Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks

Kay Ousterhout
UC Berkeley

Christopher Canel*
Carnegie Mellon University

Sylvia Ratnasamy
UC Berkeley

Scott Shenker
UC Berkeley, ICSI

ABSTRACT

In today's data analytics frameworks, many users struggle to reason about the performance of their workloads. Without an understanding of what factors are most important to performance, users can't determine what configuration parameters to set and what hardware to use to optimize runtime. This paper explores a system architecture designed to make it easy for users to reason about performance bottlenecks. Rather than breaking jobs into tasks that pipeline many resources, as in today's frameworks, we propose breaking jobs into *monotasks*: units of work that each use a single resource. We demonstrate that explicitly separating the use of different resources simplifies reasoning about performance without sacrificing performance. Monotasks provide job completion times within 9% of Apache Spark for typical scenarios, and lead to a model for job completion time that predicts runtime under different hardware and software configurations with at most 28% error. Furthermore, separating the use of different resources allows for new optimizations to improve performance.

ACM Reference Format:
Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of SOSP '17, Shanghai, China, October 28, 2017,* 17 pages.
https://doi.org/10.1145/3132747.3132766

*Work done while at UC Berkeley

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132766

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 worth it to get enough memory to cache on-disk data? How much will upgrading the network from 1Gbps to 10Gbps 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?

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 [32]. 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 workload changes shift the bottleneck.

Existing approaches have added performance visibility as an afterthought, e.g., by adding instrumentation to existing systems [4, 7, 11, 25]. 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 large-scale data analytics frameworks. We present an architecture that decomposes data analytics jobs into *monotasks* that each use exactly one of CPU, disk, or network. Each resource has a dedicated scheduler that schedules the

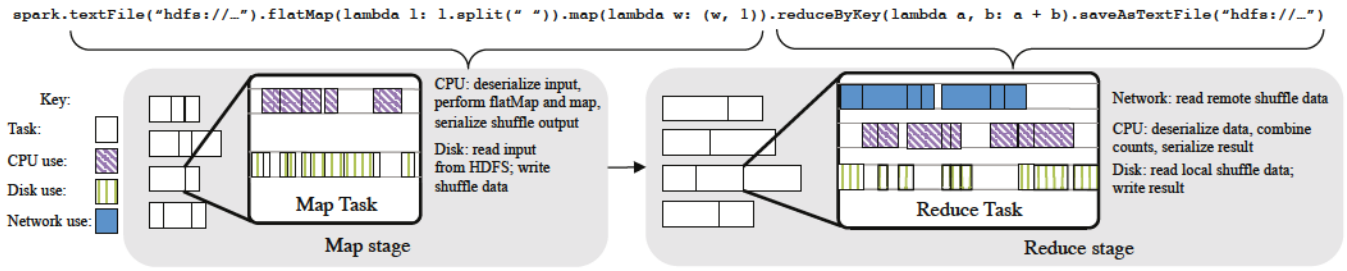


Figure 1: Execution of a Spark job, written using Spark's Python API, that performs word count. The job is broken into two stages, and each stage is broken into parallel tasks (shown as white boxes that execute in four parallel slots) that each pipeline use of different resources.

monotasks for that resource. This design contrasts with today's frameworks, which distribute work over machines by dividing it into tasks that each parallelize the use of CPU, disk, and network.

Decoupling the use of different resources into monotasks simplifies reasoning about performance. With current frameworks, the use of fine-grained pipelining to parallelize CPU, disk, and network results in ad-hoc resource use that can change at fine time granularity. This makes it difficult to reason about a job's bottleneck, because even a single task's resource bottleneck can change on short time horizons. Furthermore, tasks may contend at the granularity of the pipelining; e.g., when one task's disk read blocks waiting for a disk write triggered by a different task. This contention is handled by the operating system, making it difficult for the framework to report how different factors contributed to task runtime. 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 and, as a result, the resource use of the job as a whole. Controlling each resource with a dedicated scheduler allows that scheduler to fully utilize the resource and queue monotasks to control contention.

We used monotasks to implement MonoSpark, an API-compatible version of Apache Spark that replaces current multi-resource tasks with monotasks. MonoSpark does not sacrifice performance, despite eliminating fine-grained resource pipelining: on three benchmark workloads, MonoSpark provides job completion times within 9% of Spark for typical scenarios (§5). In some cases, MonoSpark outperforms Spark, because per-resource schedulers allow MonoSpark to avoid resource contention and under utilization that occur as a result of the non-uniform resource use during the lifespan of Spark tasks.

Using monotasks to schedule resources leads to a simple model for job completion time based on the runtimes of each type of monotask. We use the model to predict the runtime of MonoSpark workloads under different hardware configurations (e.g., with a different number or type of disk drives), different software configurations (with data stored de-serialized and in-memory, rather than on-disk), and with a combination of both hardware and software changes (§6). For most "what-if" questions, the monotasks model provides estimates within 28% of the actual job completion time; for example, monotasks correctly predicts the 10× reduction in runtime from migrating a workload to a 4× larger cluster with SSDs instead of HDDs.

Using monotasks also makes bottleneck analysis trivial. Monotask runtimes can easily be used to determine the bottleneck resource, and we demonstrate that the monotasks model can be used to replicate the findings of a recent research paper [25] that used extensive instrumentation to perform bottleneck analysis in Spark (§6.5).

Finally, using monotasks leads to new opportunities for performance optimizations. For example, MonoSpark can automatically determine the ideal task concurrency, which users are required to specify in existing frameworks. We illustrate one example where using monotasks to automatically determine the ideal concurrency improves job completion time by 30% relative to configuring a job's resource use.

2 BACKGROUND

2.1 Architecture of data analytics frameworks

This paper focuses on the design of data analytics frameworks like MapReduce [12], Dryad [18], and Spark [36] that provide an API for performing computations across large, distributed datasets. These frameworks rely on a bulk-synchronous-parallel model where jobs are broken into stages, and each stage is broken into parallel tasks that execute independently and may read shuffled data from earlier stages. Figure 1 shows an example job, expressed using Spark's Python API, and illustrates how the job would be broken into parallel tasks in two stages. Each task in a stage performs the same computation, but on a different block of input data.

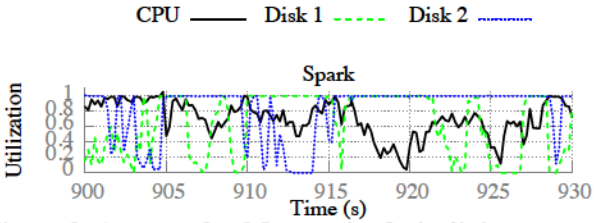


Figure 2: As a result of fine-grained pipelining, resource utilization during Spark jobs is non-uniform. This example illustrates the utilization during a 30 second period when 8 tasks were running concurrently, and when the bottleneck changes between CPU and disk.

These frameworks are widely used because they allow users to express computation using a simple API, and the framework abstracts away the details of breaking the computation into stages and tasks that can be run in parallel.

These frameworks use fine-grained pipelining to parallelize use of CPU, network, and disk within each task, as shown in Figure 1. In Spark, each task has a single thread that processes one record at a time: the map task, for example, reads one record from disk, computes on that record, and writes it back to disk. Pipelining is implemented in the background; for example, disk writes typically are written to buffer cache, and the operating system writes to disk asynchronously. Pipelining is not expressed by users as part of the API. For example, in the map stage in Figure 1, the user specifies the name of the input file and the transformation to apply to each line of input data, and the framework controls how the file contents are read from disk and passed to the user-specified computation. Frameworks implement pipelining to make tasks complete more quickly by parallelizing resource use.

2.2 The challenge of reasoning about performance

The fine-grained pipelining orchestrated by each task makes reasoning about performance difficult for three reasons:

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. For example, the reduce task in Figure 1 bottlenecks on CPU, network, and disk at different points during its execution. Figure 2 illustrates this effect during a thirty-second period of time when 8 concurrent Spark tasks were running on a machine. During this period, the resource utilization oscillates between being bottlenecked on CPU and being bottlenecked on one of the disks, as a result of fine-grained changes in each task's resource usage.

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, when multiple tasks simultaneously issue disk reads or writes for the same disk, those requests will contend and take longer to complete. This occurs in Figure 2 at 920 seconds, when all eight concurrent tasks block waiting on requests from the two disks.

Resource use occurs outside the control of the analytics framework. Resource use is often triggered by the operating system. For example, data written to disk is typically written to the buffer cache. The operating system, and not the framework, will eventually flush the cache, and this write may contend with later disk reads or writes.

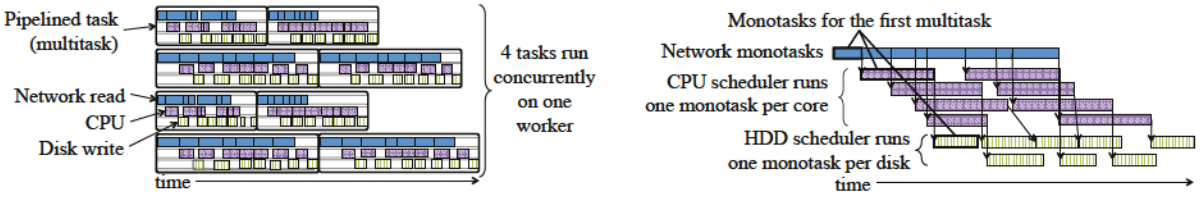
Together, these three challenges make reasoning about performance difficult. Consider a user who asks a question like "how much more quickly would my job have run if it used twice as many disks". To answer this question, she would need to walk through a task's execution at the level of detail of the pipelining shown in the example tasks in Figure 1. For each fine-grained use of network, CPU, and disk, 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.

Existing approaches to reasoning about performance have addressed this challenge in two different ways. One approach has focused on adding instrumentation to measure when tasks block on different resources; this instrumentation can be used to answer simple questions about resource bottlenecks, but cannot be used for more sophisticated what-if questions [25]. A second approach treats the analytics framework as a black box, and uses machine learning techniques to build a new performance model for each workload by running the workload (or a representative subset of the workload) repeatedly under different configurations (e.g., Ernest [32] and CherryPick [5]). These models can answer what-if questions, but require offline training for each new workload. The complexity of these approaches is mandated by the challenges of reasoning about performance in current frameworks.

Given the importance of the ability to reason about performance and the challenges to doing so today, this paper explores architectural approaches to providing performance clarity. We ask: would a different system architecture make it easy to reason about performance? And would such an architecture require a simplistic approach that sacrifices fast job completion times?

3 MONOTASKS ARCHITECTURE

This paper explores a system architecture that provides performance clarity by using single-resource units of work called monotasks. Our focus in this paper is to explore how systems can provide performance clarity, and as a result we do not focus on optimizing our implementation to maximize performance. Using monotasks enables many new scheduling



(a) Execution as today's multi-resource tasks.

(b) Execution as monotasks. Arrows represent dependencies.

Figure 3: Execution of eight multitasks on a single worker machine. With current frameworks, shown in (a), each multitask parallelizes reading data over the network, filtering some of the data (using the CPU), and writing the result to disk. Using monotasks, shown in (b), each of today's multitasks is decomposed into a DAG of monotasks, each of which uses exactly one resource. Per-resource schedulers regulate access to each resource.

optimizations, and we leave exploration of such optimizations to future work.

3.1 Design

The monotasks design replaces the fine-grained pipelining used in today's tasks – henceforth referred to as *multi*tasks – with statistical multiplexing across *mono*tasks that each use a single resource. The design is based on four principles:

Each monotask 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 use 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. These 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.

Figure 3 compares execution of multitasks on a single worker machine with current frameworks, shown in (a), with how those multitasks would be decomposed into monotasks and executed by per-resource schedulers, shown in (b).

The principles above represent the core ideas underlying monotasks. A variety of architectures could support these principles; for example, there are many approaches to scheduling monotasks. The remainder of this section focuses on the design decisions that we made to support monotasks specifically in the context of Apache Spark; we refer to the resulting system as MonoSpark.

3.2 How are multitasks decomposed into monotasks?

Decomposing jobs into monotasks does not require users to write their jobs as a sequence of monotasks, and can be done internally by the framework without changing the existing API. This is possible for frameworks such as Spark because they provide a high-level API: users describe the location of input data, the computation to perform on the data, and where to store output data, and Spark is responsible for implementing the necessary disk and network I/O. Because the framework is responsible for the details of how I/O occurs, it can replace the existing fine-grained pipelining with monotasks without requiring changes to user code.

With MonoSpark, the decomposition of jobs into monotasks is performed on worker machines rather than by the central job scheduler. The job scheduler works in the same way as in today's frameworks: it decomposes jobs into parallel multitasks that perform a computation on a single input data block, and that run on one machine. These multitasks are the same as tasks in current frameworks, and as in current frameworks, they are assigned to workers based on data locality: if a multitask reads input data that is stored on disk, it will be assigned to a worker machine that holds the input data.

Multitasks are decomposed into monotasks when they arrive on the worker machine. Figure 4 illustrates how jobs are decomposed into monotasks using the word count job from Figure 1 as an example. To read input data from HDFS, the job uses the `textFile` function, and passes in the name of a file in HDFS. With both Spark and MonoSpark, the job scheduler creates one multitask for each block of the file (HDFS breaks files into blocks, and distributes the blocks over a cluster of machines). Spark implements fine-grained pipelining to read and compute on the block: each Spark multitask pipelines reading the file block's data from disk with computation on the data. MonoSpark instead creates a disk read

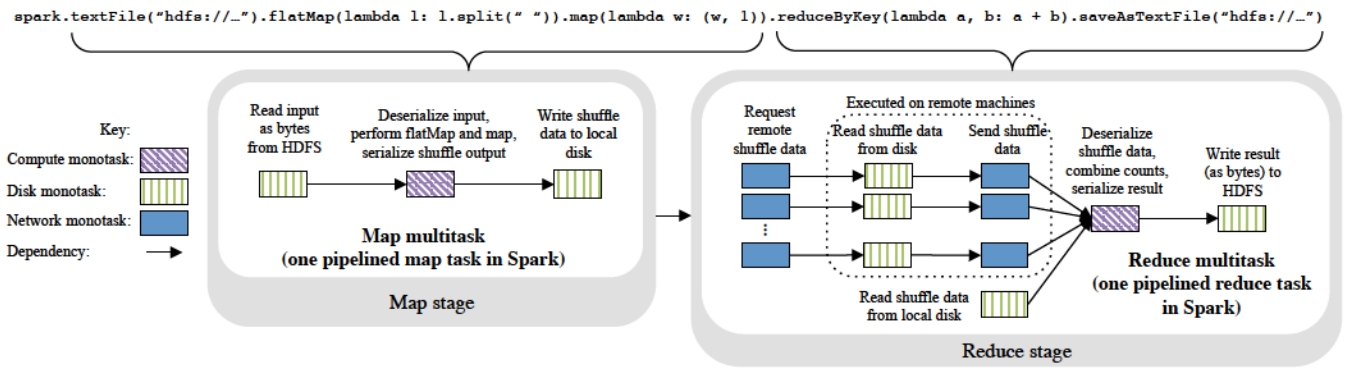


Figure 4: Decomposition of a Spark job that performs word count (written using Spark's Python API) into monotasks. The grey boxes show the two stages that the job would be broken into, and the white box within each stage illustrates how one multitask in the stage is decomposed into monotasks. For brevity the figure omits a compute monotask at the beginning of each multitask, which deserializes the task received from the scheduler and creates the DAG of monotasks, and a compute monotask at the end of each multitask, which serializes metrics about the task's execution.

monotask, which reads all of the file block's bytes from disk into a serialized, in-memory buffer, followed by a compute monotask, which deserializes the bytes, emits a count of 1 for each instance of each word, and serializes the resulting data into a new in-memory buffer. The shuffle is similarly specified using a high-level API: the function `reduceByKey` is used to aggregate all of the counts for each word, which is done using a shuffle. Spark uses pipelining to parallelize reading shuffle data from disk and over the network with deserializing the data and performing the computation specified in the `reduceByKey` call. MonoSpark implements the shuffle by breaking each reduce multitask into network monotasks that each issue a request for shuffle data to remote machines. When remote machines receive the request, they create a disk read monotask to read all of the requested shuffle data into memory, followed by a network monotask to send the data back to the machine where the reduce multitask is executing. Once all shuffle data has been received and placed in in-memory buffers, a compute monotask deserializes all of the shuffle data and performs the necessary computation. Other functions in Spark's API are implemented in a similar manner: Spark's API accepts a high-level description of the computation and the data to operate on, but not details of how resources are used, so MonoSpark can transparently change fine-grained pipelining to monotasks.

Because using monotasks eliminates the use of fine-grained pipelining to parallelize resource use, a single multitask will take longer to complete with monotasks than with today's frameworks. For example, for the map task in Figure 4, using monotasks serializes the resource use, causing the multitask to take longer than when resource use is parallelized using fine-grained pipelining (as in Figure 1). In order to avoid increasing job completion time, jobs must be broken down into enough multitasks to enable pipelining between *independent* monotasks in the same job. In other words, the monotasks design forgoes fine-grained pipelining within a single multitask and instead achieves high utilization through the statistical multiplexing of monotasks. We evaluate the sensitivity of MonoSpark's performance to the number of tasks in a job in §5.3, and discuss the resulting limitations on when MonoSpark can be used effectively in §8.

3.3 Scheduling monotasks on each worker

On each worker, monotasks are scheduled using two layers of schedulers. A top level scheduler, called the Local DAG Scheduler, manages the directed acyclic graph (DAG) of monotasks for each multitask. The Local DAG Scheduler tracks the dependencies for each monotask, and submits monotasks to the appropriate per-resource scheduler when all dependencies have completed. For example, for the map multitask in Figure 4, the Local DAG Scheduler will wait to submit the compute monotask to the compute scheduler until the disk read monotask has completed. The Local DAG Scheduler is necessary to ensure that monotasks can fully utilize the underlying resource and do not block on other monotasks during their execution.

The Local DAG Scheduler assigns monotasks to dedicated, per-resource schedulers that each seek to fully utilize the underlying resource while minimizing contention:

CPU scheduler The CPU scheduler is straightforward: one monotask can fully utilize one core, so the CPU scheduler runs one monotask per core and queues remaining monotasks.

Disk scheduler The hard disk scheduler runs one monotask per disk, because running multiple concurrent monotasks

reduces throughput due to seek time.[1] Flash drives, on the other hand, can provide higher throughput when multiple operations are outstanding. The flash scheduler exposes a configuration parameter that allows users to change the number of concurrent flash monotasks based on the underlying hardware. For the flash drives we used, we found that using four outstanding monotasks achieved nearly the maximum throughput (results omitted for brevity).

Network scheduler Efficiently scheduling the network is the most challenging, because scheduling network monotasks requires coordination across machines. If a machine A starts sending data to machine B, the transfer may contend with other flows originating at A, or other flows destined to B. Determining how to match sender demand and receiver bandwidth to optimize utilization is an NP-hard problem that has been studied extensively. MonoSpark adopts a simple approach where all scheduling occurs at the receiver, which limits its outstanding requests. We chose the number of outstanding requests to balance two objectives. Consider a machine running multiple reduce multitasks that are each fetching shuffle data from all machines where map multitasks were executed. Issuing the requests for just one multitask at a time hurts utilization: the multitask could be waiting on data from just one (slow) remote machine, causing the receiving link to be underutilized. On the other hand, issuing the requests for too many multitasks at a time also hurts performance. Because the monotasks design relies on coarse-grained pipelining between monotasks for different multitasks, jobs complete most quickly when all of the data for one multitask is received before using any of the receiver bandwidth for the next multitask's data. This way the compute monotask (to process received data) can be pipelined with the next multitask's network requests. To balance these two issues, we limit the number of outstanding requests to those coming from four multitasks, based on an experimental parameter sweep. As in other parts of the monotasks design, more sophisticated schedulers (e.g., based on distributed matching between senders and receivers, as in pHost [13] and iSlip [20]) are possible, and we leave exploration of these to future work.

Queueing monotasks When more monotasks are waiting for a resource than can run concurrently, monotasks will be queued. The queuing algorithm is important to maintaining high resource utilization when multitasks include multiple monotasks that use the same resource. Consider multitasks that are made up of three monotasks: a disk read monotask, a compute monotask, and a disk write monotask. If a queue of disk write monotasks accumulates (e.g., when disk is the bottleneck), as writes finish and new multitasks are assigned to the machine, the read monotasks for the new multitasks will be stuck in the long disk queue behind all of the writes. Because each compute monotask depends on a read monotask completing first, the CPU will remain idle until all of the disk writes have finished. After a burst of CPU use, the disk queue will again build up with the disk writes, and future CPU use will be delayed until all of the writes complete. This cycle will continue and harms utilization because it prevents CPU and disk from being used concurrently. Intuitively, to maintain high utilization, the system needs to maintain a pipeline of monotasks to execute on all resources. To solve this problem, queues implement round robin over monotasks in different phases of the multitask DAG. For example, in the example above, the disk scheduler would implement round robin between disk read and disk write monotasks.

[1] When disk monotasks transfer a small amount of data, disk throughput can be improved by running many parallel monotasks, so the disk scheduler can optimize seek time by re-ordering monotasks. We leave exploration of this to future work.

3.4 How many multitasks should be assigned concurrently to each machine?

The MonoSpark job scheduler works in the same way as the Spark job scheduler, with one exception: with MonoSpark, more multitasks need to be concurrently assigned to each machine to fully utilize the machine's resources. Consider Figure 3 as an example: with current frameworks, four tasks run concurrently (one on each core), whereas when jobs are decomposed into monotasks, four multitasks can concurrently be running CPU monotasks, while additional multitasks can use the disk and the network.

Before discussing how many multitasks to assign to each machine in MonoSpark, we note that the trade-offs in making this decision are different than in current frameworks. In current frameworks, the number of tasks concurrently assigned to each machine is intimately tied to resource use: schedulers must navigate a trade-off between assigning more tasks to each machine, to drive up utilization, and assigning fewer tasks, to avoid contention. With the per-resource monotask schedulers, on the other hand, each resource scheduler avoids contention by queuing monotasks beyond the number that can run efficiently on each resource. As a result, the job scheduler does not need to limit the number of outstanding tasks to avoid contention, and the primary risk is under-utilizing resources by assigning too few tasks per machine.

To ensure that all resources can be fully utilized, MonoSpark assigns enough multitasks that all resources can have the maximum allowed number of concurrent monotasks running, plus one additional monotask. For example, if a machine has four CPU cores and one hard disk, the job scheduler will assign ten concurrent multitasks: enough for four to have compute monotasks running, one to have a disk monotask running, four have monotasks running on the network, and one extra. The extra multitask exists for the purposes of the round-robin scheduling described in §3.3: without this extra monotask,

one of the queues in the round-robin ordering can get skipped, because it is temporarily empty while a new multitask is being requested from the job scheduler.

The strategy emphasizes simplicity, and a more sophisticated scheduling strategy that accounted for the types of monotasks that make up each multitask and the current resource queue lengths on each worker machine would likely improve performance. The goal of this paper is to explore the performance of a relatively simple and unoptimized monotasks design, so we leave exploration of more complicated scheduling techniques to future work.

3.5 How is memory access regulated?

Breaking jobs into monotasks means that more memory is used on each worker: in Figure 4, for example, all of the map multitask's data is read into memory before computation begins, whereas with current frameworks, fine-grained pipelining means that the data would be incrementally read, computed on, and written back to disk. This can result in additional garbage collection, which slows job completion time. This can also cause workers to run out of memory. Monotasks 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, to clear data out of memory. This paper does not explore strategies to regulate memory use; we discuss this limitation in more detail in §8.

4 IMPLEMENTATION

The previous section described the design of MonoSpark; this section describes lower-level implementation details. MonoSpark uses monotasks to replace the task execution code in Apache Spark [1, 36].[2] MonoSpark is compatible with Spark's public API: if a developer has written an application on top of Spark, she can change to using MonoSpark simply by changing her build file to refer to MonoSpark rather than Spark. MonoSpark inherits most of the Spark code base, and the application code running on Spark and MonoSpark is identical. For example, for a job that filters a dataset from disk and saves the result, Spark and Monotasks both read the same input data from the same place on disk, use the same code to deserialize the input data, run exactly the same Scala code to perform the filter, use the same code to serialize the output, and write identical output to disk. MonoSpark only changes the code that handles pipelining resources used by a task.

HDFS integration As part of integrating with Spark, our implementation also integrates with Hadoop Distributed File System (HDFS) [6], which is commonly used to store data that is analyzed with Spark. Spark uses a pipelined version of the HDFS API, where Spark reads one deserialized record at a time, and HDFS handles pipelining reading data from disk with decompressing and deserializing the data (writes work in a similar manner). In order to separate the disk and compute monotasks, we re-wrote the HDFS integration to decouple disk accesses from (de)serialization and (de)compression. We did this using existing HDFS APIs, so we did not need to modify HDFS.

Resource schedulers All of the per-resource schedulers are written at the application level and not within the operating system, meaning that resource use is not perfectly controlled. For example, we run one CPU monotask for each CPU core, but we do not pin these tasks to each core, and tasks may be interrupted by other monotasks that need small amounts of CPU (e.g., to initiate a disk read). We find that MonoSpark enables reasoning about performance in spite of these imperfections.

5 MONOTASKS PERFORMANCE

In this section, we compare performance of MonoSpark to Spark. We find that changing fine-grained pipelining to coarse-grained, single-resource monotasks does not sacrifice performance: MonoSpark provides job completion times within 9% of Apache Spark for typical scenarios.

5.1 Experimental setup

We ran all of our experiments on clusters of Amazon EC2 instances that have 8 vCPUs, approximately 60GB of memory, and two disks. Some experiments use m2.4xlarge instances with two hard disk drives (HDDs), while others uses i2.2xlarge instances with one or two solid state drives (SSDs), in order to illustrate that monotasks works with both types of disks. Our experiments compare Spark version 1.3 and MonoSpark, which is based on Spark 1.3. We ran at least three trials of each experiment, in addition to a warmup trial (to warmup the JVM) that was discarded, and except where otherwise noted, all plots show the median with error bars for the minimum and maximum values.

The version of Spark we compared against is known to have various CPU inefficiencies, and recent efforts have produced significant improvements in runtime, both in newer versions of Spark [33], and in alternative systems [10, 21]. As described in §4, we changed only the parts of Spark related to resource pipelining, so MonoSpark inherits Spark's CPU inefficiencies. MonoSpark would similarly inherit recent optimizations, which are orthogonal to the use of monotasks. For example, efforts to reduce serialization time would reduce the runtime for the compute monotasks that perform (de)serialization in MonoSpark. We also designed our workloads to avoid any single bottleneck resource, as we elaborate

[2]The MonoSpark code and scripts to run the experiments in the evaluation are available at https://github.com/NetSys/spark-monotasks.

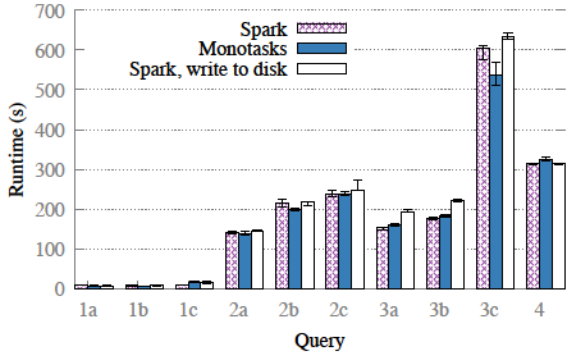


Figure 5: Comparison of Spark and MonoSpark for queries in the big data benchmark, using scale factor of 5, compressed sequence files, and 5 worker machines. Two configurations of Spark are shown: the default, and a configuration where Spark writes through to disk rather than leaving disk writes in the buffer cache.

on in §5.2, and to include a workload (the machine learning workload) that avoids Spark-specific CPU overheads.

5.2 Does getting rid of fine-grained pipelining hurt performance?

We evaluate MonoSpark using three workloads that represent a variety of applications and a variety of performance bottlenecks. In order to effectively evaluate whether changing fine-grained pipelining to single-resource monotasks harms performance, we chose workloads that do not have one resource with dramatically higher utilization than all other resources, as we elaborate on below. Workloads with several highly utilized resources put the most stress on strategies that re-arrange resource use, because resource use needs to be parallelized to maintain fast job completion time.

This section presents the high-level results of comparing MonoSpark to Spark for the three benchmark workloads; §5.3 and §5.4 describe differences in performance in more detail.

Sort The first workload sorts 600GB of random key-value pairs from disk using twenty worker machines that each have two hard disk drives. The job reads the input data from HDFS, sorts it based on the key, and stores the result back in HDFS. We tuned the workload to use CPU and disk roughly equally by adjusting the size of the value (smaller values result in more CPU time, as we elaborate on in §6.2). For this workload, Spark sorts the data in a total of 88 minutes (36 minutes for the map stage and 52 minutes for the reduce stage), and MonoSpark sorts the data in 57 minutes (22 minutes for the map stage and 35 minutes for the reduce stage). The reason MonoSpark is faster than Spark for this workload is discussed further in §5.4.

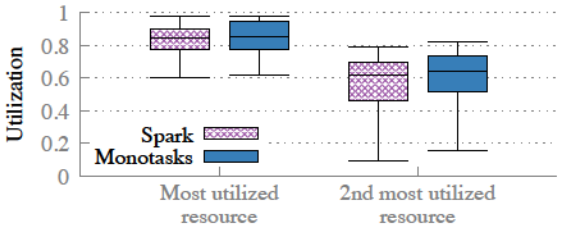


Figure 6: Utilization of the most utilized (i.e., bottleneck) resource, and the second most utilized resource during stages in the big data benchmark, for both Spark and MonoSpark. Boxes show the 25, 50, and 75th percentiles; whiskers show 5th and 95th percentiles.

Big Data Benchmark The big data benchmark [31] was developed to evaluate the differences between analytics frameworks and was derived from a benchmark developed by Pavlo et al. [27]. The input dataset consists of HTML documents from the Common Crawl document corpus [2] combined with SQL summary tables generated using Intel's Hadoop benchmark tool [35]. The benchmark consists of four queries including two exploratory SQL queries, one join query, and one page-rank-like query. The first three queries have three variants that each use the same input data size but have different result sizes to reflect a spectrum between business-intelligence-like queries (with result sizes that could fit in memory on a business intelligence tool) and ETL-like queries with large result sets that require many machines to store. The fourth query performs a transformation using a Python script. We use the same configuration that was used in published results [31]: we use a scale factor of five (which is the largest scale factor available) and a cluster of five worker machines that each have two HDDs.

Figures 5 compares runtime with MonoSpark to runtime with Spark for each query in the workload. For all queries except 1c, MonoSpark is at most 5% slower and as much as 21% faster than Spark. Query 1c takes 55% longer with MonoSpark, which we discuss in more detail, along with the second configuration of Spark, in §5.3.

Figure 6 shows the resource utilization of the two most utilized resources on each executor during each stage of the big data benchmark queries, and illustrates two things. First, multiple resources were well-utilized during most stages. Second, MonoSpark utilized resources as well as or better than Spark.

We also ran the big data benchmark queries on machines with two SSDs (as opposed to two HDDs), to understand how different hardware affects the relative performance of Spark and MonoSpark. On SSDs, the MonoSpark is at most 1% slower than Spark and up to 24% faster; detailed results are omitted for brevity.

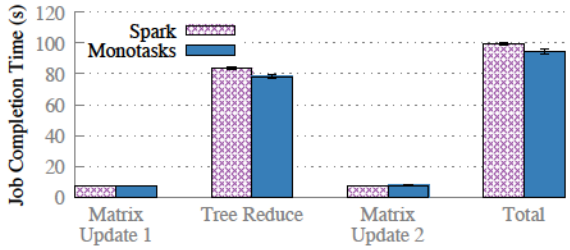


Figure 7: Comparison of Spark and MonoSpark for each stage of a machine learning workload that computes a least squares fit using 15 machines.

Machine Learning The final workload is a machine learning workload that computes a least squares solution using a series of matrix multiplications[3] on a cluster of 15 machines, each with 2 SSDs. Each multiplication involves a matrix with one million rows and 4096 columns. Tasks in the workload each perform a matrix multiplication on a block of rows. This workload differs from the earlier workloads for three reasons. First, it has been optimized to use the CPU efficiently: matrices are represented as arrays of doubles that can be serialized quickly, and each task calls out of the JVM to optimized native code written using OpenBLAS [3]. Second, a large amount of data is sent over the network in between each stage; combined with the fact that CPU use has been optimized, this workload is network-intensive. Finally, the workload does not use disk, and stores shuffle data in-memory. As with other workloads, MonoSpark provides performance on-par with Spark, as shown in Figure 7.

5.3 When is MonoSpark slower than Spark?

MonoSpark can be slower than Spark in two cases. First, MonoSpark can be slower when a workload is not broken into sufficiently many multitasks. By using monotasks, MonoSpark eliminates fine-grained pipelining, a technique often considered central to performance. MonoSpark's coarser-grained pipelining will sacrifice performance when the pipelining is too coarse, which occurs when the job has too few multitasks to pipeline one multitask's disk read, for example, with an earlier multitask's computation. Figure 8 plots the runtime of MonoSpark and Spark for a workload that reads input data from disk and then performs a computation over it, using different numbers of tasks. For each number of tasks, we repartition the input data into a number of partitions equal to the desired number of tasks. When the number of tasks is equal to the number of cores (the left most point), MonoSpark is slower than Spark, but as the number of tasks increases,

[3]The workload uses the block coordinate descent implementation in a distributed matrix library for Apache Spark, available at https://github.com/amplab/ml-matrix/.

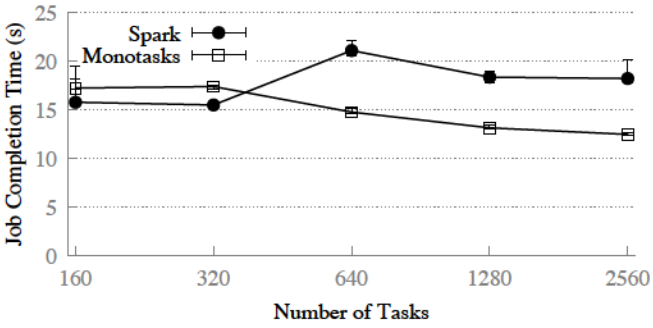


Figure 8: Comparison of runtime with Spark and MonoSpark for a job that reads input data and then computes on it, running on 20 workers (160 cores). Spark is faster than MonoSpark with only one or two waves of tasks, but by three waves, MonoSpark's pipelining across tasks has overcome the performance penalty of eliminating fine grained pipelining.

MonoSpark can do as well as Spark by pipelining at the granularity of monotasks. This effect did not appear in any of the benchmark workloads that we ran because the default configuration for all of those workloads broke jobs into enough tasks to enable pipelining across different monotasks.

The second reason MonoSpark may be slower than Spark stems from how disk writes are treated: Spark writes data to buffer cache, and does not force data to disk. Disk monotasks, on the other hand, flush all writes to disk, to ensure that future disk monotasks get dedicated use of the disk, and because the ability to measure the disk write time is critical to performance clarity. This difference explains why query 1c in the big data benchmark was 55% slower with MonoSpark: each disk writes 511MB of data for MonoSpark but less than 200KB for Spark. We configured the operating system to force Spark to flush writes to disk and the resulting big data benchmark query runtimes are show in Figure 5. When Spark writes the same amount of output to disk as MonoSpark, query 1c is only 9% slower with MonoSpark.

5.4 When is MonoSpark faster than Spark?

In some cases, MonoSpark can provide faster runtimes than Spark. This is occurs for two reasons. First, per-resource schedulers control contention, which results in higher disk bandwidth for workloads that run on hard disk drives, due to avoiding unnecessary seeks. The effect explains MonoSpark's better performance on the sort workload and on some queries in the big data benchmark, where controlling disk contention resulted in roughly twice the disk throughput compared to when the queries were run using Spark.

Second, the per-resource schedulers allow monotasks to fully utilize the bottleneck resource without unnecessary

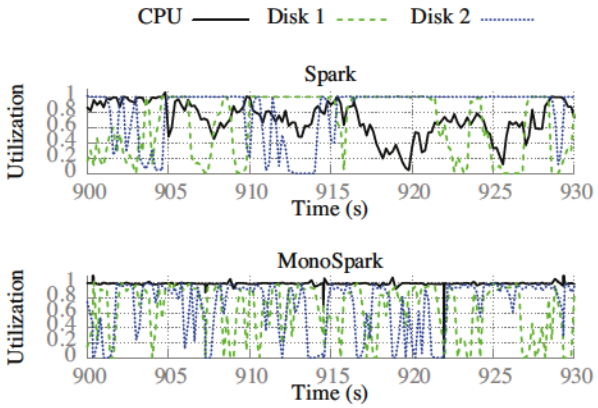


Figure 9: Utilization during the map stage of query 2c in the big data benchmark. With MonoSpark, per-resource schedulers keep the bottleneck resource fully utilized.

contention. Figure 9 shows one example when MonoSpark achieved better utilization. The figure plots the utilization on one machine during a thirty second period in the map stage of query 2c in the big data benchmark. Over the entire map stage, the per-resource schedulers in MonoSpark keep the bottleneck resource, CPU, fully utilized: the average utilization is over 92% for all machines. With Spark, on the other hand, each task independently determines when to use resources. As a result, at some points, tasks bottleneck on the disk while CPU cores are unused, leading to lower utilization of the CPU (75 - 83% across all machines) and, as a result, longer completion time. This problem would persist even with proposals that use a task resource profile to assign tasks to machines [16]. Resource profiles based on average task resource use do not account for how resource consumption arrives over the course of the task, so cannot avoid contention caused by fine-grained changes in resource use.

6 REASONING ABOUT PERFORMANCE

Explicitly separating the use of different resources into monotasks allows each job to report the time spent using each resource. These times can be used to construct a simple model for the job's completion time, which can be used to answer what-if questions about completion time under different hardware or software configurations. In this section, our goal is not to design a model that perfectly captures job completion time; instead, our goal is to design a model that is simple and easy to use, yet sufficiently accurate to provide estimates to users who are evaluating the benefit of different hardware or software configurations. We use the model to answer four "what-if" questions about how a job's runtime would change using a different hardware configuration, software configuration, or a combination of both. The model provides predictions within 28% of the actual runtime – even when the runtime changes by as much as a factor of 10. In addition, we show that our model makes it trivial to determine a job's bottleneck and demonstrate its use to replicate the bottleneck analysis results from [25]. Finally, we evaluate the effectiveness of applying the same model to Spark.

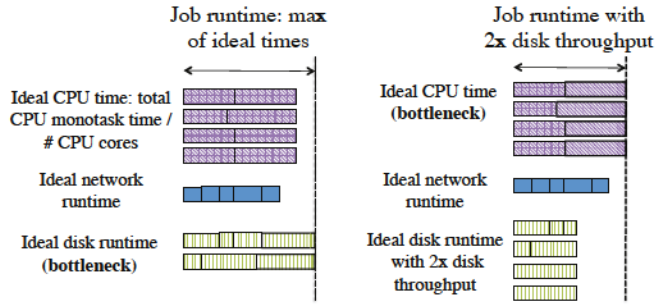


Figure 10: Monotask runtimes can be used to model job completion time as the maximum runtime on each resource. This example has 4 CPU cores and 2 hard disks.

6.1 Modeling performance

Decomposing jobs into monotasks leads to a simple model for job completion time. To model job completion time, each stage is modeled separately (since stages may have different bottlenecks), and the job completion time is the sum of the stages' completion times. Modeling the completion time of a stage involves two steps. First, information about the monotasks can be used to compute the ideal time spent running on each resource, which we call the ideal resource completion time. 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 example, if a job's CPU monotasks took a total of twenty minutes, and the job ran on eighty cores, the ideal CPU time would be fifteen seconds, because all of the CPU monotasks could have completed in fifteen seconds if perfectly parallelized. For I/O resources, the ideal resource time can be calculated using the total data transmitted and the throughput of the resource: ideal I/O resource time $= \frac{\text{sum of data transferred}}{\text{resource throughput}}$. For example, if a stage read twenty gigabytes from disk, and used ten disks that each provided 100 megabytes per second of read throughput, the ideal disk time would be twenty seconds.

The second step in building the model is to compute the ideal stage completion time, which is simply the maximum of any resource's completion time, as shown in Figure 10. In essence, the ideal stage completion time is the time spent on the bottleneck resource.

This model is simple and ignores many practicalities, including the fact that resource use cannot always be perfectly parallelized. For example, if one disk monotask reads much more data than the other disk monotasks, the disk that executes that monotask may be disproportionately highly loaded. Furthermore, all jobs have a ramp up period where only one resource is in use; e.g., while the first network monotasks are

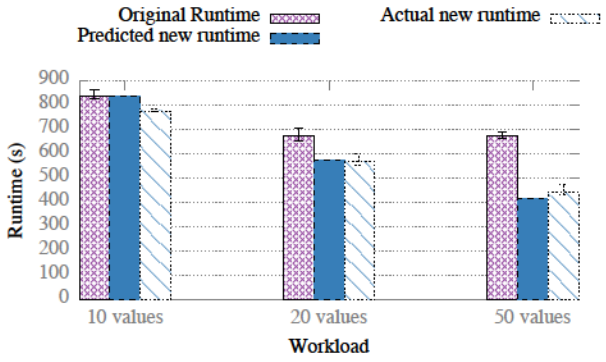


Figure 11: Monotask runtimes can be used to model job runtime on different cluster configurations. In this example, monotask runtimes from experiments on a cluster of 20 8-core, 1 SSD machines was used to predict how much faster the job would run on a cluster with twice as many disks on each machine.

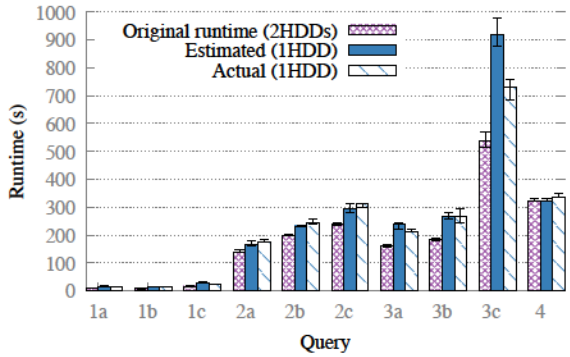


Figure 12: For all queries in the big data benchmark except 3c, the monotasks model correctly predicts the change from changing each machine to have only 1 disk instead of 2.

executing for a reduce task. Despite these omissions, we find that our model is sufficiently accurate to answer broad what-if questions, which we believe is preferable to a perfectly accurate but complex model that is difficult to understand and apply.

The model for job completion time can trivially be used to determine the bottleneck, which is simply the resource with the longest ideal resource completion time.

6.2 Predicting runtime on different hardware

While understanding the bottleneck is useful, the model provides the most value in allowing users to answer what-if questions about performance. For example, to compute how much more quickly a job would complete if the job had twice as much disk throughput available, the ideal disk time would be divided by two, and the new modeled job completion time would be the new maximum time for any of the resources. An example of this process is shown on the right side of Figure 10. To compute the new estimated job completion time, we scale the job's original completion time by the predicted change in job completion time based on the model. This helps to correct for inaccuracies in the model; e.g., not modeling time when resource use cannot be perfectly parallelized.

Figure 11 illustrates the effectiveness of the model in predicting the runtime on a cluster with twice as many SSDs for three different versions of a sort workload. The figure shows the runtime on a 20-machine cluster with 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 workload sorts 600GB of key-value pairs, where each value is an array of longs. Increasing the size of the value array while fixing the total data size decreases the CPU time (because fewer keys need to be sorted) while keeping the I/O demand fixed. We change the size of the value array (to be 10, 20, and 50) to evaluate how effective the model is for different balances of resources. With only 10 values associated with each key, the 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 (this improvement stems from reducing the duration of transient periods where the disk is the bottleneck, e.g., when starting a stage, where data must be read from disk before other resources can start). 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 isn't 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× reduction in job completion time. The monotasks model correctly captures this.

Figure 12 uses the model to predict the effect of removing one of the two disks on each machine in the big data benchmark workload from §5. A user running the workload might wonder if she could use just one disk per machine instead of two, because most queries in the workload are CPU bound (as discussed further in §6.5). The monotasks model correctly predicts that most queries change little from eliminating a disk: the predictions for all queries except query 3c are within 9% of the actual runtime.

The prediction has highest error for query 3c, when it overestimates the new runtime by 28%. This occurs because monotasks incorrectly predicts the change in runtime for a large shuffle stage that makes up approximately half of the job runtime. On-disk shuffles are the most difficult types of stage in which to get high utilization for both MonoSpark and Spark, because maintaining high utilization for the multitasks running on any one machine requires a steady flow of shuffle

data being read from *all* of the other machines in the cluster. A single slow disk can block progress on all other machines while those machines wait for shuffle data from the slow disk. This problem is most pronounced when all three resources are used equally, in which case all resources on all machines must consistently be driven to high utilization to get the best performance. All three resources are used equally for the two-disk configuration, and performance is poor as a result: the average CPU, disk, and network utilization on each executor during this stage is roughly 50%. Since all resources are evenly bottlenecked, the model predicts that the runtime for that stage will increase by a factor of two once a disk is removed. Instead, when a disk is removed, MonoSpark is able to drive the bottleneck resource (now, the disk) to a higher utilization, and as a result, the stage only gets approximately 40% slower. Improving monotasks scheduling to enable uniformly higher resource utilizations would reduce this modeling error.

6.3 Predicting runtime with deserialized 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. This requires modeling two changes. First, data no longer will be read from disk. We account for this change by not including time for disk monotasks that read input data when we compute the ideal disk time (this is possible because each monotask reports metadata that includes whether the task was to read input or write output). Second, the job will spend less time using the CPU, because input data is already deserialized. MonoSpark separates the compute monotask into a first part that deserializes all of the data, and a second part that performs the remaining computation, and the compute monotask reports how long each part took. To model job completion time when data is already deserialized, we do not include the time to deserialize input data when modeling ideal CPU time. Using the new modeled ideal times, we calculate the new job completion time. 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).

Separating the deserialization time from the rest of the computation is only possible because of the use of monotasks. Deserialization time cannot be measured in Spark because of record-level pipelining: Spark deserializes a single record and computes on that record before deserializing the next record, and processing a single record is too fast to time without significant overhead.

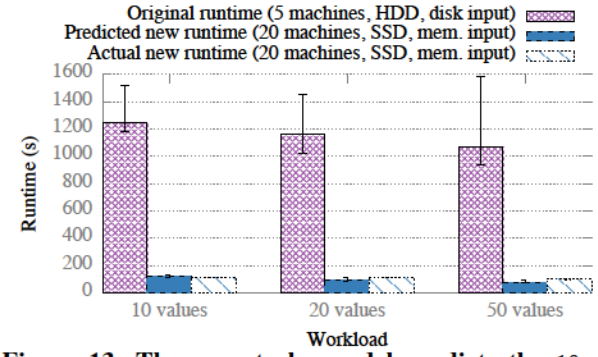


Figure 13: The monotasks model predicts the 10× improvement in runtime resulting from moving a workload from 5 machines with hard-disk-drives and input stored on-disk to 20 machines with solid-state-drives and input stored deserialized and in-memory with an error of at most 23%.

6.4 Predicting with both hardware and software changes

Thus far, we have shown how the monotasks model can be used to predict runtime changes resulting from a single change to the hardware or software configuration. The simple model can also accurately predict runtime changes stemming from multiple changes to both hardware and software. Figure 13 illustrates the change in runtime for 3 workloads that are moved from a 5-machine HDD cluster to a 20-machine SSD cluster. All workloads read 100GB of input data, sort it, and write it back to disk; the three workloads vary in the ratio of CPU to disk time, as in §6.2. The 100GB of input data did not fit in memory on the 5 machine cluster (it takes up approximately 200GB in memory), but with 20 machines, there is enough cluster memory to store input data in a deserialized format. In total, there are three changes associated with moving the workload to 20 machines: the workload runs on 4× as many machines (the number of tasks stays constant), input data is stored deserialized and in-memory rather than on-disk, and machines each have 2 SSDs rather than 2 HDDs, so shuffle and output data can be read and written more quickly. The monotasks model correctly predicts the resulting 10× change in runtime with an error of 23% in the worst case.

To give an example of the changes that occurred as a result of the hardware and software changes, the workload with 10 values was bottlenecked by disk on the 5-machine cluster in both the map and reduce stages. The monotasks model correctly predicted that, given the hardware changes, the map stage became CPU-bound (due to a combination of input data being stored in-memory, and the faster write time for the shuffle data), and the reduce stage became network bound (due to the faster read time for shuffle data and write time for output). One source of error for all three workloads was

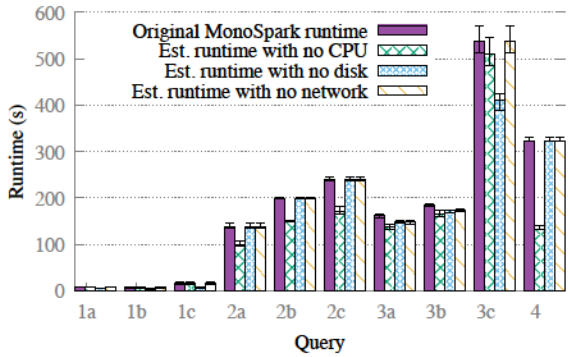


Figure 14: Monotask runtimes can be used to replicate previous work that used extensive logging to determine the job completion time if different resources were optimized to be infinitely fast (which serves as a bound on improvements from optimizing that resource).

the network time. The model assumed that the same amount of data was sent over the network in both the 5-machine and 20-machine case. However, with 5 machines, each task could read 20% of its input data locally, whereas with 20-machines, an average of only 5% of input data could be read locally. As a result, the workloads on 20-machines sent more data over the network (and, correspondingly, spent more time using the network) than the model predicted.

6.5 Understanding bottlenecks

In addition to predicting runtime under different hardware and software configurations, our model can be used for simpler bottleneck analysis. Recent work [25] dedicated to answering that question added instrumentation to Spark to measure times blocked on the network and disk, and used those measurements to determine the best-case improvement from optimizing disk and network: i.e., it used blocked times to determine how much faster jobs would run if they did not spend any time blocked on a particular resource. The analysis relied on adding extensive white-box logging to Spark; with monotasks, the necessary instrumentation (i.e., the runtime of different types of monotasks) is built into the framework's execution model. Typically the monotasks model predicts job completion time by taking the maximum of the ideal CPU, network, and disk times. The best-case job completion time if the disk were optimized can be computed by simply excluding the disk from the maximum, so instead taking the maximum over the ideal network and CPU times. Figure 14 illustrates these predictions using monotask runtimes for the big data benchmark used in [25]. We arrive at the same findings as [25]: for the big data benchmark, CPU is the bottleneck for most queries, improving disk speed could reduce runtime of some queries, and improving network speed has little effect. Some queries see improvement from optimizing multiple resources

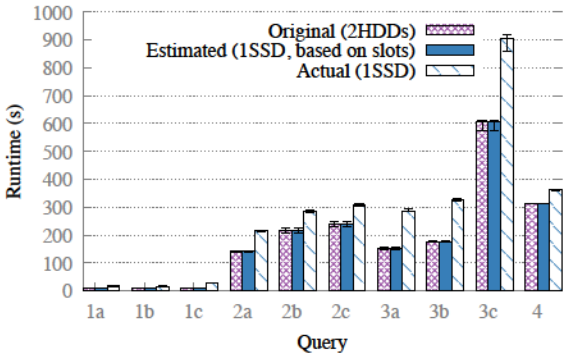


Figure 15: Using the number of slots to model the change in job completion time from reducing the number of disks does not result in correct predictions.

(e.g., query 3c) because the query consists of multiple stages that each have different bottlenecks.

6.6 Can this model be used for Spark?

This section explores whether the model we used to predict performance with MonoSpark could be applied to Spark, without re-architecting internals. Creating a model for job completion time is easy in MonoSpark because instrumentation for use of different resources is built into the framework architecture: resource use is separated into monotasks, and each monotask reports how long it took to complete. While MonoSpark controls resource use with per-resource monotasks, Spark controls resource use with slots: as mentioned in §3.4, the job scheduler assigns tasks to machines in a fixed number of slots, and controlling this number of slots is the only mechanism the scheduler has for regulating resource use. The most straightforward way to apply the monotasks model to Spark is to use slots in the same way that the monotasks model uses monotasks: for example, if a job took 10 seconds to complete on a cluster with 8 slots, it should take 5 seconds to complete on a cluster with 16 slots. Figure 15 illustrates the effectiveness of this model, for the same prediction shown in Figure 12, where the runtime of the big data benchmark on machines with 2 HDDs is used to predict runtime on a cluster with 1 HDD per machine. Spark sets the number of slots to be equal to the number of CPU cores, so changing the number of disk drives does not change the number of slots. As a result, this model is inaccurate: it does not account for the slowdown that occurs when queries become disk bound. The user could scale the number of slots to 4 rather than 8, to account for the reduction in disk usage, but this would lead to predicting that queries would take twice as long with the reduced number of disks, which is only true for disk-bound queries. Fundamentally, the problem is that Spark uses one dimension, slots, to control resource use that is multi-dimensional.

Unfortunately, measuring Spark's resource usage to create a more sophisticated model is difficult. Spark tasks on

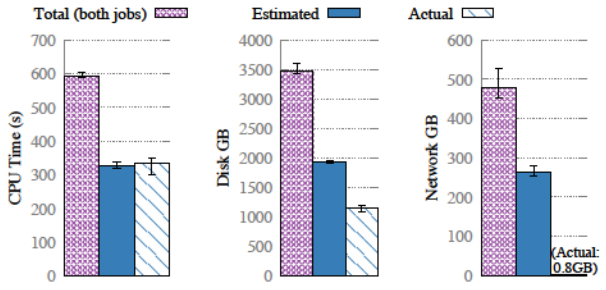


Figure 16: When multiple jobs are run concurrently in Spark, attributing resource to a particular job is difficult to do accurately. For one stage of one of the jobs, our estimate of resource usage was consistently inaccurate.

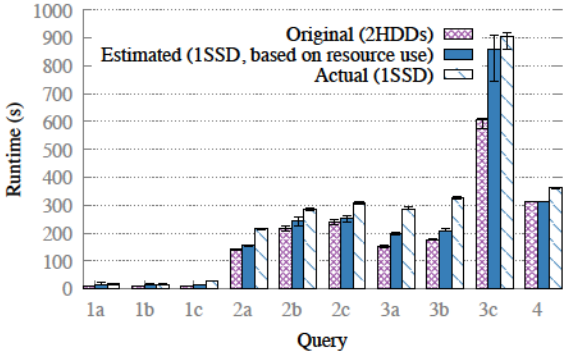


Figure 17: Even in cases where Spark's resource use can be measured to make a more accurate model than the slot-based approach in Figure 15, a Spark-based model has an error of 20-30% for most queries.

a machine all run in a single process (the Java virtual machine), and resource usage of different concurrent tasks is interleaved by both Spark and the operating system scheduler. We measured this effect by running two different sort workloads concurrently: the 10-value and 50-value workloads from §6.2. We estimated resource use for each job by measuring the total resource use on each executor while each stage was running, and estimating each stage's resource use by scaling each executor's total resource use by the fraction of slots that were used by tasks for the stage. For example, if a stage ran for 10 seconds, and one executor with eight slots ran four, five-second tasks, we would divide the total resource use on the executor by four. Figure 16 shows this estimate, for the map stage in the 10-value job. These estimates are consistently incorrect, sometimes by a factor of two or more, because resource use is attributed equally to both jobs, rather than accounting for the jobs' different resource profiles. The median and 75th percentile error for all resources in both stages of both jobs is 17% and 68%, respectively, with Spark. Monotask times can easily be used to decouple resource use for the same two jobs: with MonoSpark, the error is consistently less than 1%.

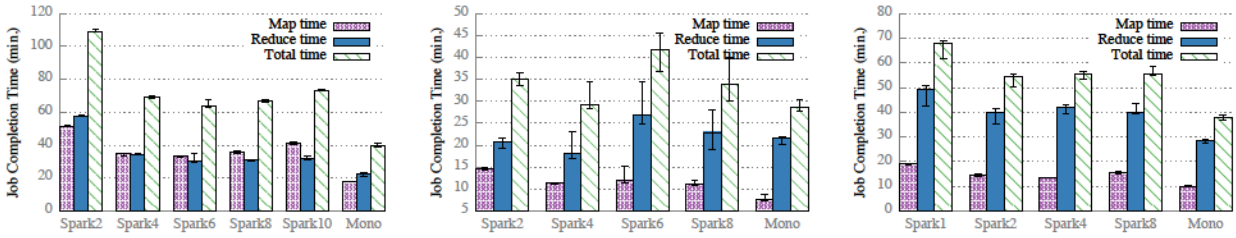
Figure 16 illustrated why measuring resource use in Spark is difficult. If we *could* measure the resource use in Spark, would the model be accurate? While we cannot measure the resource use of each task, the monotask model relies on taking the aggregate resource use for an entire stage (by summing all of the monotask times for each resource, in each stage). We approximated this process in Spark by measuring the resource use on each executor while the big data benchmark is running in isolation. Because no other workloads were running, all of the resource use on each executor can be attributed to the stage. We used these resource uses as inputs to the monotasks model; Figure 17 illustrates the results. The model underestimates the increase in job completion time that results from using only one disk: the error ranges from 3%, for query 1a, to over 50%, for query 1c. This error stems from the challenges to reasoning about performance outlined in §2. In particular, one source of error is contention: when all tasks use one disk, contention leads to reduced disk throughput, which is not captured by the model.

We are able to model Spark performance only in a restricted case (when a job runs in isolation) and even in this case, the error was higher than the error for the same scenario using MonoSpark. This model can only be used for hardware changes; as mentioned in §6.3, it is not possible to measure deserialization time in Spark.

7 LEVERAGING CLARITY: AUTO-CONFIGURATION

Because MonoSpark explicitly schedules the use of each resource, the framework has better visibility to automatically perform configuration that users are typically required to do. One configuration parameter that MonoSpark can set automatically is the appropriate amount of concurrency on each worker. Spark sets the number of concurrent tasks on each worker to be the number of cores on the worker, and allows users to change this by setting a configuration parameter. Different configurations will work better for different workloads; for example, the user might want to increase concurrency to more than the number of cores if each worker machine has a large number of disks, and tasks are not CPU-intensive. MonoSpark eliminates this configuration parameter, because concurrency is controlled by each resource scheduler, as described in §3.4.

Figure 18 compares performance with MonoSpark to performance under a variety of Spark configurations, for three different workloads. The workloads each sort randomly generated key-value pairs by key, and differ in the number of values associated with each key, as in the workload in §6.2. The different characteristics of each workload mean that the best Spark configuration differs across workloads, and to extract optimal performance, a user would need to tune this



(a) Single Long (b) 25 Longs (c) 100 Longs

Figure 18: Runtimes for three different jobs, each under different configured numbers of tasks per machine with Spark (e.g., Spark2 is Spark with 2 tasks per machine). MonoSpark automatically configures the number of tasks per machine, and performs at least as well as the best Spark configuration for all three jobs.

configuration parameter to the correct value. MonoSpark automatically uses the ideal amount of concurrency for each resource, and as a result, performs at least as well as the best Spark configuration for all workloads. In some cases, MonoSpark performs as much as 30% better than Spark. This is for two reasons: first, Spark doesn't allow users to change the number of concurrent tasks between the stages, and sometimes the ideal value differs for the two stages. Second, the disk monotask scheduler controls concurrent disk accesses to avoid unnecessary seeking.

8 LIMITATIONS AND OPPORTUNITIES

This section begins with fundamental limitations of using monotasks, and then discusses opportunities to improve our current implementation.

Jobs with few tasks As illustrated in §5.3, if a job has a small number of multitasks relative to the available resources, then there may be no opportunities for coarse-grained pipelining across different monotasks. For the benchmark workloads we ran, we did not find this to be a problem, because the default configuration of all three workloads broke jobs into a sufficiently large number of multitasks. Many frameworks today encourage the use of a large number of small multitasks for performance reasons: having a large number of small tasks mitigates the effect of stragglers [24, 34] and helps to avoid situations where the task's intermediate data is so large that it needs to be spilled to disk [29]. Workloads that have only a single wave of multitasks will need to be broken into a larger number of smaller multitasks in order to run efficiently with monotasks, which can be done by changing a parameter when jobs are submitted.

Jobs with large tasks Frameworks like Spark and Hadoop allow tasks to handle more data than can fit in memory: if a task's intermediate data does not fit in memory, tasks will spill the data to disk and merge it at the end. However, in MonoSpark, data must fit in a buffer in-memory after it is read from (or before it is written to) persistent storage. As is the case for jobs with too few tasks, jobs with multitasks that are too large will need to be run with a larger number of smaller multitasks with monotasks.

Head of line blocking A monotask that reads a large amount of data from disk may block other tasks reading from that disk. This is not an issue with current frameworks because tasks share access to each resource at fine granularity. Using smaller tasks mitigates this problem with monotasks.

Memory use As mentioned in §3.5, MonoSpark uses more memory than Spark, because data is materialized in-memory between different types of monotasks. This memory pressure can alleviated by breaking jobs into a larger number of multitasks, which results in each multitask operating on less data, and, as a result, less data needing to be kept in-memory. The monotasks design could also be augmented to include a memory manager similar to the memory manager used in Themis [28].

Caching data Disk monotasks do not write data to the buffer cache because letting the OS manage the buffer cache hurts predictability. This hurts our performance compared to Spark for some jobs, as described in §5.3. MonoSpark could leverage Spark's application-level cache to opportunistically avoid writing data to disk.

Disk scheduling The disk monotask scheduler currently balances requests across available disks, independent of load. A better strategy would consider the load on each disk in deciding which disk should write data; for example, writing to the disk with the shorter queue.

Multitask scheduling Our current implementation uses a simple multitask scheduler that assigns up to a maximum number of multitasks to each machine. This scheduler could be used to implement more sophisticated policies, e.g., to share machines between different users.

9 RELATED WORK

Performance clarity Most existing work approaches performance clarity by treating the system architecture as fixed, and either adding instrumentation or performing black-box experiments to reason about performance [4, 7, 17, 25]. Section 2.2 described existing approaches to performance clarity for data

analytics frameworks. Ongoing debate about the bottleneck for data analytics frameworks suggests that reasoning about performance remains non-trivial [22, 23, 25, 30].

Performance clarity has been studied more extensively in the context of high-performance single-server applications. Causal profiling simulates the impact of performance optimizations by measuring the relative impact on performance of slowing down concurrent code [11]. If applied to large-scale distributed systems, causal profiling could answer the type of what-if questions that we used monotasks to answer. Flux takes an architectural approach, similar to monotasks: engineers use the Flux language to write high-performance servers, which enables both bottleneck identification and performance prediction [8].

Improving analytics system performance Numerous recent efforts have improved the performance of data analytics frameworks by optimizing CPU use; e.g., by reducing serialization cost, structuring computation to more efficiently use CPU caches, taking advantage of vectorization, and more [10, 22, 23, 33]. These efforts are orthogonal to monotasks: they make workloads less CPU-bound, but do not change the fine-grained pipelining of today's multi-resource tasks.

Improving resource scheduling A variety of scheduling projects have improved on slot-based scheduling models to account for use of multiple resources [14, 16]. These schedulers use estimates of task resource use to determine how many tasks to assign to each machine. They treat the structure of the task as a black box, and do not schedule a task's access to each resource. As a result, tasks may still contend, even if the average resource use of each task is less than the amount of resources available on the machine, as discussed in §5.4. Improving resource throughput with per-resource schedulers, as is done with monotasks, has been explored in various systems that use disk schedulers to batch access and avoid seeks (e.g., Themis [28] and Impala [9]).

Granularity of pipelining Traditional data processing systems use fine-grained pipelining to stream records between operators, as in the Volcano operator model [15]. Themis [28], for example, argues for record-at-a-time pipelining, where each record is processed fully after reading, to avoid memory pressure. We do not evaluate the very large scale (TBs of data) workloads targeted by Themis. As discussed in §8, for such workloads, we anticipate borrowing their insight for reduced memory pressure: i.e., for such workloads, multi- and monotasks should act on fewer records.

Fine-grained pipelining has been revisited in the database literature to improve performance and take advantage of vectorized execution [19, 26, 37]. For example, SQL server 2012 abandoned the row-at-a-time iterator model, and instead processes a batch of (typically around 1000) rows at a time. Monotasks similarly abandons fine-grained pipelining, but with the different objective of easing users ability to reason about performance bottlenecks.

10 CONCLUSION

This paper explored a new system architecture designed to provide performance clarity. We proposed decomposing today's multi-resource tasks into smaller units of work, monotasks, that each use only one of CPU, network, or disk. This decomposition trivially enables users to understand the system bottleneck, and also allows for accurate estimates of the impact of potential system changes. Using monotasks provides performance clarity without sacrificing high performance. Performance with our implementation is comparable to Apache Spark, and using monotasks presents new opportunities for optimizations that we have not yet fully explored.

We believe that today, the primary obstacle to users who are trying to improve performance of their workloads is *not* that they have too few optimizations at their disposal, but rather that they do not know which optimization to choose. Furthermore, system bottlenecks are constantly changing, as developers work to optimize current bottlenecks and as hardware changes. As a result, optimizations that were effective a few months ago may no longer be useful. We hope that by illustrating that performance clarity can be provided as part of a system's architecture, providing performance clarity – perhaps with monotasks – will be a first-class concern in the design of new systems.

ACKNOWLEDGMENTS

We indebted to Shivaram Venkataraman, for discussions during the tiny tasks project [24] that led to the idea of breaking jobs into small, single-resource units of work, and to Max Wolffe, for helping to implement disk optimization features in early versions of MonoSpark. We thank Aurojit Panda, Eddie Kohler, and Patrick Wendell for providing helpful feedback on earlier drafts of this paper. Finally, we thank our shepherd, Miguel Castro, for helping to shape the final version of this paper. 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] Apache Spark: Lightning-Fast Cluster Computing. http://spark.apache.org/.
[2] Common Crawl. http://commoncrawl.org/.
[3] OpenBLAS: An optimized BLAS library. http://www.openblas.net/.
[4] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proc. SOSP*, 2003.
[5] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proc. NSDI*, 2017.

[6] Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org/.

[7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc. SOSP*, 2004.

[8] B. Burns, K. Grimaldi, A. Kostadinov, E. D. Berger, , and M. D. Corner. Flux: A Language for Programming High-Performance Servers. In *Proc. Usenix ATC*, 2006.

[9] Cloudera. Cloudera Impala: Open Source, Interactive SQL for Hadoop. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.

[10] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: Redefining modern analytics. *CoRR*, 2014.

[11] C. Curtsinger and E. D. Berger. COZ: Finding Code that Counts with Causal Profiling. In *Proc. SOSP*, 2015.

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

[13] P. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed Near-optimal Datacenter Transport Over Commodity Network Fabric. In *Proc. CoNext*, 2015.

[14] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proc. NSDI*, 2011.

[15] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. SIGMOD*, 1990.

[16] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-Resource Packing for Cluster Schedulers. In *Proc. SIGCOMM*, 2014.

[17] H. Herodotou. Hadoop performance models. *CoRR*, 2011.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs From Sequential Building Blocks. In *Proc. EuroSys*, 2007.

[19] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *Proc. SIGMOD*, 2014.

[20] N. McKeown. The iSLIP Scheduling Algorithm for Input-queued Switches. *IEEE/ACM Trans. Netw.*, 7(2), 1999.

[21] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at What Cost? In *Proc. Hot OS*, 2015.

[22] F. McSherry and M. Schwarzkopf. The impact of fast networks on graph analytics, part 1. http://tinyurl.com/qaw9lla.

[23] F. McSherry and M. Schwarzkopf. The impact of fast networks on graph analytics, part 2. http://tinyurl.com/q7aeajb, 2015.

[24] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The Case for Tiny Tasks in Compute Clusters. In *Proc. HotOS*, 2013.

[25] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making Sense of Performance in Data Analytics Frameworks. In *Proc. NSDI*, 2015.

[26] S. Padmanabhan, T. Malkemus, A. Jhingran, and R. Agarwal. Block oriented processing of Relational Database operations in modern Computer Architectures. In *Proc. ICDE*, 2001.

[27] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-scale Data Analysis. In *Proc. SIGMOD*, 2009.

[28] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: An I/O-efficient MapReduce. In *Proc. SoCC*, 2012.

[29] S. Ryza. How-to: Tune Your Apache Spark Jobs (Part 2). goo.gl/7gjmyfcontent_copyCopyshortURL, 2015.

[30] A. Trivedi, P. Stuedi, J. Pfefferle, R. Stoica, B. Metzler, I. Koltsidas, and N. Ioannou. On The [Ir]relevance of Network Performance for Data Processing. In *HotCloud*, 2016.

[31] UC Berkeley AmpLab. Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/, February 2014.

[32] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *NSDI*, 2016.

[33] R. Xin and J. Rosen. Project Tungsten: Bringing Spark Closer to Bare Metal. https://www.databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html, 2015.

[34] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *Proc. SIGMOD*, 2013.

[35] L. Yi, K. Wei, S. Huang, and J. Dai. Hadoop Benchmark Suite (HiBench). https://github.com/intel-hadoop/HiBench, 2012.

[36] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. NSDI*, 2012.

[37] P. Åke Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL Server Column Store Indexes. In *Proc. SIGMOD*, 2010.