Performance clarity as a first-class design principle

Kay Ousterhout
UC Berkeley

Christopher Canel
Carnegie Mellon University

Max Wolffe
UC Berkeley

Sylvia Ratnasamy
UC Berkeley

Scott Shenker
UC Berkeley, ICSI

ABSTRACT

Users often struggle to reason about the performance of today's systems. Without an understanding of what factors are most important to performance, users do not know how to tune their system's hardware and software configuration to improve performance. We argue that performance clarity – making it easy to understand where bottlenecks lie and the performance implications of various system changes – should be a first class design goal. To illustrate that this is possible, we propose an architecture for data analytics frameworks in which jobs are decomposed into schedulable units called *monotasks* that each consume a single resource. By untangling the use of different resources, using monotasks allows the system to trivially report time used on each resource and the resource bottleneck. Our prototype implementation of monotasks for Apache Spark is API-compatible and achieves performance parity with Spark, and yields a simple performance model that can predict the effects of future hardware and software changes.

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*; *Maintainability and maintenance*; • **Software and its engineering** → *Cloud computing*;

ACM Reference format:
Kay Ousterhout, Christopher Canel, Max Wolffe, Sylvia Ratnasamy, and Scott Shenker. 2017. Performance clarity as a first-class design principle. In *Proceedings of HotOS '17, Whistler, BC, Canada, May 08-10, 2017,* 6 pages.
https://doi.org/10.1145/3102980.3102981

1 INTRODUCTION

Users often spend significant energy trying to understand systems so that they can tune them for better performance. Performance questions that a user might ask include:

What hardware should I run on? Is it worthwhile to get enough memory to cache on-disk data? If I upgrade the network from 1Gbps to 10Gbps, how much will it improve performance?

What software configuration should I use? Should I store compressed or uncompressed data? How much work should be assigned concurrently to each machine?

HotOS '17, May 08-10, 2017, Whistler, BC, Canada

ACM ISBN 978-1-4503-5068-6/17/05…$15.00
https://doi.org/10.1145/3102980.3102981

Why did my workload run so slowly? Is hardware degradation leading to poor performance? Is performance affected by contention from other users?

Effectively answering the above questions can lead to significant performance improvements; for example, Venkataraman et al. demonstrated that selecting an appropriate cloud instance type could improve performance by 1.9× without increasing cost [11]. Yet answering these questions in existing systems remains difficult. Moreover, expending significant effort to answer these questions *once* is not sufficient: users must continuously re-evaluate as software optimizations, hardware improvements, and changes in workload shift the bottleneck.

Existing approaches have added performance visibility as an afterthought, e.g., by adding instrumentation to existing systems [1–3, 9]. We argue that architecting for *performance clarity* – making it easy to understand where bottlenecks lie and the performance implications of various system changes – should be an integral part of system design. Systems that simplify reasoning about performance enable users to determine what configuration parameters to set and what hardware to use to optimize runtime.

To provide performance clarity, we propose building systems in which the basic unit of scheduling consumes only one resource. In the remainder of this paper, we apply this principle to a particular type of system: large-scale data analytics frameworks. We propose decomposing data analytics jobs into *monotasks* that each use exactly one of CPU, disk, and network. Each resource has a dedicated scheduler that schedules the monotasks for that resource. This design contrasts with today's frameworks, which break jobs into tasks that each use fine-grained pipelining to parallelize the use of CPU, disk, and network.

Decoupling the use of different resources into monotasks simplifies reasoning about performance. With current frameworks, tasks use many resources, and resource use may change at fine time granularity during a task's execution. Concurrent tasks may contend for resources, even when their aggregate resource use does not exceed the capacity of the machine; for example, if two CPU-bound tasks issue disk reads at the same time. As a result, reasoning about resource use is difficult, even within a single task. In contrast, each monotask consumes a single resource fully, without blocking on other resources, making it trivial to reason about the resource use of a monotask. Controlling each resource with a dedicated scheduler allows that scheduler to fully utilize the resource and queue monotasks to avoid contention. Explicitly separating different resources makes the bottleneck visible: the bottleneck is simply the resource with the longest queue.

As evidence that using monotasks provides performance clarity, we demonstrate that monotask runtimes can be used to create a simple model for performance. We use the model to predict workload

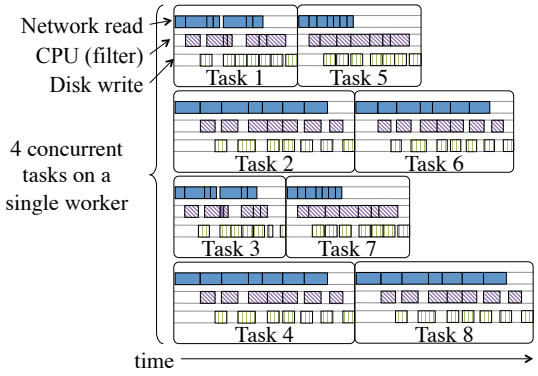

Figure 1: Example set of today's tasks running on a single worker machine. In this job, each task pipelines reading data over the network with filtering some of the data (using the CPU) and writing the result to disk. While each task typically has a dedicated core, tasks may contend for network or disk.

runtime on different hardware and software configurations, and we find that the model provides estimates within 9% of the actual runtime.

Using monotasks improves performance clarity without sacrificing high performance: our prototype implementation provides job completion times comparable to Apache Spark for three benchmark workloads.

The remainder of this paper begins with background about the difficulty of reasoning about performance in current data analytics frameworks (§2) and then proposes a new architecture based on monotasks (§3). We use our prototype implementation of monotasks for Spark to compare performance to Apache Spark (§4) and to predict workload performance on different hardware and software configurations (§5). We end by discussing limitations (§6) and ways in which the performance clarity provided by monotasks can be used to implement new performance optimizations (§7).

2 BACKGROUND

2.1 Architecture of data analytics frameworks

This paper focuses on the design of data analytics frameworks. These frameworks provide an API for users to describe a computation over a distributed dataset. For example, a user might want to add the second and third column in each row of input data, and create a new dataset with the result. The user describes this computation using the framework's API, and the framework handles accessing the distributed dataset and applying the computation to create a new distributed dataset.

Frameworks like MapReduce [4], Dryad [7], and Apache Spark [12] execute jobs on a cluster of machines using a bulk-synchronous-parallel model where each job is broken into stages that are separated by a communication barrier. Each stage is composed of parallel tasks that all perform the same computation, but do so on different blocks of input data. Each task typically uses fine-grained pipelining to parallelize the use of multiple resources, as shown in Figure 1. This pipelining is implemented by the framework: the framework reads a small amount of input data (e.g., from a distributed filesystem) and applies the user-specified computation to that data while pipelining more input reads in the background. When tasks generate output data, the framework similarly pipelines writing the output data with computation and reading input. Pipelining improves performance by utilizing all resources throughout the duration of a task, but requires careful tuning to keep all resources simultaneously utilized.

2.2 The challenge of reasoning about performance

The fine-grained pipelining orchestrated by each task makes reasoning about performance difficult. As an example, consider the challenges to understanding the performance of the tasks in Figure 1.

Tasks have non-uniform resource use: A task's resource profile may change at fine time granularity as different parts of the pipeline become a bottleneck. Task 1 in Figure 1 bottlenecks on the network during the first network read, when it is waiting for data to start processing, but bottlenecks on the CPU during other network reads, and bottlenecks on the disk at the end, when it is waiting to write the last bit of output to disk.

Concurrent tasks on a machine may contend: Each use of network, CPU, or disk may contend with other tasks running on the same machine. For example, some network reads might take longer because they were issued at the same time as requests from other tasks on the machine.

Resource use occurs outside the control of the analytics framework: Resource use is often triggered by the operating system and not the data analytics framework. For example, data written to disk is typically written to the buffer cache. Some disk writes will hit the buffer cache and complete quickly, while others may block while the operating system flushes data to disk.

Together, these three challenges make reasoning about performance difficult. Answering even simple performance questions like "what is the bottleneck for this workload" has required significant instrumentation [9], and no model exists for answering more complicated what-if questions. For example, consider questions like "how much more quickly would my job have run if it hadn't contended with other concurrent jobs" or "how much faster would my job run if twice as many disks were available". To answer these questions, a user would need to walk through a task's execution at the level of detail shown in Figure 1. For each fine-grained network read, CPU use, and disk write, the user would need to determine whether the time for that resource use would change in the new scenario, factoring in how timing would be affected by the resource use of other tasks on the same machine – which would each need to be modeled at similarly fine time granularity. The complexity of this process explains the lack of simple models for job completion time.

This paper explores re-architecting data analytics frameworks with performance clarity as the singular goal. We ask: is it possible to build a system that simplifies reasoning about performance? And, does doing so require sacrificing high performance?

3 A MONOTASKS-BASED ARCHITECTURE

We propose replacing the fine-grained pipelining of today's tasks – henceforth referred to as *multi*tasks – with statistical multiplexing across *mono*tasks that each use a single resource. Our design is based on four principles:

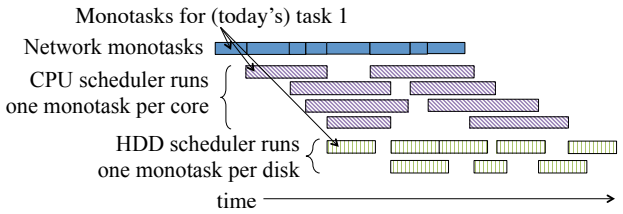

Figure 2: Execution of the eight tasks in Figure 1 as monotasks on a worker machine with four CPU cores and two disks. Each of today's tasks is decomposed into monotasks that each use one of disk, network, and the CPU. Per-resource schedulers regulate access to each resource.

Each task uses one resource: Jobs are decomposed into units of work called monotasks that each use exactly one of CPU, network, and disk. As a result, the resource profile of each task is uniform and predictable.

Monotasks execute in isolation: To ensure that each monotask can fully utilize the underlying resource, monotasks do not interact with or block on other monotasks during their execution.

Per-resource schedulers control contention: Each worker machine has a set of schedulers that are each responsible for scheduling monotasks on one resource. The resource schedulers are designed to run the minimum number of monotasks necessary to keep the underlying resource fully utilized, and queue remaining monotasks. For example, the CPU scheduler runs one monotask per CPU core. This design makes resource contention "visible" as the queue length for each resource.

Per-resource schedulers have complete control over each resource: To ensure that the per-resource schedulers can control contention for each resource, monotasks avoid optimizations that involve the operating system triggering resource use. For example, disk monotasks flush all writes to disk, to avoid situations where the OS buffer cache contends with other disk monotasks.

4 IMPLEMENTATION OF MONOTASKS

We have used monotasks to implement a prototype, MonoSpark, that is API-compatible with Apache Spark: users still write jobs by specifying a computation over a distributed dataset. Instead of pipelining resource use, with MonoSpark, the framework decomposes each of today's multitasks into a directed acyclic graph of single-resource monotasks. Each multitask in the job shown in Figure 1 would be broken into a network monotask (to read input data over the network), followed by a CPU monotask to perform the user-specified computation, followed by a disk monotask to write output data, as shown in Figure 2.

In our implementation, this decomposition is done by the framework and without operating system or hardware support. As a result, monotasks are imperfect: disk and network monotasks use a small amount of CPU as part of performing I/O. In our experiments to date, disk and network monotasks use so little CPU that this does not meaningfully impact performance clarity.

Our prototype uses a simple scheduler with two levels. A framework level scheduler assigns a fixed number of multitasks to worker machines, as in current frameworks. Workers decompose each multitask into monotasks that are executed by the relevant per-resource schedulers. To ensure that each monotask can fully utilize the underlying resource and does not block on the use of other resources, all of a monotask's dependencies must complete before a monotask is launched. For example, the network monotask to read a task's input data will complete before the CPU monotask to compute on that data is launched.

Workload	Spark	MonoSpark
Sort	88 minutes	57 minutes
Matrix multiply	100 seconds	94 seconds

Table 1: Comparison of runtimes with Spark to runtimes with MonoSpark for two benchmark workloads.

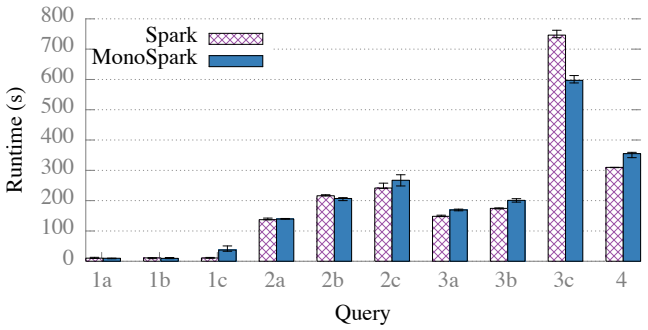

Figure 3: Comparison of runtimes with Spark to runtimes with MonoSpark for the big data benchmark workload.

Using our MonoSpark prototype, we first ask whether using monotasks hurts performance. Next, in §5, we illustrate that using monotasks provides performance clarity by showing how monotask runtimes can be used to construct a simple model for job completion time.

4.1 Does using monotasks hurt performance?

Because using monotasks serializes the resource use of a multitask, the work done by one of today's multitasks will take longer to complete using monotasks. To compensate, MonoSpark relies on the fact that today's jobs are typically broken into many more multitasks than there are slots to run those tasks, and MonoSpark can run more concurrent tasks than today's frameworks. For example, on a worker machine with four CPU cores, Apache Spark would typically run four concurrent tasks: one on each core, as shown in Figure 1. With MonoSpark, on the other hand, four CPU monotasks can run concurrently with a network monotask (to read input data for future CPU monotasks) and with two disk monotasks (to write output), so runtime of a job as a whole is similar using MonoSpark and Spark. In short, using monotasks replaces pipelining with statistical multiplexing, which is robust without requiring careful tuning.

We compared MonoSpark to Spark for three benchmark workloads, and found that MonoSpark provides performance comparable to Spark. MonoSpark is based on Spark and takes the CPU, network, and disk use of Spark tasks as fixed; for example, it uses exactly the same code to compute over input data. Our implementation rearranges the use of these resources to eliminate fine-grained pipelining and instead use monotasks. Table 1 shows that MonoSpark and Spark perform similarly for two benchmark workloads: one that

sorts 600GB of random on-disk data using 20 worker machines that each have two hard disk drives (HDDs), and a second machine learning workload that uses a series of matrix multiplications to perform a least squares fit. Figure 3 compares MonoSpark and Spark for the big data benchmark [10] using a scale factor of five on machines with HDDs. Performance on all three workloads is similar to Spark, even though MonoSpark uses coarser grained pipelining and flushes all disk writes rather than using the buffer cache. In some cases, MonoSpark provides better performance because per-resource schedulers can provide higher utilization and minimize contention.

5 REASONING ABOUT PERFORMANCE

Monotask runtimes can be used to construct a simple model for a job's completion time. This model can be used after a job has run, using information about the job's monotasks, to detect bottlenecks and answer what-if questions about different hardware or software configurations. We focus on designing the simplest model that is sufficiently accurate to evaluate the benefit of hardware or software configuration changes.

We model the runtime of a job using two steps, shown in Figure 4. First, information about the monotasks can be used to compute the ideal time spent running on each resource. For the CPU, the ideal time is the sum of the time for all of the compute monotasks, divided by the number of cores in the cluster. For I/O resources, the ideal resource time can be calculated by dividing the total data transmitted by the throughput of the resource.

At this point, determining the bottleneck resource is trivial: the bottleneck is simply the resource with the largest ideal time. The second step in building the model is to compute the ideal overall completion time, which is the ideal time for the bottleneck resource.

This model is simple and ignores many practicalities, including dependencies between monotasks that might prevent parallelization at some points in the execution. For example, contrary to the simplified execution in Figure 4, the first monotasks to read input data cannot be parallelized with other monotasks, because no data is available to compute on yet. While we could have used information about dependencies between monotasks to construct a more complex model, simpler models are easier to use and apply, and in our experiments to date, this simple model is sufficiently accurate to make coarse-grained predictions.

5.1 Predicting runtime on different hardware

First, we use the model to predict the runtime of a job if it had twice as much disk throughput available. To do this, we divide the ideal disk time by two, and determine the new modeled runtime by taking the new maximum time for any of the resources, as shown on the right side of Figure 4. To compute the new estimated job completion time, we scale the job's original completion time by the change in modeled job completion time. This helps to correct for inaccuracies in the model; e.g., not modeling time when resource use cannot be perfectly parallelized.

Figure 5 illustrates the effectiveness of the model in predicting the runtime on a cluster with twice as many SSDs for three workloads. The workloads all sort key-value pairs, but vary in how I/O bound they are, so the same hardware change results in different performance improvements. The figure shows the runtime on a cluster with

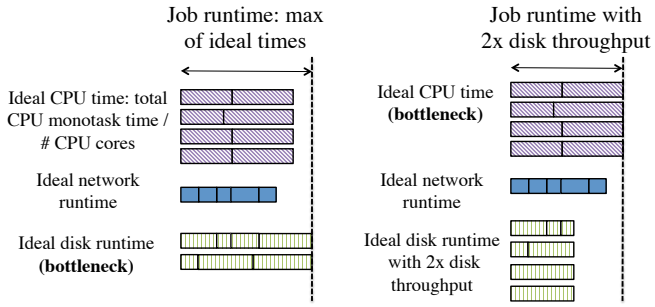

Figure 4: Monotask runtimes can be used to model job completion time as the maximum runtime on each resource. This example has four CPU cores and two disks.

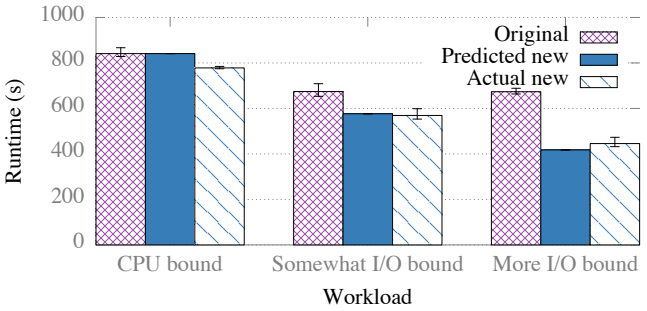

Figure 5: A simple monotask model can predict runtimes on different cluster configurations. In this example, monotask runtimes from experiments on a cluster of 20 8-core, one-SSD machines were used to predict how much faster jobs would run on a cluster with twice as many SSDs on each machine.

one SSD per worker, the predicted time on a cluster with two SSDs per worker (based on monotask runtimes on the one SSD cluster), and the actual runtimes on a cluster with two SSDs per worker. The left workload is CPU-bound, so the model predicts no change in the job's completion time as a result of adding another disk on each worker. In this case, the error is the largest (9%), because the workload does see a modest improvement from adding an extra disk as a result of shortening time at the beginning of the job when the job is waiting on data from disk before it can start computation. For the other two workloads, the model predicts the correct runtime within a 5% error. For both of these workloads, computing the new runtime with twice as many disks is not as simple as dividing the old runtime by two. In at least one of the stages of both workloads, adding an extra disk shifts the bottleneck to a different resource (e.g., to the network) leading to a smaller than $2\times$ reduction in job completion time. The monotask model correctly captures this.

5.2 Predicting runtime for in-memory data

The model can also be used for more sophisticated what-if scenarios; e.g., to estimate the improvement in runtime if input data were stored in-memory and deserialized, rather than serialized on disk. To model this scenario, the new ideal disk completion time is the original ideal disk completion time minus the time for monotasks that read input data (monotasks report metadata that includes whether the task was

to read input or write output). The new ideal compute monotask time also needs to be adjusted to eliminate time spent deserializing input data (this is also reported in monotask metadata). We used this approach to predict the runtime of a job that sorted random on-disk data if data were stored deserialized in-memory, and the model predicted the new runtime within an error of 4% (the model predicted that the job's runtime would reduce from 48.5 seconds to 38.0 seconds, and the job's actual runtime with in-memory data was 36.7 seconds).

6 LIMITATIONS

Based on our experience thus far with monotasks, using monotasks suffers from two fundamental limitations. First, if jobs are not broken into sufficiently many tasks, they will not be able to leverage the extra parallelism available with monotasks, and the explicit barriers between different resources will lead to longer runtimes. Today's jobs are broken into an increasingly large number of small tasks, driven by performance benefits of smaller tasks and decreasing task launch overheads [8], so we do not expect this limitation to be a major issue.

Another fundamental limitation of using monotasks is that all of a task's data needs to fit in memory, because the input data for a task is read in its entirety (e.g., by a disk read monotask) before computation begins. In contrast, today's frameworks include functionality to incrementally spill to disk. Jobs with large tasks will need to be broken into smaller tasks in order to be run using monotasks.

Our current MonoSpark implementation also has a few limitations that could be addressed in future work. Currently memory use is unregulated, which means workers can run out of memory. Monotask schedulers could prioritize monotasks based on the amount of remaining memory; e.g., the disk scheduler could prioritize disk write monotasks over read monotasks when memory is contended.

As mentioned in our evaluation of MonoSpark, disk monotasks do not leverage the OS buffer cache, which ensures that the disk monotask scheduler (and not the operating system) initiates all disk accesses. This hurts performance compared to Spark for some jobs. MonoSpark could leverage Spark's application-level cache to opportunistically avoid writing data to disk when it would fit in memory.

7 DISCUSSION

Thus far, this paper has argued for explicitly scheduling each resource to make it easier for *users* to reason about performance. Here, we argue that using monotasks also provides opportunities for the framework to optimize for better performance based on visibility about resource use.

Explicity separating different resources into monotasks allows the framework to automate some configuration. With today's frameworks, users need to specify how many tasks to run concurrently on each machine, either by configuring a fixed number of slots on each machine, or by specifying the aggregate resource use of each task. With monotasks, the correct amount of concurrency is built into each resource scheduler: each resource scheduler runs the minimum number of monotasks necessary to fully utilize the resource. With monotasks, the framework could also automatically determine whether to compress data, based on the relative length of the disk and CPU monotask queues. For example, disk monotasks could write uncompressed data by default, but add an extra compute monotask to compress data when the disk is contended and CPU cores are available.

Using monotasks also leads to opportunities to optimize disk use. The disk monotask scheduler currently balances requests evenly across available disks, independent of load, which is what current frameworks do. However, because queueing on each disk is explicit, the disk scheduler could assign disk writes to the disk with the shorter queue. Another optimization that the disk scheduler could implement would be to coalesce writes for data that will be read together. For example, the disk queue often contains shuffle blocks written by two different map tasks that will be read by the same reduce task. On observing these write monotasks in the queue, the disk scheduler could write them as a single block, to minimize future seeks.

Finally, using monotasks may be beneficial for emerging hardware architectures. Monotasks can be used to gracefully offload work to specialized hardware: a GPU monotask scheduler, for example, could opportunistically pick tasks off of the CPU monotask queue to execute on a GPU. Using monotasks could also be a natural fit for disaggregated datacenter architectures [5, 6].

8 CONCLUSION

The monotasks design represents a first step towards designing a system for performance clarity. Scheduling units of work, monotasks, that each use just one resource simplifies reasoning about performance and enables new optimizations that leverage visibility into performance bottlenecks. In future work, we plan to explore whether monotasks can be used in other systems beyond data analytics frameworks. We hope other system designers will propose new techniques and system architectures that, like using monotasks, make performance easier to understand and improve.

ACKNOWLEDGMENTS

We thank Radhika Mittal, John Ousterhout, Aurojit Panda, and Patrick Wendell for helpful comments on earlier drafts of this paper. We are appreciative of Shivaram Venkataraman for discussions during the tiny tasks project [8] that led to the idea of breaking jobs into small, single-resource units of work. This research was supported in part by a Hertz Foundation Fellowship, a Google PhD Fellowship, and Intel and other sponsors of UC Berkeley's NetSys Lab.

REFERENCES

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance Debugging for Distributed Systems of Black Boxes. In *Proc. SOSP*.

[2] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for Request Extraction and Workload Modelling. In *Proc. SOSP*.

[3] Charlie Curtsinger and Emery D. Berger. 2015. COZ: Finding Code that Counts with Causal Profiling. In *Proc. SOSP*.

[4] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI*.

[5] Intel. 2013. Intel, Facebook Collaborate on Future Data Center Rack Technologies. http://goo.gl/Gh2Ut. (2013).

[6] Intel. 2015. Intel Rack Scale Design. https://goo.gl/ovdKjC. (2015).

[7] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-Parallel Programs From Sequential Building Blocks. In *Proc. EuroSys*.

[8] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. 2013. The Case for Tiny Tasks in Compute Clusters. In *Proc. HotOS*.

[9] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *Proc. NSDI*.

[10] UC Berkeley AmpLab. 2014. Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/. (February 2014).

[11] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *NSDI*.

[12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. NSDI*.