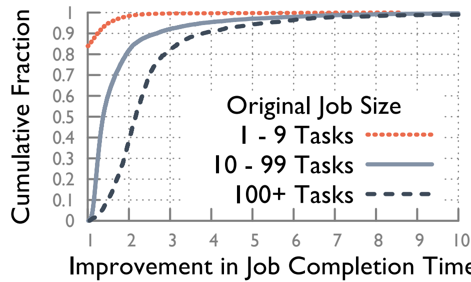


Figure 2: Improvement from perfectly balancing the total machine time for each job across its slots. Jobs that originally had more tasks see a more substantial improvement because having a large number of tasks increases the likelihood that the job includes stragglers.

2 Benefits of Tiny Tasks

Tiny tasks benefit datacenter workloads by reducing scheduling quanta. Using a trace-driven simulation and experiments with Spark [35], we demonstrate that tiny tasks can improve job response times by as much as a factor of 5.2.

2.1 Handling of Skew and Stragglers

Prior studies [4, 36] have noted that job response times in data parallel workloads tend to be dominated by straggler tasks that take much longer than other tasks in the job to complete. These outliers occur for one of two reasons. First, outliers may be caused by poorly performing machines that cause tasks scheduled on them to take longer; e.g., due to malfunctioning disks, contended CPUs, or congested networks. Second, work may have been unevenly divided across tasks, either due to partitioning skew, where data was unevenly allocated to tasks, or due to computational skew, where some data is more expensive to process.

Tiny tasks transform both the slow resources and the data skew problem into a scheduling problem. Rather than needing to predict which resources are slow or statically partition data across tasks, a job is divided into thousands or millions of sub-second tiny tasks, and each task is scheduled as resources become available (shown in Figure 1). In this manner, work is automatically distributed evenly over available resources, without requiring complex skew mitigation techniques: if a machine runs a computationally expensive task, it will simply be assigned fewer total tasks. Similarly, slow resources will automatically be assigned less of a job's work, without needing to predict which machines will perform poorly.

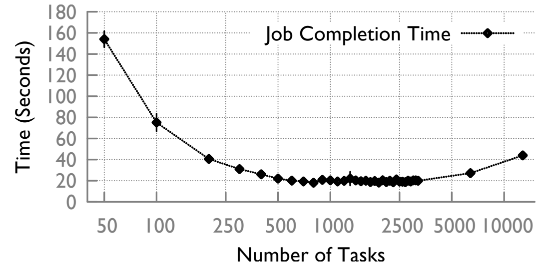
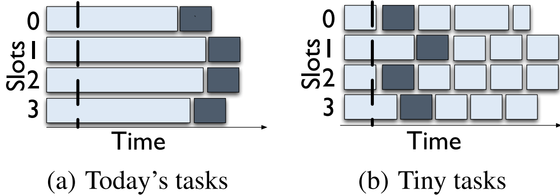


Figure 3: Completion time for a Spark job that is split into various numbers of tasks. The job is executed in a cluster where 20% of machines take 20x longer to run each task. Error bars depict standard deviation.

We use a 54,976 job Facebook trace [8] to quantify how much job response time would improve if work were perfectly partitioned across machines. For each job in the trace, we ask how long the job would have taken if the total time for all tasks in the job were perfectly divided over the slots used by the job. Figure 2 compares this bin packed completion time to the job's original completion time. Evenly balancing work improves performance by 2.2x and 5.2x at the median and 95th percentile, respectively, for jobs that originally had 100 or more tasks. This experiment provides a conservative bound on the speedup for two reasons. First, jobs may have been using fewer than their fair share of slots in the original trace, simply because there were not enough tasks to occupy more slots; in this case, tiny tasks will further improve performance by increasing parallelism. Second, if a task is running on a slow machine, it may take less total time when broken into tiny tasks, because some of the tiny tasks will be run on faster machines. This experiment underestimates the possible improvements from evenly balancing work but ignores task launch overheads, which we address in the next experiment.

To demonstrate that using tiny tasks improves performance in the presence of slow machines, we modified some machines in a cluster to run more slowly, and then ran a Spark [35] job using different numbers of tasks. We used 50 m1.medium EC2 instances, 10 of which were modified to take 20x longer to run each task. Figure 3 demonstrates that using a larger number of tasks improves response time by 8x compared to using the same number of tasks as machines in the cluster, because the slow machines are assigned fewer tasks. Beyond 2000 tasks, task launch overheads cause increased response time; we discuss how to decrease overheads in §3.3.



(a) Today's tasks (b) Tiny tasks

Figure 4: A high priority dark job arrives (at the dashed line) in a cluster where all slots are in use by a low priority light job. With today's tasks, tasks for the high-priority job must wait for long-running tasks from the light job to complete before being launched. With tiny tasks, resource allocation is fine-grained, so resources will quickly be allocated to the higher priority job.

2.2 Improved Sharing

Today, sharing a cluster between interactive and batch jobs involves trading off responsiveness and utilization. If a cluster is highly utilized, an interactive job may need to wait for long-running batch tasks to complete before it can be serviced; reserving slots for interactive jobs avoids this problem but results in lower utilization. Using tiny tasks for all jobs avoids this trade off: the cluster can run at high utilization, while simultaneously guaranteeing that interactive jobs will only need to wait for a short time before being serviced. Figure 4 depicts a simple example with only two jobs, and demonstrates that with tiny tasks, a newly arriving job can quickly obtain resources, even in the presence of batch jobs.

3 Architecting for Tiny Tasks

While existing frameworks can benefit from the use of smaller tasks (as shown in Figure 3), supporting tiny tasks for all jobs in a large cluster requires addressing numerous challenges. A cluster supporting tiny tasks must provide low task launch overheads and use a highly scalable scheduler that can handle hundreds of thousands of scheduling decisions per second. Tiny tasks operate on small blocks of data, and hence require a scalable file system. To ensure that tasks can complete quickly, we propose giving the framework more control over I/O. Finally, ensuring that all jobs can be broken into tiny tasks requires some improvements to the programming model; e.g., support for framework-managed temporary storage. In this section, we discuss a preliminary architecture to address each of these challenges and the lower bound that each challenge places on response time; we aim for tasks that are as small as possible. We find that tasks that complete in hundreds of milliseconds are practical in the short-term, and that we can drive task launch overhead down to further reduce task runtime in the future.

3.1 Execution Model

We propose a task execution model that supports data-parallel computations expressed using a variety of popular programming frameworks (e.g., MapReduce [13], Spark [35], DryadLINQ [32]). A job is composed of a number of tasks, each representing an atomic and idempotent unit of execution. A task consists of a set of named inputs and code that operates on these inputs, and each task runs on a single machine. We assume a cooperative scheduling model; i.e., tasks explicitly release resources on completion (in contrast to preemptive models, discussed in §4.1).

3.2 Scalable Storage Systems

In the short term, we expect the time needed to read input data to be the limiting factor in driving down task durations. Previous work has shown that 8MB random disk reads can achieve approximately 88% of the throughput of sequential reads, and that smaller random reads cause significant performance degradation due to disk seeks [23]. Thus, as long as input data is stored on disk, we require that tasks read at least 8MB of input data. Data-parallel workloads are commonly I/O bound, so we expect task runtime to be dominated by time taken to read input data; thus, assuming 100MB/s disk throughput, 8MB input data sizes should result in task durations of hundreds of milliseconds.

Using small data blocks requires a move away from traditional distributed file systems like HDFS, where scalability limitations prevent the use of small blocks. While HDFS does allow tasks to read only part of a block, having multiple tiny tasks that operate on the same file block limits parallelism. Recent work on distributed filesystems, e.g., Flat Data Center Storage (FDS) [23], addresses these scalability concerns by distributing metadata across multiple servers. As discussed in §3.4, a framework built for tiny tasks can improve on FDS performance by more closely integrating the file system and the task scheduler.

3.3 Low-Latency Scheduling

Supporting tiny tasks requires a low-latency, high throughput task scheduler. Handling 100ms tasks in a cluster with $160,000$ cores (e.g., 10,000 16-core machines), requires a scheduler that can, on average, make 1.6 million scheduling decisions per second. Today's centralized schedulers have well-known scalability limits [27] that hinder their ability to support tiny tasks in a large cluster. Engineering improvements such as compressing task descriptions, avoiding sending the same task description to the same machine multiple times, and using more efficient networking have helped some centralized schedulers provide higher throughput [33]. However, handling large clusters and very short tasks

will require a decentralized scheduler design. Recently proposed distributed schedulers (e.g., Sparrow [24]) can scale well beyond millions of decisions per second while providing near-optimal response times. Our proposed system relies on the use of such distributed schedulers.

In addition to providing high throughput scheduling decisions, a framework for tiny tasks must also reduce the overhead for launching individual tasks. Popular frameworks like Hadoop MapReduce have task launch overheads of many seconds, due to a variety of factors including the need to launch a new JVM for each task; newer frameworks like Spark [35] reduce the overhead to 5ms. To support tasks that complete in hundreds of milliseconds, we argue for reducing task launch overhead even further to 1ms so that launch overhead constitutes at most 1% of task runtime. By maintaining an active thread pool for task execution on each worker node and caching binaries, task launch overhead can be reduced to the time to make a remote procedure call to the slave machine to launch the task. Today's datacenter networks easily allow a RPC to complete within 1ms. In fact, recent work showed that $10\mu s$ RPCs are possible in the short term [26]; thus, with careful engineering, we believe task launch overheads of $50\mu s$ are attainable. $50\mu s$ task launch overheads would enable even smaller tasks that could read data from in-memory or from flash storage in order to complete in milliseconds.

3.4 Framework-Controlled I/O

Using tiny tasks fundamentally changes the resource footprint of tasks, giving the framework more control to optimize I/O. Today's large tasks accumulate a large amount of output data in memory. Often this output data will exceed available memory and spill onto disk, leading to poor MapReduce performance [21]. Tiny tasks fundamentally change the task resource footprint: since tasks run for a shorter period of time, they generate less output data and thus use less memory. The framework can explicitly control the remaining memory, caching the most important data and storing remaining data on disk or on a different machine. For MapReduce-style jobs, the framework could store the map outputs that will be used for the first set of reduce tasks in memory, and store remaining outputs on disk. While the first set of tiny reduce tasks are running, the framework can pipeline reading data for the next set of tasks. This approach considers file system scheduling, network scheduling, task scheduling, and caching holistically. The benefits of a more holistic approach to scheduling have been illustrated in more restricted settings (e.g., in the context of network scheduling [9, 10]), but validating our broader approach remains for future work.

3.5 Programming Model

Most tasks in a data parallel framework can be split into tiny tasks by reducing the input size; however, some types of tasks cannot easily be parallelized. Parallelizing all jobs is the most significant challenge in realizing tiny tasks. Consider, for example, reduce tasks in a MapReduce job. In the limit, one reduce task can be launched to handle each key. However, if all values map to the same key, that key cannot easily be split into multiple tasks. When the reduce function is associative and commutative, such tasks can be parallelized using techniques like map-side combiners or aggregation trees. These techniques are already used by existing frameworks [32], and will allow most jobs to be split into tiny tasks.

Despite the use of aggregation trees, some tasks may remain difficult to divide into tiny tasks. To split these tasks, we propose providing a framework-managed temporary state store that can be used to communicate and share data between a job's tiny tasks. For instance, to implement a job that computes distinct values, we could store hashes of all values seen so far in the state store. We envision that this store would have a key-value interface and provide strong consistency guarantees.

Inevitably, some tasks will be impractical to split, despite these tools. To accommodate such tasks, we plan on allowing some large tasks to run on the cluster. We expect that if a small percentage of tasks remain large, they can be run on the same infrastructure as tiny tasks without impacting the performance of remaining tasks. Exploring the impact of such sharing is the subject of ongoing research.

4 Alternate Designs and Related Work

Tiny tasks solve two major problems: stragglers, and sharing a cluster between batch and interactive jobs. A variety of approaches solve one of the two problems in isolation; e.g., skew handling techniques mitigate stragglers, and process migration allows improved sharing between long and short jobs.

4.1 Preemption and Process Migration

Our choice of a cooperative multi-tasking scheme with tiny tasks contrasts with preemption based schemes commonly used in operating systems. Compared to tiny tasks, one advantage of using preemption to guarantee sharing properties is that the system can tightly control scheduling quanta [28, 30]; however, preemption has other disadvantages:

Cost of task-switching: Prior work has proposed mechanisms to migrate processes [14, 22], virtual machines [11], and services [25] across machines. Migrating tasks involves transferring inputs, context, and inter-

mediate data; for data parallel applications, input data and intermediate data can be several gigabytes, incurring high migration overhead.

Fault tolerance: If a long task fails, it must be re-executed from the beginning. Periodic task checkpointing can speed recovery, but at great expense [15]. On the other hand, the short duration of tiny tasks limits the amount of lost work after a failure, and tiny tasks can be executed in parallel to speed recovery.

In spite of these drawbacks, some schedulers for data parallel applications use preemption. Quincy [19] kills tasks on preemption, trading wasted work for responsiveness. Amoeba [2] uses preemption to provide improved elasticity in the context of MapReduce-like cluster computing frameworks. Amoeba identifies safe-points when a task can be paused and restarted elsewhere without wasted work. The main drawback of Amoeba is that it does not provide a mechanism for determining safe-points, which is difficult for general tasks (even tasks that use the MapReduce programming model). The Amoeba authors choose preemptability rather than small tasks for two reasons. First, they cite high task launch overheads in systems like Hadoop; as described in §3.5, these overheads are not fundamental and can be solved with improved engineering. Second, they note that creating *uniformly* sized small tasks is difficult. Tasks need not be uniformly sized for the benefits of tiny tasks to hold; rather, tasks must be orders of magnitude smaller than today's tasks.

4.2 Coarse-Grained Resource Allocation

Many clusters use static resource allocation to ensure that long-running jobs do not affect high priority jobs due to head of line blocking [31]. Static partitioning limits utilization because each partition must be provisioned to handle peak load, and extra capacity cannot easily be re-allocated.

Omega [27], Google's cluster scheduler, uses flexible coarse grained resource allocation. Omega shares a cluster across many frameworks that each perform their own task-level scheduling. Tiny tasks can improve performance for a single framework in this context, but performing task-level scheduling separately for each framework limits cluster utilization. Tiny tasks reap the greatest efficiency benefits when used to share resources between multiple users and frameworks.

4.3 Fine-Grained Sharing

The idea of using smaller units of work to improve load balancing is well studied. In multi-threaded applications, work-stealing based schedulers [6] divide work at very fine granularities to provide better load balancing. Smaller units of work have also been used in operating systems [28], storage systems [7, 16] and distributed hash tables [29]. We apply this principle to tasks scheduled in large-scale clusters. The idea of sharing cluster resources at task-level granularity has been used in existing cluster schedulers [18, 34] and prior proposals have also looked at splitting MapReduce tasks [5]. Tiny tasks drive the idea of small tasks to the extreme to enable improved sharing of cluster resources and better responsiveness.

4.4 Skew-Handling

A separate line of research has focused on skew and straggler mitigation. Examples of such work include Mantri [4], SkewTune [20], Scarlett [1], work on task speculation [36], and work on task cloning [3, 12]. Mantri and Scarlett attempt to mitigate task runtime skew by modeling the causes for skew and accounting for these causes when scheduling tasks. In particular, Mantri performs resource aware scheduling to decrease the probability of observing task skews, while Scarlett replicates storage blocks based on their probability to decrease the wait time for a popular block. Both of these systems moderately reduce task skew, but they rely on a fragile set of signals and do not work in all cases. Furthermore, existing straggler mitigation techniques use additional cluster resources to gain more predictable task runtimes. This limits the applicability of these techniques under situations of high load, when achieving predictability may be most important. SkewTune reactively mitigates data skew by splitting large tasks at runtime, while tiny tasks preemptively avoid data skew.

5 Conclusion

Tiny tasks represent a simple design paradigm that mitigates stragglers and allows increased utilization without sacrificing fairness or responsiveness. We have presented an architecture that represents a first step towards realizing tiny tasks. Given the benefits of using smaller tasks, we believe that the systems community should focus on efforts to reduce overheads and provide improved scalability in cluster frameworks.

6 Acknowledgments

We thank Matei Zaharia, Colin Scott, John Ousterhout, and Patrick Wendell for useful feedback on earlier drafts of this paper. This research is supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331; gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!; and a Hertz Foundation Fellowship.

References

[1] Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A., Stoica, I., Harlan, D., and Harris, E. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *Proc. EuroSys* (2011).

[2] Ananthanarayanan, G., Douglas, C., Ramakrishnan, R., Rao, S., and Stoica, I. True Elasticity in Multi-Tenant Data-Intensive Compute Clusters. In *Proc. SOCC* (2012).

[3] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. Effective Straggler Mitigation: Attack of the Clones. In *Proc. NSDI* (2013).

[4] Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., and Harris, E. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. OSDI* (2010).

[5] Bhatotia, P., Wieder, A., Rodrigues, R., Acar, U., and Pasquin, R. Incoop: MapReduce for Incremental Computations. In *Proc. SOCC* (2011).

[6] Blumofe, R., and Leiserson, C. Scheduling Multithreaded Computations by Work Stealing. In *Proc. FOCS* (1994), pp. 356–368.

[7] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. Bigtable: A Distributed Storage System for Structured Data. In *Proc. OSDI* (2006).

[8] Chen, Y., Alspaugh, S., and Katz, R. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. *Proc. of the VLDB Endow. 5*, 12 (2012), 1802–1813.

[9] Chowdhury, M., and Stoica, I. Coflow: A Networking Abstraction for Cluster Applications. In *Proc. HotNets* (2012), pp. 31–36.

[10] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M., and Stoica, I. Managing Data Transfers in Computer Clusters with Orchestra. In *Proc. SIGCOMM* (2011), pp. 98–109.

[11] Clark, C., Fraser, K., Hand, S., Hansen, J., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live Migration of Virtual Machines. In *Proc. NSDI* (2005), vol. 2, pp. 273–286.

[12] Dean, J., and Barroso, L. A. The Tail at Scale. *CACM 56*, 2 (2013), 74–80.

[13] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Proc. OSDI* (2004).

[14] Douglis, F., and Ousterhout, J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software: Practice and Experience 21*, 8 (1991), 757–785.

[15] Dunlap, G., King, S., Cinar, S., Basrai, M., and Chen, P. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proc. OSDI* (2002), pp. 211–224.

[16] Ghemawat, S., Gobioff, H., and Leung, S. The Google File System. In *Proc. SOSP* (2003), pp. 29–43.

[17] Gufler, B., Augsten, N., Reiser, A., and Kemper, A. Load Balancing in MapReduce Based on Scalable Cardinality Estimates. In *Proc. ICDE* (2012), pp. 522–533.

[18] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A., Katz, R., Shenker, S., and Stoica, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proc. NSDI* (2011).

[19] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proc. SOSP* (2009), pp. 261–276.

[20] Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. SkewTune: Mitigating Skew in MapReduce Applications. In *Proc. SIGMOD* (2012), pp. 25–36.

[21] Lipcon, T. Optimizing MapReduce Job Performance, June 2012. `http://www.slideshare.net/cloudera/mr-perf`.

[22] Milojičić, D., Douglis, F., Paindaveine, Y., Wheeler, R., and Zhou, S. Process migration. *CSUR* (2000).

[23] Nightingale, E., Elson, J., Hofmann, O., Suzue, Y., Fan, J., and Howell, J. Flat Datacenter Storage. In *Proc. OSDI* (2012).

[24] Ousterhout, K., Wendell, P., Zaharia, M., and Stoica, I. Sparrow: Scalable scheduling for sub-second parallel jobs. Tech. Rep. UCB/EECS-2013-29, EECS Department, University of California, Berkeley, Apr 2013.

[25] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Lonard, P., and Neuhauser, W. Overview of the CHORUS Distributed Operating Systems. *Computing Systems 1* (1991), 39–69.

[26] Rumble, S. M., Ongaro, D., Stutsman, R., Rosenblum, M., and Ousterhout, J. K. It's Time for Low Latency. In *Proc. HotOS* (2011).

[27] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. Omega: flexible, scalable schedulers for large compute clusters. In *Proc. EuroSys* (2013).

[28] Sherman, S., Baskett III, F., and Browne, J. Trace-Driven Modeling and Analysis of CPU Scheduling in a Multiprogramming System. *CACM 15*, 12 (1972), 1063–1069.

[29] Stoica, I., Morris, R., Karger, D., Kaashoek, M., and Balakrishnan, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. SIGCOMM* (2001), pp. 149–160.

[30] Tanenbaum, A., Van Renesse, R., Van Staveren, H., Sharp, G., and Mullender, S. Experiences with the Amoeba Distributed Operating System. *CACM 33*, 12 (1990), 46–63.

[31] Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R., and Liu, H. Data Warehousing and Analytics Infrastructure at Facebook. In *Proc. SIGMOD* (2010).

[32] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P., and Currey, J. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proc. OSDI* (2008).

[33] Zaharia, M. Developer Meetup: Intro to Spark Internals. `http://files.meetup.com/3138542/dev-meetup-dec-2012.pptx`.

[34] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proc. EuroSys* (2010), pp. 265–278.

[35] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. NSDI* (2011).

[36] Zaharia, M., Konwinski, A., Joseph, A., Katz, R., and Stoica, I. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. OSDI* (2008).