Jointly Optimizing Task Granularity and Concurrency for In-Memory MapReduce Frameworks

Jonghyun Bae*§, Hakbeom Jang†§, Wenjing Jin*, Jun Heo*, Jaeyoung Jang†
Joo-Young Hwang‡, Sangyeun Cho‡ and Jae W. Lee*
*Department of Computer Science and Engineering, Seoul National University
†Department of Electrical and Computer Engineering, Sungkyunkwan University
‡Software Development Team, Memory Business, Samsung Electronics Co., Ltd.

***Abstract*—Recently, in-memory big data processing frameworks have emerged, such as Apache Spark and Ignite, to accelerate workloads requiring frequent data reuse. With effective in-memory caching these frameworks eliminate most of I/O operations, which would otherwise be necessary for communication between producer and consumer tasks. However, this performance benefit is nullified if the memory footprint exceeds available memory size, due to excessive spill and garbage collection (GC) operations. To fit the working set in memory, two system parameters play an important role: number of data partitions ($N_{partitions}$) specifying task granularity, and number of tasks per each executor ($N_{threads}$) specifying the degree of parallelism in execution. Existing approaches to optimizing these parameters either do not take into account workload characteristics, or optimize only one of the parameters in isolation, thus yielding suboptimal performance. This paper introduces WASP, a workload-aware task scheduler and partitioner, which jointly optimizes both parameters at runtime. To find an optimal setting, WASP first analyzes the DAG structure of a given workload, and uses an analytical model to predict optimal settings of $N_{partitions}$ and $N_{threads}$ for all stages based on their computation types. Taking this as input, the WASP scheduler employs a hill climbing algorithm to find an optimal $N_{threads}$ for each stage, thus maximizing concurrency while minimizing data spills and GCs. We prototype WASP on Spark and evaluate it using six workloads on three different parallel platforms. WASP improves performance by up to 3.22× and reduces the cluster operating cost on cloud by up to 40%, over the baseline following Spark Tuning Guidelines and provides robust performance for both shuffle-heavy and shuffle-light workloads.**

I. Introduction

Recently, the in-memory processing paradigm has been embraced by big data analytics frameworks, such as Apache Spark [1] and Ignite [2]. For example, Spark is reported to achieve much higher performance than Hadoop MapReduce [3] by up to 100× for those workloads with frequent data reuse [1]. Resilient Distributed Datasets (RDDs) [4] are the basic data structure of Spark, which allow a programmer to cache intermediate data in fast memory instead of writing them to slow disks. The arduous task of providing fault tolerance, data distribution, load balancing, and scheduling is handled by these frameworks; thus, the programmer can focus on the logic of the program instead of parallel execution details.

However, these in-memory data analytics frameworks are prone to severe performance degradation if the system runs out of memory. In case of Spark, the main memory is shared by cached RDDs as well as other heap objects of Java Virtual Machine (JVM). Once memory is nearly full, it triggers a large number of disk spills (i.e., serializing and storing JVM objects to the local disk) and GCs, which nullify the performance benefit of in-memory processing. Shuffle operations are known to be particularly susceptible to this slowdown [5], [6], thus requiring careful optimization in memory usage.

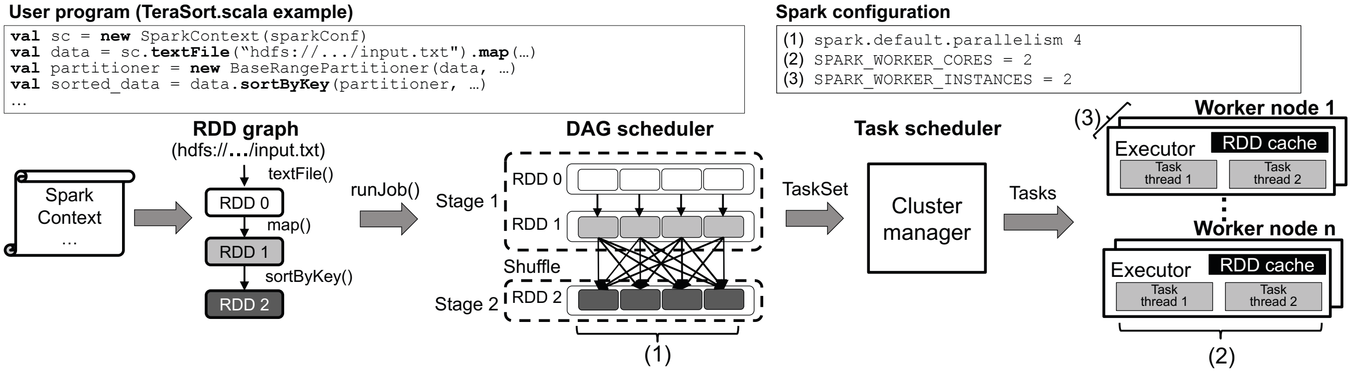
There are two runtime parameters that have profound impact on memory usage in Spark and hence its performance: task granularity and degree of parallelism. The first parameter, denoted by $N_{partitions}$, specifies how many data partitions are created from a single RDD, where a single task processes one partition. If this parameter is set too low (i.e., too few partitions), it can cause excessive spills and GCs by increasing memory pressure; if too high, it incurs significant overhead for scheduling and shuffle operations. The second parameter, denoted by $N_{threads}$, specifies how many threads are allocated to a single executor, where each thread executes one task at a time. Setting this parameter too low yields suboptimal performance due to underutilization of processing elements; setting it too high degrades performance due to increased memory pressure and other resource contentions. Thus, these parameters should be carefully tuned by considering both hardware resources and workload characteristics to achieve optimal performance.

Existing proposals to address this problem either do not take into account workload characteristics or optimize a single parameter in isolation, to yield suboptimal performance. For example, Spark Tuning Guidelines [7], [8], [9] recommend the user should set both $N_{partitions}$ and $N_{threads}$ as a function of the number of processor cores without considering workloads. Furthermore, a single set of $N_{partitions}$ and $N_{threads}$ is used by the entire program even if optimal parameter settings can vary widely among RDDs in a single program. Existing optimizers for these parameters optimize either $N_{partitions}$ [10], [11], [12] or $N_{threads}$ [5], [13], [14] in isolation, but not both. This leads to suboptimal performance as the two parameters are not independent, and selection of one parameter directly affects the optimal setting for the other.

In this paper we argue for jointly optimizing both $N_{partitions}$ and $N_{threads}$ at runtime by considering both workload characteristics (RDD graph and input data) and the

[§] These authors contributed equally to this work.

978-1-5386-2715-0/17/$31.00 ©2017 IEEE


User program (TeraSort.scala example)

```
val sc = new SparkContext(sparkConf)
val data = sc.textFile("hdfs://.../input.txt").map(…)
val partitioner = new BaseRangePartitioner(data, …)
val sorted_data = data.sortByKey(partitioner, …)
…
```

Spark configuration

```
(1) spark.default.parallelism 4
(2) SPARK_WORKER_CORES = 2
(3) SPARK_WORKER_INSTANCES = 2
```

Fig. 1. Spark execution model

execution environment. To realize this, we propose *WASP*, a workload-aware task scheduler and partitioner for in-memory MapReduce frameworks. WASP first analyzes the DAG structure of a given workload and uses an analytical model that predicts an optimal setting of $N_{partitions}$ and $N_{threads}$ for each stage based on both workload and platform parameters. Taking this as input, the GC-aware task scheduler further optimizes $N_{threads}$ during task execution via runtime monitoring of individual tasks. Thus, WASP maximizes CPU utilization while minimizing the overhead of data spills and GCs.

We prototype WASP on Spark and evaluate it using six applications from the HiBench benchmark suite [15] on two native parallel platforms and a cluster of virtual machines (VMs) on Amazon Elastic Compute Cloud (EC2) [16]. WASP improves the performance by up to 3.22× over the baseline configuration following Spark Tuning Guidelines [7], [8] with a geomean speedup of 1.74× on a 4-node native cluster with 64 fat cores (Intel Xeon) and 1.56× on a single-node machine with 64 thin cores (Intel Xeon Phi). These numbers fall within 6.3% and 12.3% of the static optimal configuration based on an impractical three-dimensional exhaustive search in the parameter space, respectively. Furthermore, WASP achieves a geomean (maximum) speedup of 1.31× (1.67×) for a 64-node VM cluster on Amazon EC2, which translates to a reduction of the operating cost by 24% (40%) over the baseline.

In summary, this paper makes the following contributions:

- Introduction of an effective analytical model that predicts optimal $N_{partitions}$ and $N_{threads}$ values for in-memory MapReduce frameworks
- Design and implementation of the WASP scheduler on Spark, which maximizes task concurrency (and hence CPU utilization) at runtime, while minimizing disk spills and GCs
- Detailed evaluation and analysis of WASP performance on two native parallel platforms (4-node cluster with 64 fat cores (Intel Xeon) and single-node machine with 64 thin cores (Intel Xeon Phi [17])) and one virtual platform (64-node VM cluster with 256 fat cores (Intel Xeon))

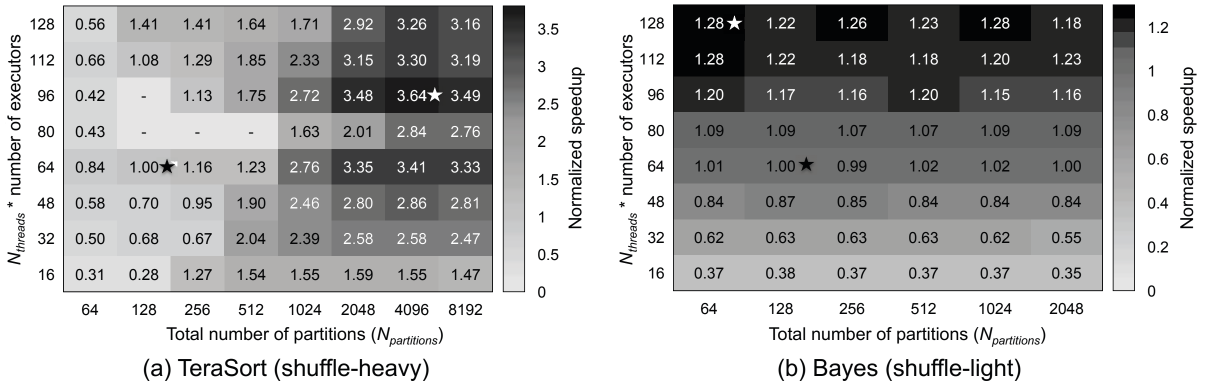
The rest of this paper is organized as follows: Section II presents backgrounds and motivates this work. Section III describes the design and implementation of WASP. Section IV and V provide our evaluation setup and performance results, followed by discussion of related work in Section VI. Finally, Section VII summarizes the conclusion.

II. Background and Motivation

A. Spark Execution Model

Spark employs a dataflow execution model, where a sequence of operations are performed on *resilient distributed datasets (RDDs)* [4]. RDDs are the primary data abstraction for Spark, which provide fault tolerance as well as efficient in-memory caching in a distributed environment. Spark operations are divided into two categories: *transformations* and *actions*. A transformation defines a new RDD to add to a directed acyclic graph (DAG) representing the execution plan (called *lineage* in a Spark term), which is executed lazily. An action is an operation that exports data to a storage system or returns a result to the main (driver) program, such as saving an RDD to a file and collecting an RDD to print its contents to the console. Every action in a user program will create a Spark *job*, which may consist of multiple transformations. Only when an action is called, Spark starts execution of the job made up of the DAG.

Fig. 1 illustrates the Spark execution model using TeraSort from HiBench benchmark suite [15] as an example. A Spark application consists of a driver program and a pool of executors distributed over multiple cluster nodes. The user program typically begins with creating a Spark context by invoking `SparkContext()`. In response to an action (not shown in Fig. 1), a *job* is created with a sequence of transformations recorded so far. The DAG scheduler receives this job and splits it into multiple *stages* at every shuffle boundary. A shuffle is an operation that exchanges data between a pair of mapper and reducer stages, where the mapper stage writes its output into the local disk and the reducer stage pulls it from remote disks over the network. An RDD is partitioned into a set of immutable blocks (or partitions) and distributed across worker nodes. For every stage of the Spark job, a set of tasks are created, which apply the same lineage of transformations to different RDD partitions. These task sets are passed to a task scheduler which schedules tasks to a collection of executors running on multiple worker nodes.



(a) TeraSort (shuffle-heavy)

(b) Bayes (shuffle-light)

Fig. 2. Normalized speedups over the baseline with varying parameter settings of $N_{partitions}$ and $N_{threads}$

Spark Memory is the memory pool managed by Spark, whose size is controlled by the parameter `spark.memory.fraction`. By default in Spark 1.6.1 this parameter is set to 75% of the JVM heap space; the remaining space is User Memory, which the user can allocate for any auxiliary data structures. Spark Memory is divided into storage memory and execution memory, which are equally sized initially. The former is used for RDD caching, which is particularly useful for iterative algorithms featuring frequent data reuse, and the latter for Spark computations. Note that the boundary between storage memory and execution memory is soft, which means that one of the two regions can claim the other's space dynamically if it is underutilized.

However, the benefit of in-memory RDD caching and computation is nullified due to expensive data *spills* if Spark Memory becomes (nearly) full. A spill occurs when Spark runs out of executor memory. Once a spill is triggered by a task, Spark serializes the partially generated output RDD for the task and writes it to the local disk. Spills are expensive operations (more detailed analysis available in Section II-B), and should be avoided to achieve robust performance. To fit the working set in memory, two runtime parameters play an important role: $N_{partitions}$ and $N_{threads}$. The first parameter is $N_{partitions}$ (`(1) spark.default.parallelism` in Fig. 1), which determines how many partitions will be created from a single RDD. This parameter controls the *granularity* of a task, and hence the memory footprint of a single task. The second parameter is $N_{threads}$ (`(2) SPARK_WORKER_CORES` in Fig. 1), which determines how many tasks are processed concurrently at each executor. This parameter controls the *concurrency* of tasks, and hence the memory footprint at an executor (and node) level. Therefore, these parameters should be tuned carefully to achieve robust performance.

B. Finding Optimal $N_{partitions}$ and $N_{threads}$

To ensure high utilization of cluster resources, Spark provides a tuning guideline for both parameters: $N_{partitions}$ and $N_{threads}$. The guideline says, "In general, we recommend 2-3 tasks per CPU core in your cluster" [7]. Thus, $N_{partitions}$ is set to be 2-3 times of the total number of CPU cores. $N_{threads}$ controls the degree of parallelism (concurrency) in task execution, which is set by `SPARK_WORKER_CORES`. This parameter is set to use "all available cores" by default [8].

Although simple, this guideline has several problems. First of all, both parameters are derived solely from a single cluster parameter (i.e., number of CPU cores), which fail to yield robust performance over a variety of workloads with different characteristics. To confirm this point, we measure speedups with varying both parameters on a 4-node Intel Xeon cluster (having 64 cores in total) for two Spark applications from HiBench: TeraSort and Bayesian Classification (Bayes). The details of the cluster setup are available in Table I in Section IV. $N_{partitions}$ on X axis ranges from 64 to 2048 (8192 for TeraSort); the total number of worker threads ($16{\times}N_{threads}$) on Y axis from 16 to 128.

Fig. 2 shows the speedups normalized to the baseline configuration that follows the Spark tuning guideline. We use a heat map to visualize speedups, and darker box is higher speedup. The baseline configuration is marked by a black star (★), and the best performing one by a white star (☆). For both workloads the baseline configuration is clearly suboptimal—especially for TeraSort demonstrating a 3.64× performance gap between the best and the baseline configurations. Besides, the best configurations for the two workloads are far away from each other as they have different workload characteristics. For example, the best configuration for Bayes on the upperleft corner yields worse performance than the baseline (0.56×) for TeraSort. Thus, without considering workload characteristics, it is difficult to find an optimal setting for these parameters.

According to our analysis, the large performance gap for TeraSort is attributed to excessive memory spills in the baseline configuration. Fig. 3(a) shows an execution time breakdown of the reduce stage (Stage 2 in Fig. 1), which suffers the most slowdown, with varying $N_{partitions}$ from 128 to 4096. If $N_{partitions}$ is less than 1024, the spill overhead dominates the execution time as the working set does not fit in the execution memory of Spark. The spill time (when $N_{partitions}$ is 128) is further broken down in Fig. 3(b), which shows the full GC time dominates it. This happens because a spill operation takes a considerable amount of time due to

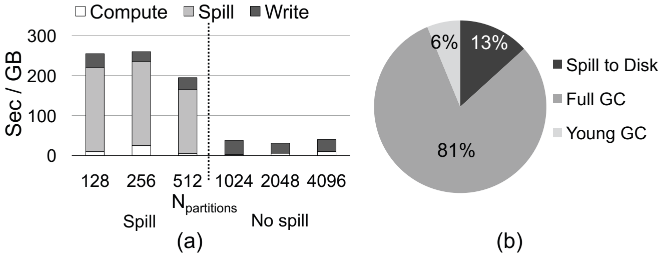


Fig. 3. TeraSort performance analysis: (a) execution time breakdown of the reduce stage; (b) spill time breakdown when $N_{partitions}$ is 128

the overhead of serialization and disk writes, but the spilled object in JVM's old-generational heap space will not be freed until the entire spill operation is completed. Meanwhile, Spark keeps creating small heap objects, some of which get tenured from a young-generational heap to the old-generational heap to trigger frequent full GCs. Thus, in this case, it is highly desirable to set $N_{partitions}$ to a value large enough (i.e., $\geq$1024) to eliminate spills and achieve good performance.

To address this problem, *workload-aware* optimizers for $N_{partitions}$ and $N_{threads}$ have been recently proposed for in-memory MapReduce frameworks [10], [11], [12], [5], [13], [14]. However, these optimizers tune either $N_{partitions}$ [10], [11], [12] or $N_{threads}$ [5], [13], [14] in isolation, but not both. This leads to suboptimal performance. For example, in Fig. 2, starting from the baseline configuration (marked by ★), TeraSort benefits more from optimizing $N_{partitions}$, whereas Bayes more from optimizing $N_{threads}$. By optimizing only one of the two parameters, Spark cannot achieve good performance for both workloads. Note that the selection of one parameter directly affects the optimal setting for the other as the two parameters are not independent. Thus, it is highly desirable to jointly optimize both parameters in a single unified framework to make best use of cluster resources without causing excessive spills and GCs.

III. Workload-Aware Scheduler and Partitioner

This section presents the design and implementation of WASP (**W**orkload-**A**ware task **S**cheduler and **P**artitioner). WASP is a low-cost runtime framework that jointly optimizes both $N_{partitions}$ and $N_{threads}$ for in-memory MapReduce frameworks, to provide robust performance over a wide variety of workloads.

A. Overview

Fig. 4 shows the overall structure of WASP, which consists of two parts shaded in gray. The first component is an analytical model that predicts an optimal setting of $N_{partitions}$ and $N_{threads}$, derived from both workload and platform parameters (Section III-B). This model is executed whenever a new job is created, and the predicted optimal parameter values are annotated in the RDD graph before task execution begins.

The second component is GC-aware task scheduler, which further optimizes $N_{threads}$ during task execution (Section III-C). The proposed scheduler employs a feedback loop between the master node and worker nodes to find the maximum number of worker threads that can run concurrently without causing excessive GCs. In this way WASP can find a near-optimal parameter setting at a low-cost without performing an exhaustive search for each stage, whose cost would be prohibitive.

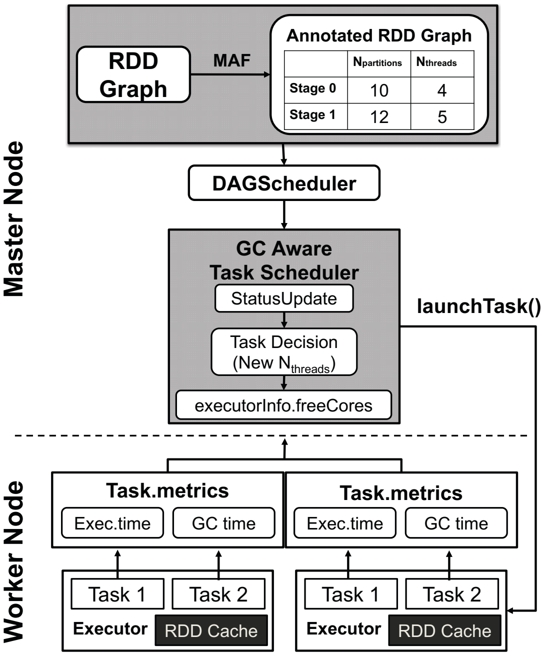


Fig. 4. WASP overview

B. Analytical Model

When a new job is created, the Spark runtime has complete information about the RDD graph and input file(s) for the job. Using this information WASP executes an analytical model to predict an optimal setting for $N_{partitions}$ and $N_{threads}$ for each stage before invoking the DAG scheduler. The analytical model takes a two-pass approach. The first pass predicts an initial setting for the two parameters (denoted by $N_{partitions(init)}$ and $N_{threads(init)}$). These values are used as input to the second pass, which employs a gradient search algorithm in a two-dimensional parameter space using a simple analytical model for stage execution time. As a result of executing the second pass, we get an optimal parameter pair for each stage that likely yields the minimum execution time.

First Pass: Calculating $N_{partitions(init)}$ and $N_{threads(init)}$. The relationship between the total number of partitions ($N_{partitions}$) and the number of threads per executor ($N_{threads}$) for the k-th stage is described as follows:

$$N_{threads(k)(init)} = max\left(\frac{Execution\ memory}{\frac{RDD\ size_{\ k}}{N_{partitions(k)(init)}}}, 1\right) \qquad (1)$$

where $RDD\ size_{\ k}$ is the maximum RDD size in the lineage of the RDD graph of the k-th stage, and $Execution\ memory$ is the size of the available execution memory for a single

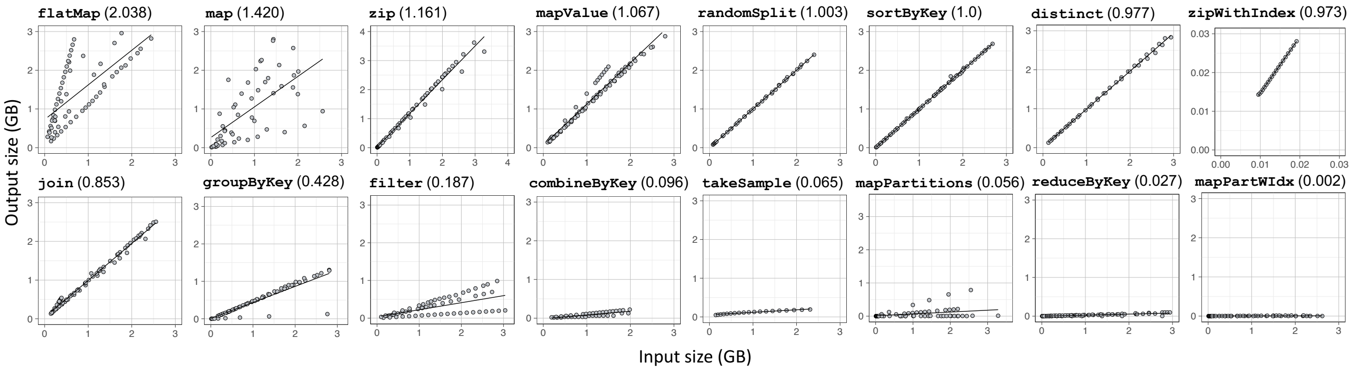


Fig. 5. Memory Amplification Factors (MAF) for transformation functions

executor. The first term in the $max()$ function calculates how many tasks can be accommodated without causing a spill if each task requires execution memory of the maximum RDD partition size in the stage.

Assuming the k-th stage has a lineage of n_k RDDs whose transformation functions are f_1^k, f_2^k, ..., $f_{n_k}^k$, $RDD\ size_{\,k}$ is given as follows:

$$RDD\ size_{\,k} = Input\ size_{\,k} \times \max_{1\leq i\leq n_k} (\prod MAF(f_i^k)) \quad (2)$$

where $Input\ size_{\,k}$ is the input RDD size for the stage and $MAF(f_i^k)$ is the *memory amplification factor* of the function i. The memory amplification factor (MAF) of a transformation function is defined as the ratio of the output RDD size to the input RDD size after the transformation is applied. Fig. 5 shows the MAF values for widely used transformation functions. MAFs are estimated by performing linear regression with various workloads (HiBench [15], SparkBench [18], spark-perf [19], BigDataBench [20]) and inputs. Unlike other transformation functions, outputs of various sizes are produced even if the input size is similar. `map()` and `flatMap()` generate values in a one-to-one, many-to-one, one-to-many, and many-to-many manner, depending on user definition.

Note that these values are purely software parameters and *platform-independent*. Thus, there is no need to measure these values on individual platforms. The terms of maximum by $MAF(f_i^k)$ in (2) finds the maximum RDD size at the given stage. The maximum RDD may be obtained in the middle (not the end) of the stage as $MAF(f_i^k)$ can be less than 1.

Now the only missing parameter in (2) is $Input\ size_{\,k}$, which is estimated as follows:

$$Input\ size_{\,k} = Input\ File\ Size \times \prod_{i=1}^{k-1} (\prod_{j=1}^{n_j} MAF(f_j^i)) \quad (3)$$

where $Input\ File\ Size$ is the input file size for the job. Note that the second term as MAF has the value of multiplying MAF values of all transformation functions from Stage 1 through $k-1$. We use this formula to estimate the input RDD size for Stage k. While this size can be measured more precisely at runtime, re-partitioning the input RDD right before Stage k has very high overhead, so we use an estimation model ahead of execution to avoid this overhead.

Finally, assuming CPU is the performance bottleneck [21], we set $N_{threads(init)}$ to be the number of available physical cores per executor. Then, using (1), we obtain $N_{partitions(init)}$.

Second Pass: Calculating $N_{partitions}$ and $N_{threads}$ via Gradient Search. In the second pass we first introduce a simple analytical model of the execution time of Stage k and use it to find an optimal setting of $N_{partitions}$ and $N_{threads}$ that minimize the execution time. For this we apply a gradient search in the two-dimensional parameter space with $N_{partitions(init)}$ and $N_{threads(init)}$ (from the first pass) as a starting point. The execution time of Stage k is estimated as follows:

$$ExecTime_{\,k} \propto \frac{RDD\ size_{\,k}}{N_{partitions(k)}} \times \alpha_{\,k} \times \frac{N_{partitions(k)}}{N_{threads(k)}} \quad (4a)$$

$$\propto RDD\ size_{\,k} \times \frac{\alpha_{\,k}}{N_{threads(k)}} \quad (4b)$$

where

$$\alpha_{\,k} = max\left(\frac{\frac{RDD\ size_{\,k}}{N_{partitions(k)}} \times N_{threads(k)}}{Execution\ memory}, 1\right) \quad (5)$$

According to (4a), the execution time is proportional to a product of the three terms. The first term quantifies the granularity of a task using the partition size as a proxy metric. The second term ($\alpha_{\,k}$) captures the cost of spills. If the estimated memory footprint for an executor in the numerator exceeds the size of the execution memory in the denominator, data spills will prolong the execution time in proportion to the ratio of the two; if not, $\alpha_{\,k}$ in the (5) has no impact. Finally, the third term calculates how many rounds it takes to execute the stage, assuming each round completes $N_{threads}$ tasks. WASP uses this model to perform a gradient search that quickly finds a local optimum of the execution time. Starting at ($N_{partitions(k)(init)}$, $N_{threads(k)(init)}$),

TABLE I
Setup for three evaluation platforms

	Native cluster	Knights Landing (KNL)	Amazon Web Services EC2
	System configurations		
Server	Dell R730 × 4 (+ 1 master)	SuperServer 5038k-i	m4.xlarge VM × 64 (+ 1 master)
CPU	Intel Xeon CPU E5-2640v3 8 cores @ 2.60GHz × 2 sockets	Intel Xeon Phi (Knights Landing) 7210 64 cores @ 1.30GHz	Intel Xeon CPU E5-2676v4 4 cores @ 2.4GHz
Memory	16GB DDR4 × 8	16GB MCDRAM & 32GB DDR4 × 6	16GB
Disk	NVMe SSD 1.6TB	NVMe SSD 1.6TB	EBS 100GB
Network	40Gbps InfiniBand	1Gbps Ethernet	10Gbps Ethernet
OS	Ubuntu 14.04 LTS	CentOS Linux release 7.2.1511	Ubuntu Server 16.04 LTS
	Spark configurations		
# of worker nodes	4	1	64
# of executors / node	4 (16 executors in total)	4 (4 executors in total)	1 (64 executors in total)
Spark memory / executor	5GB	20GB	10GB
$N_{threads}$	4 (64 threads in total)	16 (64 threads in total)	4 (256 threads in total)
$N_{partitions}$	128	128	512

Algorithm 1 Determining analytical $N_{partitions}$ and $N_{threads}$

```
INPUT: RDD size, # of cores, size of executor memory
OUTPUT: <$N_{part(k)}$, $N_{thd(k)}$>
Estimate $N_{part(k)(init)}$ and $N_{thd(k)(init)}$ from (1)
Set initial $ExecTime_k$ from (4a)
<$N_{part(k)}$, $N_{thd(k)}$> = <$N_{part(k)(init)}$, $N_{thd(k)(init)}$>
$minExecTime$ = $ExecTime_k$
do
    for rotate 8 directions (closest neighboring values) do
        calculate $ExecTime_{new}$
        if $ExecTime_{new} < minExecTime$ then
            $minExecTime$ = $ExecTime_{new}$
            <$N_{part(k)}$, $N_{thd(k)}$> = <$N_{part(k)(new)}$, $N_{thd(k)(new)}$>
        end if
    end for
while $minExecTime$ / $ExecTime_k < 0.9$
```

WASP compares its execution time with eight neighboring configurations in the two-dimensional parameter space, enclosed by ($N_{partitions(k)(init)}$/2, $N_{threads(k)(init)}$/2) and ($N_{partitions(k)(init)}$×2, $N_{threads(k)(init)}$×2). If there is no other configuration whose execution time is estimated to be $>$10% smaller than the one at the center, the search process is terminated. If not, the configuration with the smallest execution time becomes the new center, and WASP repeats this search process until either of the two termination conditions is satisfied. Note that a large $N_{partitions}$ increases the cost of shuffles. Thus, WASP finds the minimum number of $N_{partitions}$ that do not cause GCs and spills. Equation (5) shows that the value of $N_{partitions}$ is upper-bounded by $RDDsize_k \times N_{threads(k)}$/*Execution memory* to prevent the situation of creating too many partitions. Algorithm 1 summarizes the two-pass procedure of predicting $N_{partitions}$ and $N_{threads}$.

C. Garbage Collection-Aware Task Scheduler

Once the analytical model in Section III-B estimates the $N_{partitions}$ and $N_{threads}$ for each stage, these parameters are passed to the DAG scheduler, which is responsible for partitioning the Spark job into stages of tasks. Then the task set is passed to the task scheduler, and executors start executing multiple tasks in parallel. However, these parameters are derived from a simple analytical model, and the runtime environment may vary widely to make them suboptimal. Note that the design objective of WASP is to maximize the degree of parallelism without causing excessive spills and GCs, hence maximizing CPU utilization.

Thus, WASP employs a *GC-aware task scheduler* to further optimize $N_{threads}$ at runtime. While changing $N_{partitions}$ at runtime requires costly re-partitioning of the entire RDD adjusting $N_{threads}$ can be easily done. WASP exploits a thread pool provided by Spark, to allow us to adjust the number of threads easily as there are no data dependences among concurrent tasks, obviating the need for such synchronization. To realize this, WASP creates a feedback loop between the master and worker nodes. Upon completion of the last task of a round (with $N_{threads}$ tasks, initially), the task scheduler on the master node monitors the fraction of execution time spent for GC operations. If the GC time is less than a threshold (20% of total execution time by default in Section V-D), $N_{threads}$ is multiplied by a factor of 2 until it reaches the upper-bound of $N_{threads}$, which is set to 2× of the total number of (logical) cores. If (1) the GC time goes above the threshold, or (2) the execution time itself increases by more than 20%, $N_{threads}$ is decrementing by one at the next round. Note that the second condition is included to detect any straggler task [21]. The thread count keeps decrementing until the GC time goes below the threshold again or the stage is finished.

IV. Methodology

To demonstrate the effectiveness of WASP, we evaluate it in three different execution environments as shown in Table I. The same versions of software packages are used for all three: Hadoop version 1.2.1, Spark version 1.6.1, and OpenJDK version 1.7.0_111. The default settings are used unless noted otherwise. The platform setups are summarized as follows:

Native 4-node cluster with 64 fat cores. A single node has two Intel Xeon E5-2640v3 CPUs running at 2.60GHz, each of which has 8 physical cores (16 logical cores via Hyper-Threading). The node has a 1TB hard disk for the host OS (Ubuntu 14.04 LTS) and a 1.6TB NVMe SSD for HDFS and Spark shuffle operations. Master and worker nodes

TABLE II
WORKLOAD CHARACTERISTICS

Workloads		**Domain**	**Data Size**		
			Cluster	**Intel KNL**	**AWS EC2**
Shuffle-heavy	TeraSort	Sorting	128GB	80GB	1TB
	PageRank	Web search	pages: 25M iterations: 3	pages: 15M iterations: 3	pages: 100M iterations: 3
	Sort	Sorting	128GB	80GB	1TB
Shuffle-light	WordCount	Text processing	240GB	128GB	5TB
	Bayesian Classification	Machine learning	pages: 20M classes: 20K	pages: 10M classes: 10K	pages: 160M classes: 160K
	Kmeans	Machine learning	samples: 200M samples / input: 30M max iterations: 5	samples: 150M samples / input: 30M max iterations: 5	samples: 400M samples / input: 60M max iterations: 5

are connected by 40Gbps InfiniBand links. We choose to create four executors per node as this configuration yields the best out-of-the-box performance; we observe performance degradation with more executors on this cluster [22]. To fully utilize CPU cores, each executor runs on four physical cores.

Native manycore machine with 64 thin cores. We also evaluate WASP on Intel's Knights Landing (KNL) machine [17]. The CPU model is Intel Xeon Phi 7210 running at 1.30GHz, which has 64 physical cores (and 256 logical cores via 4-way Hyper-Threading). KNL features a near-far memory system composed of wide in-package 16GB MCDRAM and narrow off-package 192GB DDR4 DRAM. We use MCDRAM in the default cache mode.

Virtual 64-node cluster on Amazon Web Services EC2. We also evaluate the robustness and scalability of WASP in a cloud environment by using a virtualized 64-node Spark cluster on Amazon EC2. We use the m4.xlarge VM image and create one executor per node (i.e., 64 executors in total). Each executor is allocated four threads with a 10GB of Spark Memory.

For workloads we use six applications from Intel's HiBench suite 5.0 [15]. Among the 10 Spark applications in HiBench, three SQL applications (Join, Scan, and Aggregate) are excluded because they process a SQL query using `HiveContext`, instead of `SqlContext`, in an opaque manner; if the query was processed in a Spark SQL context, WASP would be readily applicable to those applications. We also exclude Sleep as it is a micro-benchmark with no real computations.

Table II summarizes workload characteristics and inputs for the six applications. We classify them into two categories based on the amount of shuffled data: shuffle-heavy and shuffle-light. For Intel KNL we use smaller inputs than those for the 4-node cluster, which would generate excessive task failures for the baseline configuration. We take an average of three measurements for every data point except for the KNL machine, for which we take the *minimum* of the three measurements as KNL suffers frequent task failures causing wide variations of execution time.

We compare WASP to the following four designs:

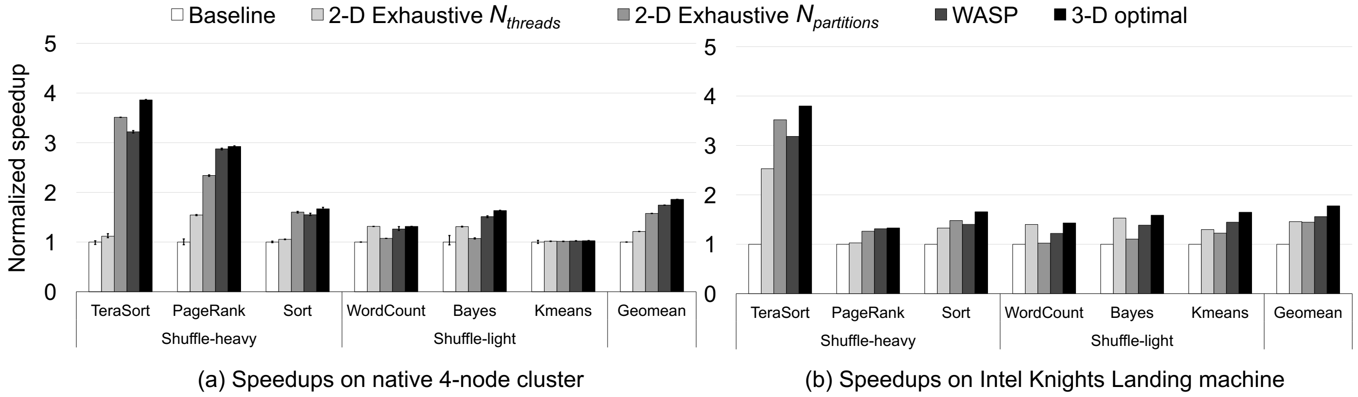
- **Baseline.** This is the out-of-the-box Spark following Spark Tuning Guidelines [7]. Thus, $N_{threads}$ is set to fully utilize the available physical cores, and $N_{partitions}$ to run two tasks per CPU core as shown in Table I.
- **2-D Exhaustive** $N_{threads}$**.** We perform a two-dimensional exhaustive search (stage, $N_{threads}$) to find the best-performing $N_{threads}$ for each stage and use the same $N_{partitions}$ as in the baseline. This configuration yields the *theoretical maximum performance* for the optimizers that tune $N_{threads}$ in isolation [5], [13], [14].
- **2-D Exhaustive** $N_{partitions}$**.** Again, we perform a two-dimensional exhaustive search (stage, $N_{partitions}$) to find the best-performing $N_{partitions}$ for each stage and use the same $N_{threads}$ as in the baseline. This configuration yields the *theoretical maximum performance* for the optimizers that tune $N_{partitions}$ in isolation [10], [11], [12].
- **3-D optimal.** We perform a three-dimensional exhaustive search (stage, $N_{partitions}$, $N_{threads}$) to find the best-performing pair of $N_{partitions}$ and $N_{threads}$ for each stage. This number serves as the (impractical) *theoretical maximum performance* that can be achieved by optimizing these two parameters.

V. EVALUATION

A. Results on Native 4-node Cluster

Fig. 6(a) compares the speedups of the five designs in Section IV, normalized to the baseline. WASP achieves a geomean speedup of 1.74×, with a maximum speedup of 3.22×. WASP outperforms 2-D Exhaustive $N_{partitions}$ and $N_{threads}$ by 1.11× and 1.44×, respectively, which represent the performance upper bound of optimizing one parameter in isolation. More importantly, regardless of the workload type, WASP achieves robust performance comparable to the best of both 2-D Exhaustive $N_{partitions}$ and 2-D Exhaustive $N_{threads}$. Following is the discussion of both types of workloads in greater details.

Shuffle-heavy workloads. The first three applications in Fig. 6(a) are shuffle-heavy workloads, which exchange a large volume of data between stages. For example, both TeraSort and Sort use a `sortByKey` transformation, which shuffles the entire RDD. In PageRank a shuffle is preceded by transformations that amplify memory footprint (i.e., MAF>1), such as `flatMap` and `map`, to further increase the overhead. Since these shuffle-heavy workloads usually take large memory footprint, their performance is more sensitive to a change in



(a) Speedups on native 4-node cluster

(b) Speedups on Intel Knights Landing machine

Fig. 6. Normalized speedups for six workloads on two native platforms: native 4-node cluster and single-node manycore platform (Intel KNL)

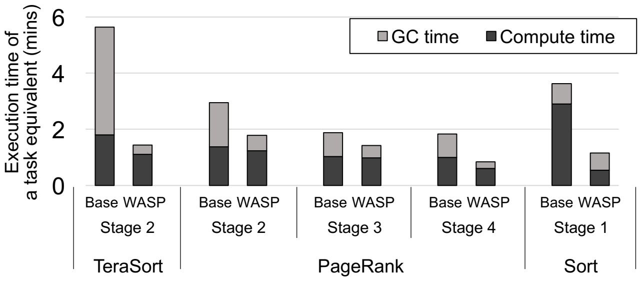


Fig. 7. Execution time breakdown of the reduce stage for shuffle-heavy workloads; each bar shows the average execution time of equivalent task(s) processing a 1/128 of the RDD.

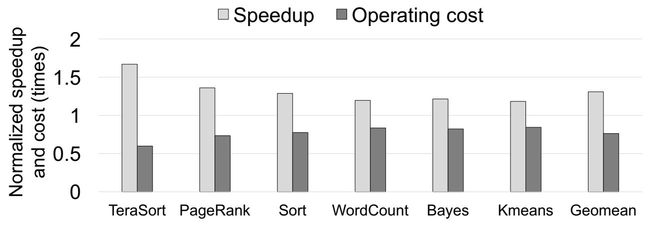


Fig. 8. Normalized speedups and operating cost on cloud for 64-node Amazon EC2 cluster

$N_{partitions}$. Thus, optimizing $N_{threads}$ alone yields relatively poor performance; even with a stage-wise exhaustive search 2-D Exhaustive $N_{threads}$ achieves only a 1.22× geomean speedup for the three applications. In contrast, WASP achieves a 2.43× geomean speedup, which is comparable to 2-D Exhaustive $N_{partitions}$ (with a 2.36× geomean speedup), but without requiring a costly exhaustive search.

To identify the sources of improvement, Fig. 7 shows an execution time breakdown of the reduce stages for the three workloads. The primary source of performance improvement with WASP is a reduction in the amount of spilled data, which eliminates most of long GCs. As a result, WASP reduces the GC time of TeraSort 91%, for example. Besides, the compute time is also reduced by 37% as WASP further increases task concurrency by maximizing $N_{threads}$ while maintaining a low overhead of GCs. The benefits of optimizing $N_{threads}$ are most pronounced in Sort, for which the compute time is reduced by 81%. For PageRank, WASP effectively reduce both compute and GC times for Stage 2 through 4, which dominate the total execution time.

Shuffle-light workloads. The next three applications in Fig. 6(a) are classified as shuffle-light. These workloads consist of several Spark jobs, most of which have a single stage. The overhead of shuffles is small for them, as a shuffle is triggered by a summarizing function with low MAF (e.g., `reduceByKey`) to reduce the volume of exchanged data. Therefore, these workloads are characterized by very low spill overhead, and optimizing $N_{threads}$ has much greater impact on performance. As a result, 2-D Exhaustive $N_{partitions}$ yields only a marginal performance gain with a 1.05× geomean speedup. In contrast, WASP achieves more substantial performance improvement with a geomean speedup of 1.25×. This number is comparable to 2-D Exhaustive $N_{threads}$ with a 1.21× geomean speedup. Note that the geomean speedup of 3-D Optimal is 1.31×, which is not far off from WASP. For Kmeans all five designs demonstrate similar performance as the baseline configuration works well.

B. Results on 64-core Intel Knights Landing Machine

Fig. 6(b) shows the performance speedups on a 64-core Intel KNL platform. For the baseline the default parameters of $N_{partitions}$ and $N_{threads}$ in Table I are used except for TeraSort and Sort. Since these two applications do not run to completion with the baseline configuration due to excessive task failures, we use higher $N_{partitions}$'s to keep the number of task failures reasonable. More specifically, $N_{partitions}$ is set to 288 for TeraSort and 224 for Sort for the baseline. Preventing excessive task failures is crucial to achieve robust performance on KNL.

Overall, WASP also works well on the single-node manycore system with a geomean speedup of 1.56× and a maximum speedup of 3.18×. WASP achieves comparable performance to the best of both 2-D Exhaustive $N_{partitions}$ and 2-D Exhaustive $N_{threads}$, whose geomean speedups are 1.45× and 1.46×, respectively. Like the 4-node cluster in Section V-A, optimizing $N_{partitions}$ ($N_{threads}$) is more important

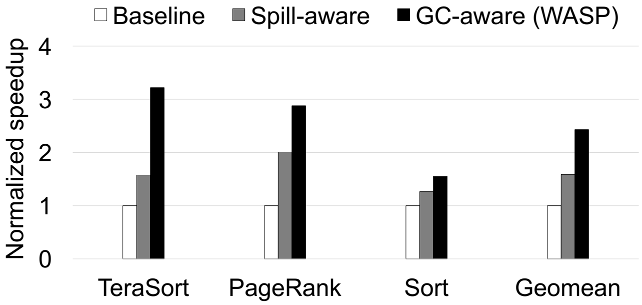


Fig. 9. Spill-aware scheduler vs. GC-aware scheduler

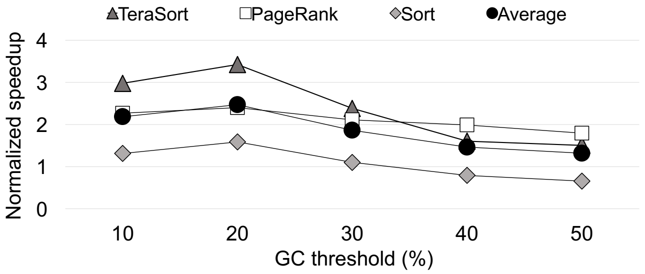


Fig. 10. Performance change with varying GC threshold. All numbers are normalized to the baseline.

for shuffle-heavy (shuffle-light) workloads. Thus, optimizing one of the two parameters in isolation is clearly suboptimal. In contrast, WASP adapts to the workload without employing a costly exhaustive search. Furthermore, WASP effectively eliminates most of task failures, thus improving performance.

C. Results on 64-node Amazon EC2 Cluster

Finally, we evaluate WASP on a 64-node Amazon EC2 cluster. Using this platform we aim to characterize the performance robustness and scalability of WASP in a virtualized execution environment at a larger scale. Besides, we quantify the cost reduction in cloud brought by the improved performance, which we calculate from the bills charged by Amazon.

Fig. 8 shows both performance improvements and cost savings using a virtualized Spark cluster with 64 worker nodes. WASP shows a geomean speedup of 1.31× with a maximum speedup of 1.67× for TeraSort. This translates to an average reduction of the Spark cluster operating cost on cloud by 24% over the baseline, with a maximum reduction of 40%.

D. Performance Analysis

Spill-aware scheduler vs. GC-aware scheduler. It is our design decision for WASP to adopt *GC-aware* scheduling at runtime, instead of spill-aware scheduling. Our finding is that excessive GCs are the primary factor to slow down Spark applications at high memory usage, and that a small amount of spills are tolerable to maximize task concurrency. A spill-aware scheduler like [5] yields suboptimal performance due to too conservative setting of $N_{threads}$.

Fig. 9 compares the performance of WASP with the spill-aware scheduler [5]. The spill-aware scheduler conservatively limits $N_{threads}$ when the first spill is detected. In contrast, WASP continues to increase $N_{threads}$ until the overhead of GC time reaches a certain threshold. For the three shuffle-heavy workloads, WASP outperforms the spill-aware scheduler by 57% on average. Thus, we conclude that the GC time is a better metric to estimate memory pressure than the amount of spills.

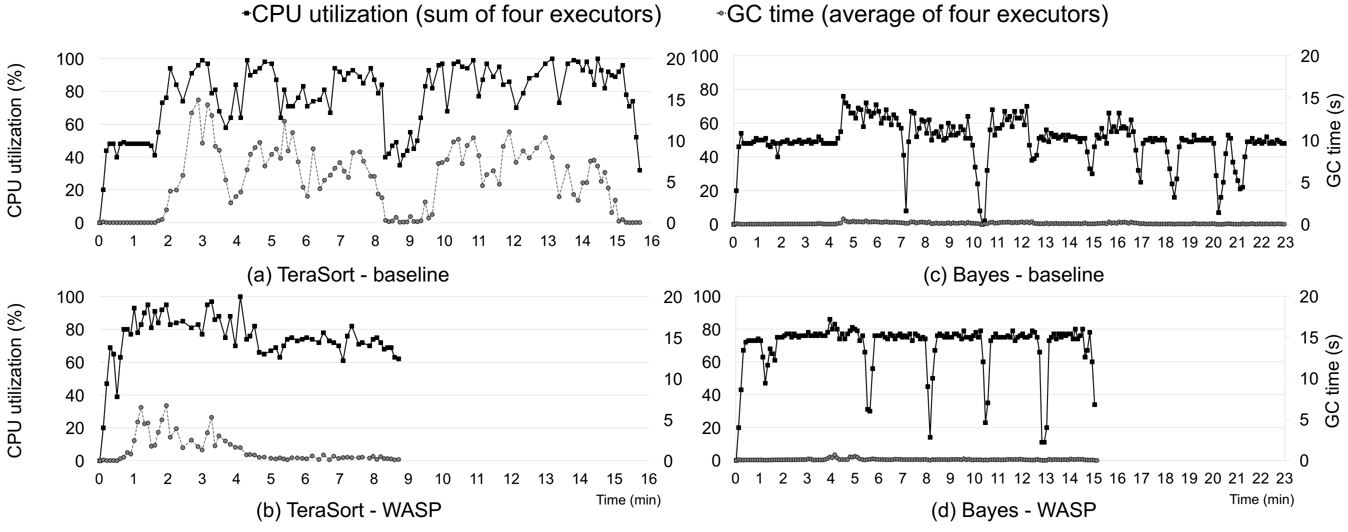
Sensitivity on GC threshold. GC threshold is an important system parameter for WASP. Setting it too low limits task concurrency by capping $N_{threads}$ too conservatively; setting it too high incurs larger GC overhead. Thus, we perform a sensitivity analysis over this parameter. Fig. 10 shows speedups of the three shuffle-heavy workloads with varying GC thresholds from 10% through 50% of execution time. Our experiment demonstrates that 20% is the optimal value balancing task concurrency and GC overhead. Therefore, We select this value as default.

Impact on CPU utilization and GC time. Fig. 11 shows the runtime behaviors of the baseline and WASP using two workloads: TeraSort (shuffle-heavy) and Bayes (shuffle-light). There are two graphs in each subgraph: CPU utilization and GC time sampled at a regular time interval. In Fig. 11(a), although maintaining high CPU utilization, TeraSort wastes a significant portion of CPU cycles for GCs. In contrast, WASP in Fig. 11(b) effectively reduces the GC time to spend most of CPU cycles for useful work, which is the primary source of performance improvement. Bayes is a shuffle-light workload causing a very small number of GCs. Thus, it is more important to increase CPU utilization by optimizing $N_{threads}$. In Fig. 11(c) the baseline achieves some 50% of CPU utilization. However, WASP demonstrates about 80% of CPU utilization by runtime optimization of $N_{threads}$ (in Fig. 11(d)). Note that the CPU utilization is dropped at the end of a job, which is shown in both graphs.

VI. Related Work

Guidelines for Spark performance tuning. Spark Tuning Guidelines provide best practices for parameter tuning to fully utilize hardware resources [7], [9], [23]. However, those parameters, including $N_{partitions}$ and $N_{threads}$, are typically set at program launch without considering workload and platform characteristics, thus leading to suboptimal performance. Tous et al. [24] performs an in-depth performance analysis for two benchmarks, Kmeans and sortByKey, to understand the impact of parameter setting on Spark performance in an HPC environment. Likewise, Gounaris et al. [25] investigate the impact of the most important tunable Spark parameters for shuffling, compression, and serialization, on application performance to guide parameter setting. However, unlike WASP, both guidelines do not take into account runtime behaviors (e.g., amount of spills and GCs) and use a single set of parameters for the entire application.

Fine-grained tuning of $N_{threads}$. Jia et al. [14] propose a prediction-based dynamic SMT threading (PBDST) framework, which dynamically adjusts $N_{threads}$ at each stage. Their prediction model for $N_{threads}$ takes microarchitectural event



(a) TeraSort - baseline

(c) Bayes - baseline

(b) TeraSort - WASP

(d) Bayes - WASP

Fig. 11. Resource utilization of baseline and WASP for TeraSort and Bayes

counters, such as cache miss rate, branch miss rate, as input, but fails to capture software-level events, such as GCs, spills, and straggler tasks. Kc et al. [13] propose to adjust $N_{threads}$ in Hadoop [3] at runtime based on the degree of resource contention. However, their proposal does not consider $N_{partitions}$ to yield suboptimal performance, especially for shuffle-heavy workloads. Pei et al. [5] propose a spill-aware scheduler to adjust $N_{threads}$ by estimating memory pressure. We have demonstrated the *GC-aware* scheduler of WASP yields better performance in Section V-D. Finally, Shi et al. [26] propose a memory usage rate-based scheduler (MURS), which suspends heavy tasks and/or adjust $N_{threads}$ to reduce memory pressure in an multi-application environment, which is orthogonal to our work.

Fine-grained tuning of $N_{partitions}$. There are many proposals that address the problem of suboptimal data partitioning in MapReduce frameworks [27], [3], to reduce the cost of shuffles. Paul et al. [10] propose CHOPPER to optimally repartition RDDs using workload DB, thus yielding minimum data skew across tasks. Gounaris et al. [11] propose a novel algorithm to dynamically adjust $N_{partitions}$ based on an analytical cost model to minimize resource usage without degrading performance. GraphChi [12] is a disk-based graph processing engine that partitions data at runtime for efficient use of memory. If a data partition, called shard, is too large to fit in memory, GraphChi breaks the interval associated with the shard into several sub-intervals (i.e., increasing $N_{partitions}$) to reduce memory pressure. Chen et al. [28] propose Tiled-MapReduce, which provides data-parallelism to a multicore system by partitioning a large MapReduce job into small subjobs to reduce hardware resource contentions, such as cache and memory bandwidth. However, all the aforementioned proposals yield suboptimal performance as they optimize $N_{partitions}$ only without considering $N_{threads}$, especially for shuffle-light workloads.

General-purpose $N_{threads}$ tuning frameworks. There are proposals for finding an optimal $N_{threads}$ for a given parallel program, based on runtime monitoring [29], [30], [31] and offline profiling [32]. DoPE [29] and Parcae [30] are compiler-assisted runtime systems that dynamically adjust $N_{threads}$ on multicore CPUs. They also extract the multiple types parallelism from an application and reconfigure the parallel execution mode based on runtime monitoring. PD [31] is similar to DoPE and Parcae in spirit, but improves resource utilization by removing pause time for reconfiguration. However, these frameworks optimize $N_{threads}$ in isolation, and do not take into account memory usage-related issues (e.g., GCs and spills) pertinent to the performance of in-memory MapReduce frameworks.

VII. Conclusion

This paper proposes WASP, a workload-aware task scheduler and partitioner for in-memory MapReduce frameworks, which jointly optimizes both $N_{partitions}$ and $N_{threads}$ at runtime. Whenever a new Spark job is created, WASP executes an analytical model that predicts optimal settings of $N_{partitions}$ and $N_{threads}$ for all of its stages based on their computation types. The GC-aware task scheduler of WASP takes this as input to further optimize $N_{threads}$ during task execution to maximize CPU utilization while minimizing the overhead of data spills and GCs. Our evaluation of WASP on two native platforms and a cluster of VMs on Amazon EC2 with 6 HiBench applications demonstrates promising results. WASP improves the performance by up to 3.22× over the baseline configuration following Spark Tuning Guidelines with a geomean speedup of 1.74× on a 4-node cluster with 64 fat cores and 1.56× on a single-node machine with 64 thin cores. WASP also improves the performance by 1.31× on average while reducing the operating cost by up to 40% on a 64-node Amazon EC2 cluster. WASP allows a Spark user to focus on program logic instead of tedious tasks of parameter tuning.

Acknowledgements

This work was supported by a research grant from Samsung Electronics and by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. B0101-17-0644, Research on High Performance and Scalable Manycore OS).

References

[1] "Apache Spark," http://spark.apache.org/.

[2] "Apache Ignite," https://ignite.apache.org/.

[3] "Apache Hadoop," http://hadoop.apache.org/.

[4] 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 *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. USENIX Association, 2012, pp. 2–2.

[5] C. Pei, X. Shi, and H. Jin, *Improving the Memory Efficiency of In-Memory MapReduce based HPC Systems*. Springer International Publishing, 2015, pp. 170–184.

[6] B. Nicolae, C. Costa, C. Misale, K. Katrinis, and Y. Park, "Towards Memory-optimized Data Shuffling Patterns for Big Data Analytics," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 409–412.

[7] "Apache Spark: Tuning Spark," http://spark.apache.org/docs/latest/tuning.html/.

[8] "Apache Spark: Spark Configuration," http://spark.apache.org/docs/latest/configuration.html/.

[9] "How-to: Tune Your Apache Spark Jobs," http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/.

[10] A. K. Paul, W. Zhuang, L. Xu, M. Li, M. M. Rafique, and A. R. Butt, "CHOPPER: Optimizing Data Partitioning for In-memory Data Analytics Frameworks," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2016, pp. 110–119.

[11] A. Gounaris, G. Kougka, R. Tous, C. Tripiana, and J. Torres, "Dynamic Configuration of Partitioning in Spark Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1891–1904, 2017.

[12] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. USENIX Association, 2012, pp. 31–46.

[13] K. Kc and V. W. Freeh, "Dynamically Controlling Node-level Parallelism in Hadoop," in *Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing*, ser. CLOUD '15. IEEE Computer Society, 2015, pp. 309–316.

[14] Z. Jia, C. Xue, G. Chen, J. Zhan, L. Zhang, Y. Lin, and P. Hofstee, "Auto-tuning Spark Big Data Workloads on POWER8: Prediction-based Dynamic SMT Threading," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. ACM, 2016, pp. 387–400.

[15] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 41–51.

[16] "Amazone Elastic Compute Cloud," https://aws.amazon.com/ec2/.

[17] A. Sodani, "Knights Landing (KNL): 2nd Generation Intel® Xeon Phi processor," in *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE, 2015, pp. 1–24.

[18] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, ser. CF '15. ACM, 2015, pp. 53:1–53:8.

[19] "Spark Performance Tests," https://github.com/databricks/spark-perf/.

[20] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 488–499.

[21] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making Sense of Performance in Data Analytics Frameworks," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. USENIX Association, 2015, pp. 293–307.

[22] A. J. Awan, V. Vlassov, M. Brorsson, and E. Ayguade, "Node Architecture Implications for In-memory Data Analytics on Scale-in Clusters," in *Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, ser. BDCAT '16. ACM, 2016, pp. 237–246.

[23] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, "Learning Spark: Lightning-fast Big Data Analysis." O'Reilly Media, Inc., 2015.

[24] R. Tous, A. Gounaris, C. Tripiana, J. Torres, S. Girona, E. Ayguadé, J. Labarta, Y. Becerra, D. Carrera, and M. Valero, "Spark Deployment and Performance Evaluation on the Marenostrum Supercomputer," in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015, pp. 299–306.

[25] A. Gounaris and J. Torres, "A Methodology for Spark Parameter Tuning," *Big Data Research*, 2017.

[26] X. Shi, X. Zhang, L. He, H. Jin, Z. Ke, and S. Wu, "MURS: mitigating memory pressure in service-oriented data processing systems," *CoRR*, vol. abs/1703.08981, 2017. [Online]. Available: http://arxiv.org/abs/1703.08981

[27] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[28] R. Chen and H. Chen, "Tiled-MapReduce: Efficient and Flexible MapReduce Processing on Multicore with Tiling," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 1, p. 3, 2013.

[29] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August, "Parallelism Orchestration Using DoPE: The Degree of Parallelism Executive," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 26–37.

[30] A. Raman, A. Zaks, J. W. Lee, and D. I. August, "Parcae: A System for Flexible Parallel Execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. ACM, 2012, pp. 133–144.

[31] S. Sridharan, G. Gupta, and G. S. Sohi, "Holistic Run-time Parallelism Management for Time and Energy Efficiency," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. ACM, 2013, pp. 337–348.

[32] Z. Wang and M. F. O'Boyle, "Mapping Parallelism to Multi-cores: A Machine Learning based Approach," in *ACM Sigplan notices*, vol. 44, no. 4. ACM, 2009, pp. 75–84.