Improving Spark Application Throughput Via Memory Aware Task Co-location: A Mixture of Experts Approach

Anonymous Author(s)

Abstract

Data analytic applications built upon big data processing frameworks such as Apache Spark are an important class of applications. Many of these applications are not latency-sensitive and thus can run as batch jobs in data centers. By running multiple applications on a computing host, task co-location can significantly improve the server utilization and system throughput. However, effective task co-location is a non-trivial task, as it requires an understanding of the computing resource requirement of the co-running applications, in order to determine what tasks, and how many of them, can be co-located. State-of-the-art co-location schemes either require the user to supply the resource demands which are often far beyond what is needed; or use a one-size-fits-all function to estimate the requirement, which, unfortunately, is unlikely to capture the diverse behaviors of applications.

In this paper, we present a mixture-of-experts approach to model the memory behavior of Spark applications. We achieve this by learning, off-line, a range of specialized memory models on a range of typical applications; we then determine at runtime which of the memory models, or experts, best describes the memory behavior of the target application. We show that by accurately estimating the resource level that is needed, a co-location scheme can effectively determine how many applications can be co-located on the same host to improve the system throughput, by taking into consideration the memory and CPU requirements of co-running application tasks. Our technique is applied to a set of representative data analytic applications built upon the Apache Spark framework. We evaluated our approach for system throughput and average normalized turnaround time on a multi-core cluster. Our approach achieves over 86.4% of the performance delivered using an ideal memory predictor. We obtain, on average, 8.75x improvement on system throughput and a 51% reduction on turnaround time over executing application tasks in isolation, which translates to a 1.31x and 1.75x improvement over a state-of-the-art co-location scheme for system throughput and turnaround time respectively.

CCS Concepts • **Computing methodologies → Distributed algorithms**;

Keywords Resource Modeling, Memory Management, Apache Spark, Task Scheduling, Predictive Modeling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Middleware'17,
© 2017 ACM. xxx-xxxx-xx-xxx/xx/xx. . . $xx.00
DOI: xx.xxx/xxx_x

ACM Reference format:
Anonymous Author(s). 2017. Improving Spark Application Throughput Via Memory Aware Task Co-location: A Mixture of Experts Approach. In *Proceedings of ACM/IFIP/USENIX Middleware, , (Middleware'17),* 14 pages.
DOI: xx.xxx/xxx_x

1 Introduction

Big data applications built upon frameworks such as Hive [41], Hadoop [36] and Spark [49] are commonplace. Unlike interactive jobs, many of the data analytic applications are not latency-sensitive. Therefore, they often run as batch jobs in a data center. However, how to effectively schedule such applications to improve the server utilization and the system throughput remains a challenge.

Specifically, if an application task is given the entirety of main memory on each host to which it is deployed, it is effectively preventing the host machine from being used for any other application until the current one has finished, even if the task does not use all of the memory. Because many data analytic tasks do not use 100% of the CPU during execution [2, 24] there is a significant portion of unused processing capacity. An alternate approach is to share the computing host between multiple application tasks (where each task does not use all of the memory), this could save time and energy by co-locating processes more effectively on fewer machines.

Effective task co-locations require knowledge of the application's resource demand. For in-memory data processing frameworks like Apache Spark, RAM consumption is a major concern [27]. It is particularly important to understand the memory behavior of the application. If we co-locate too many applications or give too much data to a single task, such that their total memory consumption exceeds the physical memory of the host, we could cause memory paging onto the hard disk, or an "out-of-memory" error, slowing down the overall system. To achieve this we need a technique to predict the precise memory requirement of any given Spark application.

Existing task co-location schemes require either: the user to provide information of the resource requirement [21], or employ an analytical [17] or statistical model [10, 18, 31] to estimate the resource requirement based on historical jobs or runtime profiling. These approaches, however, have significant drawbacks. Firstly, it is difficult for a user to give a precise estimation of the application's requirement; and thus, the supplied information is often over-conservative, asking far more resources than the application needs. Secondly, a one-size-fits-all function is unlikely to precisely capture behaviors of diverse applications, and no matter how parameterized the model is, it is highly unlikely that a model developed today will always be suited for tomorrow.

In this paper, we present a generic framework to model the memory behavior of Spark applications. As a departure from prior work that uses a fixed utility function to model the resource requirement, we use multiple linear and non-linear functions to model the memory requirements of various applications. We then build a machine learning classifier to select which function should be used for a given application and dataset at runtime. As the program implementation, workload and underlying hardware changes, different models will be dynamically selected at runtime. Such an approach is known as *mixture-of-experts* [23]. The central idea is that instead of using a single monolithic model, we use multiple models (*experts*) where each expert is specialized for modeling a subset of applications. Using this approach, each memory model is used only for the applications for which its predictions are effective. One of the advantages of our approach is that new functions can easily be added and are selected only when appropriate. This means that the system can evolve over time to target a wider range of applications, by simply inserting new functions. The result is a new way of using machine learning for system optimization, with a generalized framework for a diverse set of applications.

We evaluate our approach on a 40-node multi-core cluster using 44 Spark applications that cover a wide range of application domains. We show that the accurate memory-footprint prediction given by our approach enables the runtime scheduler to make better use of spare computing resources to improve the overall system throughput via task co-location. We use two distinct metrics to quantify our results: *system throughput* and *average normalized turnaround time*, and compare our approach against a state-of-art resource and task scheduler [10]. Experimental results show that our approach is highly accurate in predicting the application's memory requirement, with an average error of 5%. By better utilizing the memory resources of a host, our system achieves 8.75x improvement of system throughput and a 51% reduction in application turnaround time. This translates to a 1.31x and 1.78x improvement over the state-of-art respectively on throughput and turnaround time.

This paper makes the following contributions:

- We present a novel machine learning based approach to automatically learn how to model the memory behavior of Spark applications (Section 3);
- Our work is the first to employ mixture-of-experts for resource demand modeling. Our generic framework allows new models to be easily added to target a wider range of applications and performance metrics;
- We show how to combine this resource modeling framework with runtime task co-location policies to improve system throughput for Spark applications (Section 4);
- Our system is immediately deployable on real systems and does not require any modification to the application source code.

2 Background and Overview

2.1 Apache Spark

Apache Spark is a general-purpose cluster computing framework [49]. with APIs in Java, Scala and Python and libraries for streaming, graph processing and machine learning [49]. It is one of the most active open source projects for big data processing, with over 2,000 contributors in 2016. Each Spark application runs as an independent set of *executor* processes, each with dedicated memory space for executing parallel jobs within the application. The executors are coordinated by the *driver* program running on a *coordinating node*. Input data of Spark applications is stored in a shared filesystem and organized as *resilient distributed datasets* (`RDDs`) – a collection of objects that can be operated on in parallel. Each Spark executor allocates its own heap memory space for caching `RDDs`. This work exploits the data parallel property of `RDDs` to characterize (or fingerprint) the application's memory behavior without wasting computing cycles.

2.2 Problem Scope

Our goal is to develop a framework to accurately predict the resource requirement of Spark applications for arbitrary inputs. In this work, we focus on the memory requirement as RAM resources are a major concern for in-memory data processing frameworks like Apache Spark [27]. To demonstrate the usefulness of our approach, we apply it to perform task co-location for batched, data-analytic Spark applications. We do not consider latency-sensitive applications, such as search, as their stringent response time targets often require isolated execution [30].

Our approach estimates the memory footprint of a Spark executor for a given input dataset. It then uses this information to determine if there are enough spare resources (i.e. memory and CPU) to co-locate tasks; if there is, it calculates how many tasks could be co-located and how much work should be given to each task. We exploit the fact that many big data applications do not spend all of their time at 100% CPU [24] (see also Section 6.6). This observation suggests that there are opportunities to co-locate Spark tasks without significantly increasing the CPU contention and slow down the performance of co-running applications (see also Section 6.7). Our approach is applied to a simple task co-location policy in this work, yet the resulted scheme outperforms the state-of-the-art task scheduling scheme. We want to stress that our framework can be used by other scheduling policies to provide an estimation of the application's resource demand to support decision making.

Our current implementation is restricted to applications whose memory footprint is a function of their input size, this is a typical behavior for many data analytical applications. In this work, we do not explicitly model disk and network I/O contention, because prior research suggests that they have little impact on the performance of in-memory processing frameworks including Apache Spark [35]. Nonetheless, our framework is general and allows new models to be easily added to target different applications, or other performance and resource metrics in the future.

2.3 Overview of Our Approach

Our approach, depicted in Figure 1, is *completely automated*, and no modification to the application source code is required.

Our mixture-of-experts framework for memory footprint prediction consists of a range of distinct models built off-line.

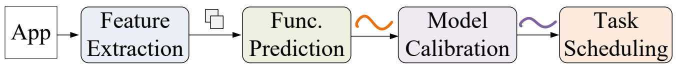


Figure 1. Overview of how our approach can be used for task scheduling. For an incoming application, our approach first extracts the features of the program. Based on the feature values, it predicts which of the off-line learned memory functions best describes the memory behavior of the application. It then instantiates the function parameters by profiling the application on some small sets of the input data items. A runtime scheduler then utilizes the memory function to perform task co-location.

An expert selector decides which model should be invoked, based on the runtime information of the application. To use our resource modeling framework to perform task co-location, a task scheduler follows a number of steps described as follows.

For each "*new*" application that is ready to run, we predict which of the *off-line* learned experts, *termed 'memory function' in this paper*, best describes its memory behavior, i.e. how the memory footprint changes as the input size varies. The selection of the memory function is based on runtime information of the program, such as the number of L1 data and instruction cache misses. This information is collected by running the application on a small portion (around 100MB) of the input data items[1].

We then calibrate the selected function to tailor its parameters to the target program and input. We do so by first profiling the application with two small different-sized parts of the application input to instantiate two function parameters; we then use the measured memory footprints to instantiate the parameter values. The calibrated memory function is then used to determine how many unprocessed data items should be allocated to an executor under a given memory budget. During the profiling run, we also record the average CPU usage of the application. After determining which memory function to use and obtaining the CPU usage of the application, the runtime scheduler can spawn new executors to run on servers that have spare memory, and if the aggregate CPU load of co-running tasks will go over 100% (i.e. to avoid CPU contention).

Since runtime information collection and model calibration are all performed on some unprocessed data items and contribute to the final output, no computing cycle is wasted on profiling. Furthermore, we will re-run an executor process in isolation if it fails because of an "out-of-memory" error, but this was not observed in our experiments.

The key to our approach is choosing the right memory function and then using lightweight profiling to instantiate the function parameters. An alternative is to use extensive profiling runs at runtime to find a model to fit the application's memory behavior. However, doing so will incur significant overhead and could outweigh the benefit (see Section 6.4).

[1]We choose this modest input size as an input of this size typically takes a short time to process, while at the same time, it is sufficiently large (i.e. this often results in a working set that is larger than the size of the L3 data cache in most of the high-end CPUs) to capture the cache behavior of the application.

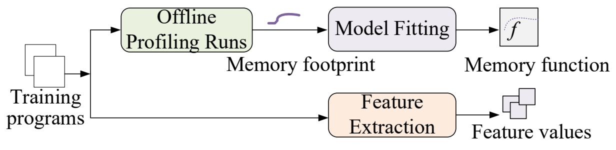


Figure 2. The training process.

Table 1. Memory functions used in this work

Modeling Technique	Formula
(Piecewise) Linear Regression	$y = m * x^b$
Exponential Regression	$y = m * (1 - e^{(-b*x)})$
Napierian Logarithmic Regression	$y = m + ln(x) * b$

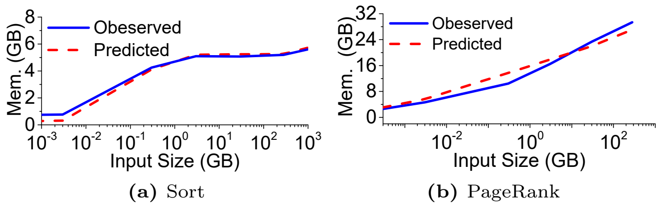


Figure 3. The observed and predicted memory footprints for `Sort` and `PageRank` from HiBench. The memory footprint of the two applications can be accurately described using one of the memory functions listed in Table 1.

In the next section, we will describe how supervised machine learning [1] can be used to construct the memory functions (experts) and the expert selector to choose which function to use for any "*unseen*" applications.

3 Predictive Modeling

Our approach involves using multiple memory functions (experts) to capture the memory requirement of an application for a specific runtime input. The set of memory functions are constructed offline on a set of example programs, and then an expert selector dynamically chooses the best expert to use at runtime.

Our expert selector for determining the memory function is a K-nearest neighbour (`KNN`) classifier [25][2]. The input to the classifier is a set of runtime features. Its output is a label to the memory function that describes the memory behavior of the target application and the specific dataset.

3.1 Learning Memory Functions

Our memory functions and expert selector are trained *off-line* using a set of training benchmarks. The learned expert selector can then be used to predict which memory function to use for any *new, unseen* application. Figure 2 depicts the process of collecting training data to learn the memory functions to build the `KNN` classifier. This involves finding a mathematical function to model the memory footprint for each benchmark and collecting feature values of each training program.

[2]We have also explored several alternative classification techniques, including decision trees and neural networks. This is discussed in Section 6.8.

During the training process, we run selected training programs in isolation on a computing host. We profiled each training application with different sized inputs. For each program input, we record the memory footprint of the Spark executor process. Next, we try different mathematical modeling techniques to discover which model best describes the relationship between input size and memory allocation, that is, as the input size increases, how does the memory allocation change. In this training phase, we record the memory function used to describe each training program. Our intuition is that the memory behavior for programs with similar characteristics will be similar. This hypothesis is confirmed in Section 6.8.

We use a set of linear and non-linear regression techniques to model the application's memory behavior. Table 1 gives the full list of modeling techniques we used in this work. Each of our models has two parameters, m and b, to be instantiated during runtime model calibration. Here x and y are the input size (i.e. the number of `RDD` objects in our case) and the predicted memory footprint respectively. It is worth mentioning that all the memory functions are automatically learned from training data, treating the applications as black boxes; new applications would similarly be learned automatically, potentially causing the addition of new memory functions.

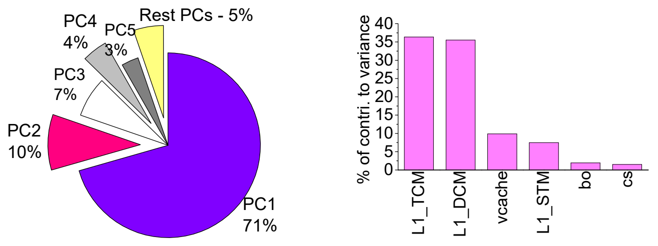
Example. Figure 3 shows the observed memory footprint and the prediction given by our memory function for `Sort` and `PageRank`. For these two applications, the memory functions used in this work can accurately model their memory behaviors. Specifically, the memory footprint, y, of `Sort` and `PageRank` for a given input size, x, can be precisely described using an exponential function , $y = m * (1 - e^{(-b*x)})$, where $m = 5.768$, $b = 4.479$ and a Napierian logarithmic function, $y = m + ln(x) * b$, where $m = 16.333$, $b = 1.79$ respectively.

After building the memory functions, we need to have a mechanism to decide which of the functions to use. One of the key aspects in building a successful expert predictor is finding the right features to characterize the input application task. This process of feature selection is described in the next section. This is followed by sections describing training data generation and then how to use the expert selector at runtime.

3.2 Runtime Features

Raw Features. Expert selection is based on runtime characteristics of the application task. These characteristics, called *features*, are collected using system-wide profiling tools: `vmstat`, Linux `perf` and performance counter tool `PAPI`. Collected feature values are encoded to a vector of real values. We considered 22 raw features in this work, which are given in Table 2. Some of these features are selected based on our intuition, while others are chosen based on prior work [11, 48]. All these features can be automatically and externally observed, without needing access to the source code.

Feature Scaling. Supervised learning typically works better if the feature values lie in a certain range. Therefore, we scaled the value for each of our features between the range of 0 and 1. We record the maximum and minimum value of each feature found at the training phase, and use these values



(a) Principal components **(b)** Most important raw features

Figure 4. The percentage of principal components (`PCs`) to the overall feature variance (a), and contributions of the 5 most important raw features in the `PCA` space (b).

to scale features extracted from a new application during runtime deployment.

Feature Reduction. Given the relatively small number of training applications, we need to find a compact set of features in order to build an effective predictor. Feature reduction is automatically performed through applying Principal Component Analysis (`PCA`) on the scaled raw features. This technique removes the redundant features by linearly aggregating features that are highly correlated. After application of `PCA`, we use the top 5 principal components (`PCs`) which account for 95% of the variance of the original feature space. We record the `PCA` transformation matrix and use it to transform the raw features of the target application to `PCs` during runtime deployment. Figure 4a illustrates how much feature variance that each component accounts for. This figure shows that prediction can accurately draw upon a subset of aggregated feature values.

Feature Analysis. To understand the usefulness of each raw feature, we apply the Varimax rotation [32] to the `PCA` space. This technique quantifies the contribution of each feature to each `PC`. Figure 4b shows the top 5 dominant features based on their contributions to the `PCs`. Cache features, `L1_TCM`, `L1_DCM` and `L1_STM`, are found to be important for describing memory behaviors. This is not supervising as cache hit/miss rates are shown to be useful in characterizing the application behavior in prior works [6, 37]. Other features of virtual memory usage (`vcache`), I/O (`bo`) and thread contention (`cs`) are also considered to be useful, but are less important compared to cache features. Using this technique, we sort the raw features listed in Table 2 according to the importance. The advantage of our feature selection process is that it automatically determines what features are useful when targeting a new computing environment where the importance of features may change.

3.3 Collect Training Data

We use *cross-validation* to construct memory functions and the `KNN` classifer to select which function to use. This standard evaluation technique works by picking some target programs for testing and using the remaining ones for training.

In this work, we use benchmarks from the HiBench [22] and BigDataBench [16] suites to build the memory models. Later we show that our approach works well on benchmarks

Table 2. Raw features, sorted by their importance

Feature	Description	Feature	Description
`L1_TCM`	L1 total cache miss rate	`L1_DCM`	L1 data cache miss rate
`vcache`	% of memory used as cache	`L1_STM`	L1 cache store miss rate
`bo`	# blocks sent (/s)	`L2_TCM`	L2 data cache miss rate
`L3_TCM`	L2 total cache miss rate	`cs`	# context switches / s
`FLOPs`	# floating point operations /s	`in`	# interrupts / s
`L2_DCM`	L3 cache total miss rate	`L2_LDM`	L2 cache load miss rate
`L1_ICM`	L1 instr. cache miss rate	`swpd`	% of virtual memory used
`L2_STM`	L2 cache store miss rate	`IPC`	instruction per cycle
`L1_LDM`	L1 cache load miss rate	`L2_ICM`	L2 instr. cache miss rate
`ID`	% of idle time	`WA`	% of time on IO watting
`US`	% spent on user time	`SY`	% spent on kernel time

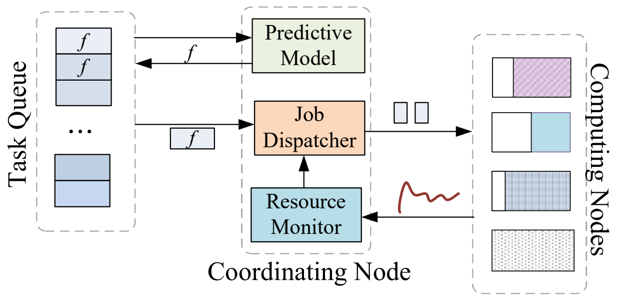


Figure 5. Our system predicts the memory function for each application and monitors the memory resources of computing nodes. The runtime scheduler creates new executors to run on computing nodes that have spare memory, and uses the memory function to determine how much data should be given to the executor under a memory budget.

from the Spark-Perf [9] and the Spark-Bench [27] suites, although we did not directly train our models on them. The process of collecting training data is described in Figure 2. To collect training data, we first extract the feature values of each training program by running a single executor process in isolation, using inputs with an average size of 100MB. Next, we run each training program with different sized inputs (ranging from ∼300MB to ∼1TB) and record the observed memory footprints. We then find a memory function to closely fit the curve. For each training program, we store its principal component values and the memory function in a database. Since training is only performed once, it is a *one-off* cost.

4 Runtime Deployment

Once we have learned the memory functions as described above, we can use a KNN algorithm to choose an appropriate function to estimate the memory footprint for any *unseen* applications with a given input, and to use the prediction to co-locate Spark executor tasks at runtime.

Our runtime system built upon YARN [42], a task and resource manager for Spark. The co-location scheme will be triggered when more than one Spark application is waiting to be scheduled. Figure 5 illustrates the architecture of our system. For each application task, we predict its memory function for its input dataset, and then use the memory function to co-locate Spark executor processes whenever possible.

4.1 Memory Requirement Prediction

To determine the memory function for an application task, a runtime system follows two steps, described as follows.

Memory Function Prediction. We run the incoming application on a small set of the input `RDD` objects (with an aggregated size of around 100MB) to collect and normalize feature values, and to perform the `PCA` transformation. We then calculate the Euclidean distance between the transformed input program feature vector and the feature vector of each training program to find out the *nearest neighbor*, i.e. the training program that is closest to the input program in the feature space (see also Section 6.8). We use the memory function of the nearest neighbor as the prediction. We also record the average CPU usage during this profiling run, and use this information later to determine whether co-location will cause CPU contention among co-running tasks. Our current implementation performs feature extraction by running the application on the lightly-loaded coordinating node (where the driver program runs). The results generated in the feature extraction phase will contribute to the final output of the application.

Model Calibration. After we have determined the memory function, we need to instantiate the function coefficients (i.e. m and b in Table 1). We calculate these by running the application on two sets of unprocessed input data items, where the first and the second sets contain 5% and 10% of the input items, respectively. To determine the function parameters, we measured the memory footprints during profiling runs, and use them together with the corresponding input sizes (i.e. the number of data objects) to solve the memory function equation. At this stage, we are only concerned with the application's memory footprint but not runtime. Therefore, profiling runs can be performed by either grouping different application tasks to run on a single host or running the target application with other latency-insensitive tasks. Again, the results generated during this phase will contribute to the final output of the application, no computing cycle is wasted.

Furthermore, since the input and output data of the Spark application typically stored in a shared filesystem, we do not need to explicitly move the data in or out from the profiling host.

4.2 Resource Monitor

Each computing node runs a daemon that periodically reports to the resource monitor its memory usage and CPU load. Our current implementation reports the average memory usage and system load within a 5-minute window. The information is retrieved from the Linux "`/proc`" system. Since this is performed at a coarse-grained level (i.e. minutes), the overhead of monitoring and communication is negligible. With this monitoring scheme in place, a task scheduler can respond to execution phase changes and load variations, avoiding over-subscribing the computing resources.

4.3 Job Dispatcher

By default, we use the dynamic allocation scheme of Spark to determine how many free server nodes to use to run an application. However, the Spark dynamic scheme is not perfect, so we utilize spare memory to spawn *additional* executors to run on servers that have spare resources. Also, instead of waiting for the servers to become completely free, our approach starts executing waiting applications as soon as possible, reducing the turnaround time.

Once we have the memory function of the highest-priority application, the job dispatcher will spawn a new executor for the application to run on severs that have spare memory and if the aggregate CPU load of all co-running tasks will not go over 100%. The dispatcher uses the memory function to determine how much memory is needed for the remaining input (to allow us to co-locate more applications if possible), and how many data items can be cached by the executor under a given memory budget. To estimate the aggregate CPU load, we add up the CPU load of the computing host (which is reported by the resource monitor) and the average CPU usage of the application to be scheduled (which is obtained during the profiling run for feature collection). Furthermore, the number of data items to give to the co-located executor is dynamically adjusted over time, adapting to the changes of execution stages and memory resources. A naive alternative is to statically set the executor heap size to the size of free memory. But doing so can over-subscribe the memory resources than necessary and precludes co-locating more than two applications (see Section 6.1).

To minimize the potential thread contention, we dynamically adjust the number of threads (tasks) created by each executor to evenly distribute processor cores across currently-running executors on a single host. Furthermore, to enforce a certain degree of fairness, it is important to make sure that the new co-running task does not use the resources that are deemed to be essential for the currently running application. While fairness is not a focus of this work, our prediction framework helps the scheduler in this endeavor.

Table 3. Application task mixes used in the experiments

Label	#App.	Label	#App.	Label	#App.	Label	#App.
L1	2	L2	6	L3	7	L4	9
L5	11	L6	13	L7	19	L8	23
L9	26	L10	30				

5 Experimental Setup

5.1 Platform and Benchmarks

Hardware. We use a multi-core cluster with 40 nodes, each with an 8-core Xeon E5-2650 CPU @ 2.6GHz (16 threads with hyper-threading), 64GB of DDR4 RAM, and 16GB of swap. Nodes have SSD storage and are connected through 10Gbps Ethernet, precluding disk and network contention.

Software. Each computing node runs CentOS 7.2 with Linux kernel 3.12. We rely on the local OS to schedule processes and do not bind tasks to specific cores. We use Apache Spark 1.3.0 with Hadoop Yarn 2.4 as the cluster manager and HDFS as the Spark file management system. We use the Oracle Java runtime, Java SE 8u. We run Spark in the cluster mode. We also use the dynamic resource allocation scheme, so that memory will be given back to Spark when an application task completes. We run the Spark driver on a dedicated coordinating node and try to run multiple Spark executors on a single host to improve the system throughput. Finally, we use the Spark default configuration for memory management.

Workloads. We used 44 Java-based Spark applications from four widely used suites: HiBench [22], BigDataBench [16], Spark-Perf [9] and Spark-Bench [27]. These benchmarks implement the core algorithms used in real-life applications e.g. machine learning, image and natural language processing, and web analysis.

5.2 Evaluation Methodology

Runtime Scenarios. We evaluated our scheme using ten runtime scenarios with a mix of 2 to 30 randomly selected applications, detailed in Table 3. For each scenario, we try $\sim$100 different application mixes and make sure all benchmarks are included in each scenario. The input size ranges from small ($\sim$300MB) and medium ($\sim$30GB) to large ($\sim$1TB). Inputs were generated using the input generation tool provided by each benchmark suite. In the experiments, all tasks are scheduled on a first come first serve basis, but we stress that our technique can be applied to *any scheduling policy*.

Predictive Model Evaluation. Our memory functions and predictor are trained using 16 benchmarks from HiBench and BigDataBench. We then apply the trained models to all 44 benchmarks from the four benchmark suites. When there are benchmarks from HiBench and BigDataBench present in the task group, we use the standard *leave-one-out-cross-validation*, i.e. to exclude the target applications from the training program set and use the remaining benchmarks from HiBench and BigDataBench to build our model. To provide a fair comparison, when testing an application from one benchmark suite that has an equivalent implementation in the other suite, we also exclude the benchmark from other suite from the training set. For example, when testing `Sort` from HiBench, we exclude `Sort` from BigDataBench from training.

Performance Report. For each test case, we report the performance as *the geometric mean* across all configurations. We replay the schedule decisions for each test case multiple times, until the difference between the upper and lower confidence bounds under a 95% confidence interval setting is smaller than 5%. Furthermore, the time spent on feature extraction, model calibration, and prediction is included in our results.

5.3 Evaluation Metrics

We use two standard evaluation metrics for multi-programmed workloads: *system throughput* and *turnaround time*. We use the definitions given in [13], defined as follows.

1. System throughput (STP) is a *higher is better metric.* It describes the aggregated progress of all jobs under co-location execution over running each job one by one using isolated execution. This is calculated as:

$$STP = \sum_{i=1}^{n} \frac{C_i^{is}}{C_i^{cl}} \tag{1}$$

where n is the number of application tasks to be scheduled, and C_i^{is} and C_i^{cl} are the execution time for task i under the isolated execution mode (`is`) where the task uses all available memory; and the co-locating mode (`cl`) where there may be multiple tasks running on the same host.

2. Average normalized turnaround time (ANTT) is a *smaller is better* metric. It quantifies the time between a task being created and its completion, indicating the average user-perceived delay. This metric is defined as:

$$ANTT = \frac{1}{n} \sum_{i=1}^{n} \frac{C_i^{cl}}{C_i^{is}} \tag{2}$$

5.4 Comparative Approaches

Quasar. This is a state-of-the-art co-location scheme [10]. QUASAR uses classification techniques to determine the characteristics of the application to perform resource allocation, and task assignment and co-location. Similar to our dynamic scheme, QUASAR monitors workload performance to adjust resource allocation and assignment when needed. Unlike our approach, QUASAR uses a single model for resource estimation. To provide a fair comparison, we have implemented the QUASAR classification scheme using the same set of training programs that we used to build our models.

Pairwise. This pairwise co-location scheme looks for servers with spare memory to co-locate an additional task on the host. It sets the maximum heap size of the co-locating task to the size of free memory, and relies on the Spark default scheduler to determine how many `RDD` data items to be allocated to the co-running task. This represents the default resource allocation policy used by many co-location schemes [29].

Oracle. We also compare our approach to the performance of an ideal predictor (ORACLE) that gives the perfect memory prediction for an application. This comparison indicates how close our approach is to the theoretically perfect solution. The prediction given by the ORACLE scheme is obtained through profiling the application on a given set of input `RDD` data items, but the profiling overhead is not included in the results since we assume the ORACLE predictor has the ability to make prophetic prediction. Using the ORACLE predictor, the runtime scheduler can then search for the optimal number of data items to be given to a co-running task.

5.5 Highlights

The highlights of our evaluation are as follows:

- With the help of our mixture-of-experts approach, a simple task co-location scheme achieves, on average, a 8.75x improvement on STP and a 51% reduction on ANTT over isolated execution. This translates to a 1.31x and 1.75x improvement on STP and ANTT respectively, when compared to QUASAR. See Section 6.1;
- Our approach is highly accurate in predicting the memory footprint of Spark applications, with an error of less than 5% for most cases. See Section 6.8;
- Our scheme is low-overhead. The time spent on feature extraction and model calibration is less than 10% of the total application execution time, and the profiling runs contribute to the final results. See Section 6.5;
- We thoroughly evaluate our scheme by comparing it against several alternative task co-location schemes and modeling techniques, and performing a detailed analysis on the working mechanism of the approach.

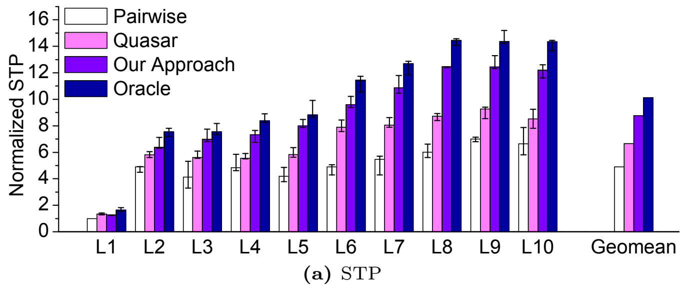
6 Experimental Results

In this section we first show the overall performance of our approach against alternative schemes. We then provide analysis of the working mechanism of our approach.

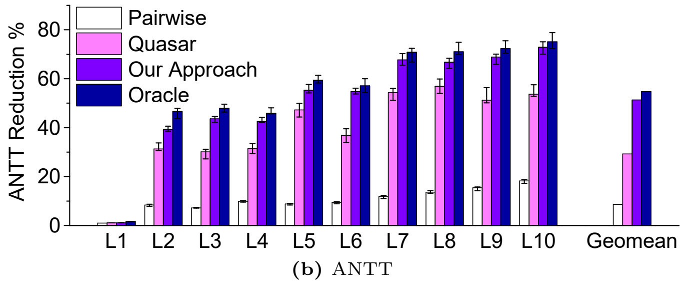
Unless stated otherwise, we report each approach's performance on STP and ANTT, by normalizing the results to a *baseline* that schedules the applications one by one with each application exclusively using all the memory of each allocated computing node. The normalized STP and ANTT are referred to as *normalized STP* and *ANTT reduction* (shown in percentage) respectively.

6.1 Overall Performance

STP. Figure 6 (a) confirms that task co-location improves system throughput. As the number of tasks to be scheduled increases, we see an overall increase in the STP. PAIRWISE performs reasonably well for small task groups, but it misses significant opportunities for large task groups. For L9 and L10, PAIRWISE only delivers half of the ORACLE performance. This is because PAIRWISE does not scale up beyond pairwise co-location. QUASAR performs significantly better than PAIRWISE by using a classifier model to coordinate resources among co-locating tasks, but it is not as good as our approach. By employing multiple functions to model diverse applications, our approach constantly outperforms PAIRWISE and QUASAR across all task groups. For large task groups (L8 - L10), our approach delivers over 1.8x and 1.51x improvement on the STP over PAIRWISE and QUASAR respectively. Overall, QUASAR gives on average 6.6x improvement on STP,



(a) STP



(b) ANTT

Figure 6. Our approach outperforms Pairwise and Quasar on STP (a) and ANTT (b). The baseline is running the applications one by one using isolated execution. The min-max bars show the range of performance achieved across task mixes for each runtime scenario.

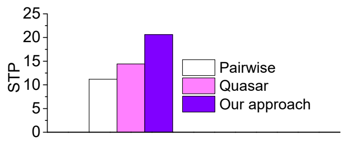
which translates to 65.7% of the Oracle performance. Our approach achieves 8.75x improvement on STP, which translates to a 1.31x improvement over Quasar or 86.4% of the Oracle performance.

ANTT. Figure 6 (b) shows the ANTT reduction over the baseline. By maximizing the system throughput, task co-location in general leads to favorable ANTT results, particularly for large task groups. Quasar and our approach outperforms Pairwise on ANTT by a factor of over 4x from L2 onward. Our approach delivers better turnaround time over Quasar, by avoiding memory contention among co-locating Spark tasks. On average, our approach reduces the turnaround time by 51% across different task groups. This translates to 94.6% of the Oracle performance. When compared to the 54% Oracle performance given by Quasar, our approach achieves 1.75x better turnaround time.

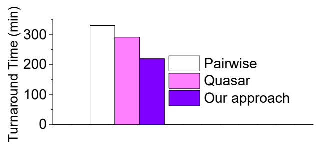
Summary. We achieve 86.4% and 94.6% of the Oracle performance for STP and ANTT respectively, outperforming Pairwise, a widely used co-location policy, and Quasar, a state-of-the-art co-location policy. The advantage of our approach is largely attributed to its use of multiple models instead of just one to precisely capture an applications' memory behavior. Without this accurate information, the alternative scheme often over- or under-provisions resources, leading to worse performance.

6.2 Server Utilization

Figure 7 shows the CPU utilization across 40 computing nodes for Pairwise, Quasar and our approach when scheduling 30 Spark applications (L10), and Figure 8 presents the

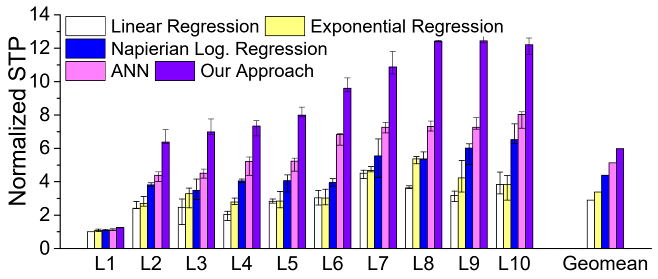


(a) STP (higher is better)

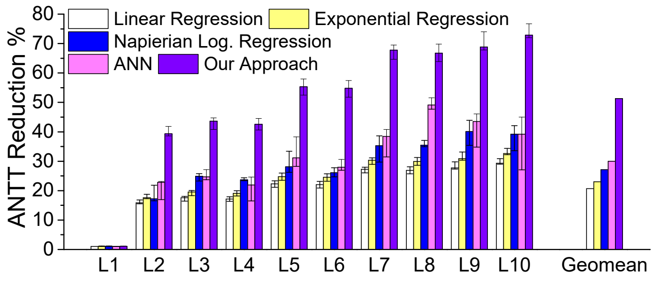


(b) Turnaround Time (lower is better)

Figure 8. Resultant STP (a) and turnaround time (b) for the scheduling scenario in Figure 7. Our approach gives better STP and faster turnaround time when compared to alternative co-location schemes.



(a) STP



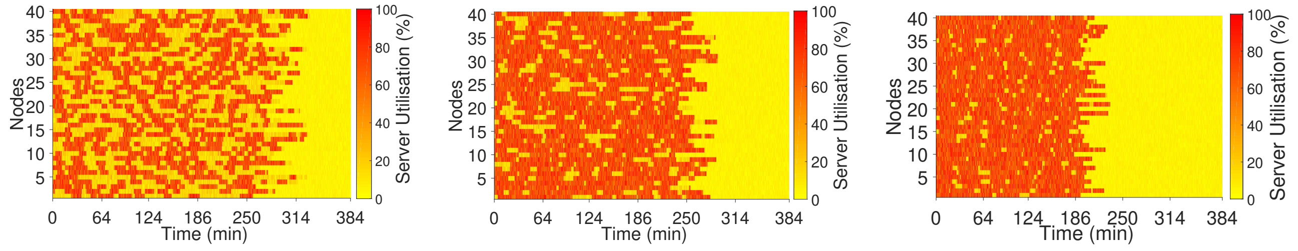
(b) ANTT

Figure 9. Compare to unified model based approaches that use a single modeling technique to describe the application's memory behavior.

turnaround time (i.e. the wall clock time to finish the set of jobs) given by each approach. By carefully co-locating tasks using memory footprint predictions, our approach gives the best server utilization, which in turn leads to the highest STP (1.84x and 1.43x higher STP over Pairwise and Quasar respectively) quickest turnaround time (1.5x and 1.32x faster turnaround time over Pairwise and Quasar respectively).

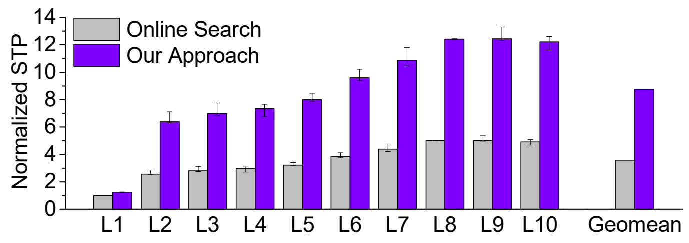
6.3 Compare to Unified Models

Figure 9 compares our scheme to approaches that use one modeling technique to predict the application's memory footprint. In addition to the three memory functions listed in Table 1, we also compare our scheme to a 3-layer artificial neural network (ANN) trained using a backpropagation algorithm. We use the same training data to build the ANN model to predict the memory footprint. The input to the ANN model is the same set of features used by our approach. Among the single model approaches, the ANN gives the best performance due to its ability to model linear and non-linear

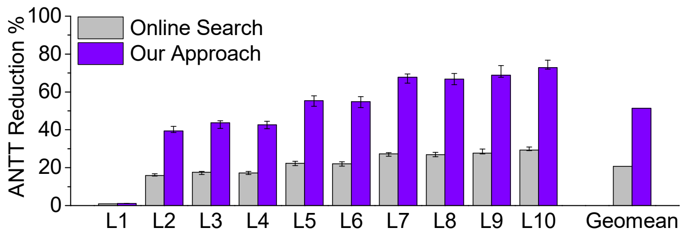


(a) Pairwise (b) Quasar (c) Our approach

Figure 7. CPU utilization across servers when scheduling 30 Spark applications (L10). The right-most non-zero point indicates the time when all applications finish. Our approach leads to the highest server utilization and quickest turnaround time.



(a) STP



(b) ANTT

Figure 10. Compare to using online search to allocate input for a given memory budget. Our approach significantly outperforms the online search scheme, because it avoids the runtime overhead associated with finding the optimal number of data items to be given to the co-running task.

behaviors. Our approach outperforms ANN and all other approaches on STP and ANTT. The results suggest the need for using multiple modeling techniques to capture the diverse application behaviors. This work develops a generic framework to support this.

6.4 Compare to Online Search

Figure 10 compares our approach to a method that uses descent gradient search to dynamically adjust the right input size for a given memory budget. The online search based approach gives rather disappointing results due to the large overhead involved in finding the right input size. Furthermore, this approach also suffers from a scalability issue, i.e. the searching overhead grows as the number of computing nodes increases. Our approach avoids the overhead by directly predicting the memory footprint, leading to 2.4x and 2.6x remarkably better performance on STP and ANTT respectively.

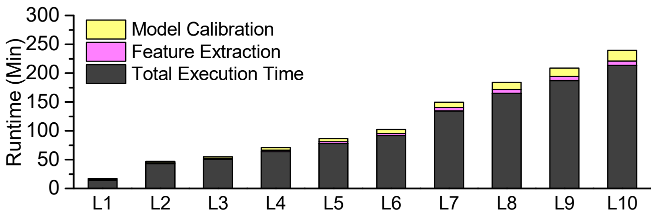


Figure 11. Average profiling time to total task execution time. It is to note that during feature extraction and model calibration, the application always executes a portion of the unprocessed data, no computing cycles are wasted.

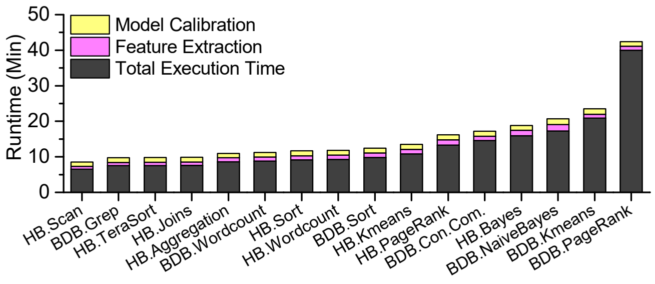


Figure 12. Average profiling time to total runtime per program for HiBench and BigDataBench.

6.5 Profiling Overhead

The stack chart in Figure 11 shows the average time spent on feature extraction and model calibration with respect to the total execution time per evaluation scenario. Figure 12 gives a breakdown on per benchmark basis using an input size of around 280GB. As profiling is performed on a single host (thus having little communication overhead) using small inputs, the cost is moderate. Overall, the time spent on feature extraction and model calibration accounts for 5% and 8% respectively to the total task execution time. It is worth mentioning that profiling runs also contribute to the final output of the task, so no computing cycles are wasted; and profiling is performed while the application is waiting to be scheduled.

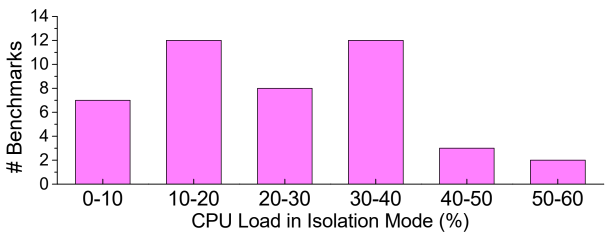


Figure 13. CPU load distributions across benchmarks when the application is executed in the isolation mode.

6.6 CPU Load in Isolation Mode

Figure 13 shows the average CPU load when a benchmark is running in isolation using all the system's memory exclusively. The CPU load for most of the 44 benchmarks is under 40%. As a result, the CPU is often not fully utilized when just running on application. This is in line with the finding reported by other researchers [2]. Our approach exploits this characteristic to improve the system throughput through task co-location.

6.7 Co-location Interferences

Interferences among Spark Benchmarks. The violin plot in Figure 14 shows the distribution of slowdown when running each of the 16 benchmarks from HiBench and BigDataBench along with each of the remaining 43 benchmarks using our scheme. The shape of the violin corresponds to the slowdown distribution. The thick black line shows where 50% of the data lies. The white dot is the position of the median. In the experiment, we first launch the target application and then use the spare memory to co-locate another competing workload. The input size of the target program is ∼280GB. As can be seen from the figure, the slowdown across applications is less than 25% and is less than 10% on average. For applications with little computation demand, such as `HB.Sort`, the slowdown is minor (less than 5%). For benchmarks with higher computation demand, such as `HB.Aggregation`, we observe greater slowdown due to competing of computing resources among co-locating tasks. Overall, our co-location scheme has little impact on the application's performance.

Interferences to PARSEC Applications. We further extend our experiments to investigate the impact for co-locating Spark tasks with other computation-intensive applications. For this purpose, we run some computation-intensive C/C++ applications from the PARSEC benchmark suite (v3.0) [3] using the large, native input provided by the suite. Figure 15 shows the slowdown distribution of each PARSEC benchmark when they run together with each of the 44 Spark benchmarks under our scheme. As all PARSEC benchmarks are share-memory programs, this experiment was conducted on a single host. As expected, we observe some slowdown to the computation-intensive PARSEC benchmark, but the slowdown is modest – less than 30%. For most of cases, the slowdown is less than 20%. Given the significant benefit on system throughput and server utilization given by our approach, we argue that such a small slowdown is acceptable when maximizing the server utilization is desired (which is typical for many data center applications). There are other

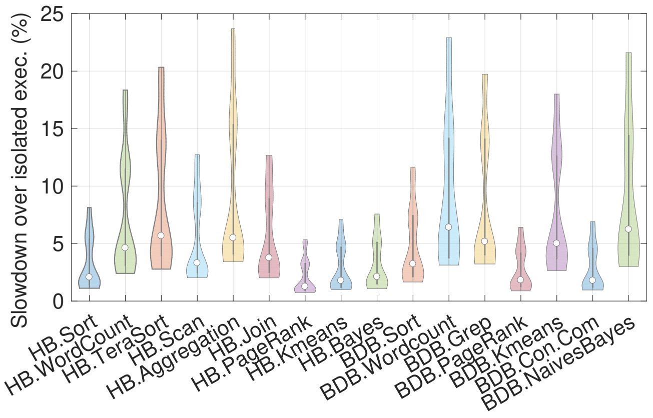


Figure 14. Violin plot showing the distribution of slowdown when using our scheme to co-locate the target benchmark with another application on a single host. The baseline is running the target application in isolation. Here we run each of the 16 target benchmarks from HiBench and BigDataBench along with each of the remaining 43 benchmarks.

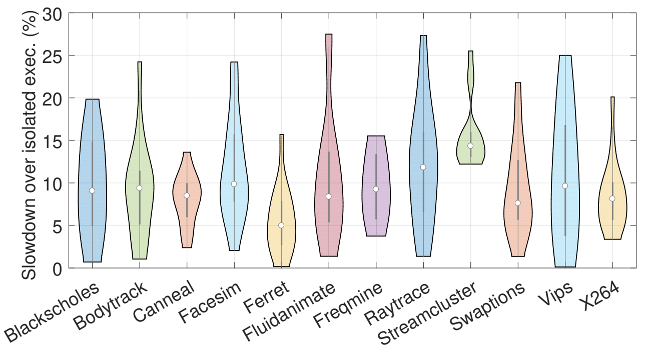


Figure 15. The slowdown distribution of computation-intensive PARSEC benchmarks when they run with a Spark task under our scheme.

schemes such as Bubble-Flux [47] for reducing the interference via dynamically pausing non-critical tasks, which are orthogonal to our scheme.

6.8 Model Analysis

Program Distribution. Figure 16 visually depicts the distribution of benchmarks on the feature space. To aid clarity, we use `PCA` to project the dimension of the original feature space down to two. Each point in the figure is one of the 44 benchmarks. This diagram clearly shows that the 44 benchmarks can be grouped into three clusters. After inspecting each cluster, we found that we indeed use the same memory function (given on the figure) for all benchmarks in a cluster. This diagram justifies the chosen number of memory functions. It also confirms our assumption that programs with similar features can be modeled using similar memory functions. We want to highlight that one of the advantages of our `KNN` classifier is that the distance used to choose the *nearest neighbor* program gives a confidence estimation of

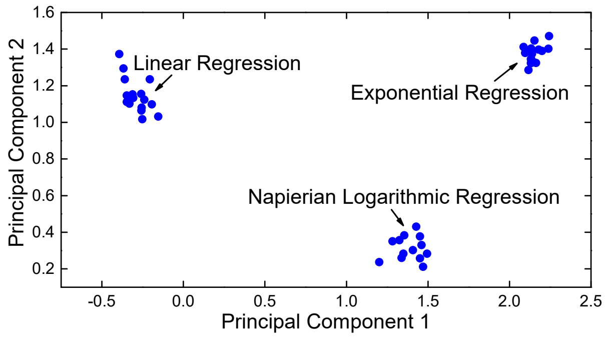


Figure 16. Program feature space. The original feature space is projected into 2 dimensions using `PCA`. Programs can be grouped into three clusters. For all benchmarks in a cluster, their memory behaviors can be described using one of the three modeling techniques in Table 1.

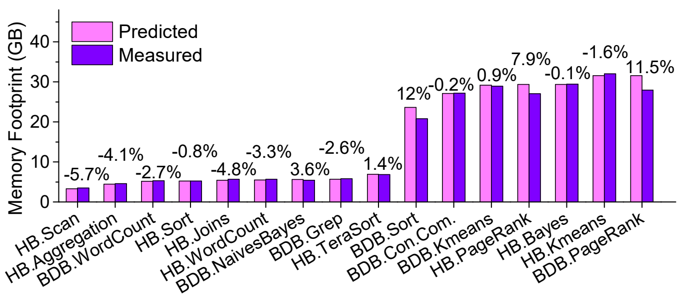


Figure 17. Predicted memory footprints vs measured values for HiBench (HB) and BigDataBench (BDB).

how good the predicted memory function will be. If the target application is far from any of the clusters in the feature space, it suggests that a new memory modeling technique will be required (and our approach allows new memory functions to be easily inserted), or a conservative co-location policy should be used to avoid saturating the memory system.

Prediction Accuracy. Figure 17 compares the predicted optimal memory allocation against the measured value, using an input size of around 280GB. The prediction error of our approach is less than 5% in most cases except for `HB.PageRank`, `BDB.PageRank` and `BDB.Sort` for which our approach over-provisions around 8% to 12% of the memory. This translates to 1.5GB to 2GB of memory. Our approach also slightly under-estimates the memory requirement for some of the benchmarks, but the difference is small so it does not significantly affect the performance. In general, the accuracy can be improved by using more training programs and more sophisticated modeling techniques to better capture the application memory requirement, which is our future work. In practical terms, one can also slightly over-provision (e.g. 10%) the memory allocation to applications with higher priorities to tolerate potential prediction errors. Overall, our approach can accurately predict the optimal memory allocation, with an average prediction error of 5%.

Table 4. Prediction accuracy for different classifiers

Classifier	Accuracy (%)	Classifier	Accuracy (%)
Naive Bayes	92.5	SVM	95.4
MLP	94.1	Rand. Decision Forests	95.5
Decision Tree	96.8	ANN	96.9
KNN	97.4		

Compare to Alternative Classifiers. Table 4 gives the memory function prediction accuracy (averaged across benchmarks and inputs) of various alternative classification techniques and our `KNN` model. The alternative models were built using the same features and training data. Thanks to the high-quality features, all classifiers are highly accurate in predicting the memory function. We choose `KNN` simply because it gives a similar prediction accuracy to alternative techniques but does not require re-training when a new memory function is added.

Memory Functions. Figure 18 compares the memory footprint given by our approach to the measured values for HiBench and BigDataBench, showing that our memory function can precisely capture the application's memory footprint. This figure also shows that a single model is unlikely to capture diverse application behaviors. We address this by developing an extensible framework into which we can easily plug-in multiple models to capture diverse application behaviors.

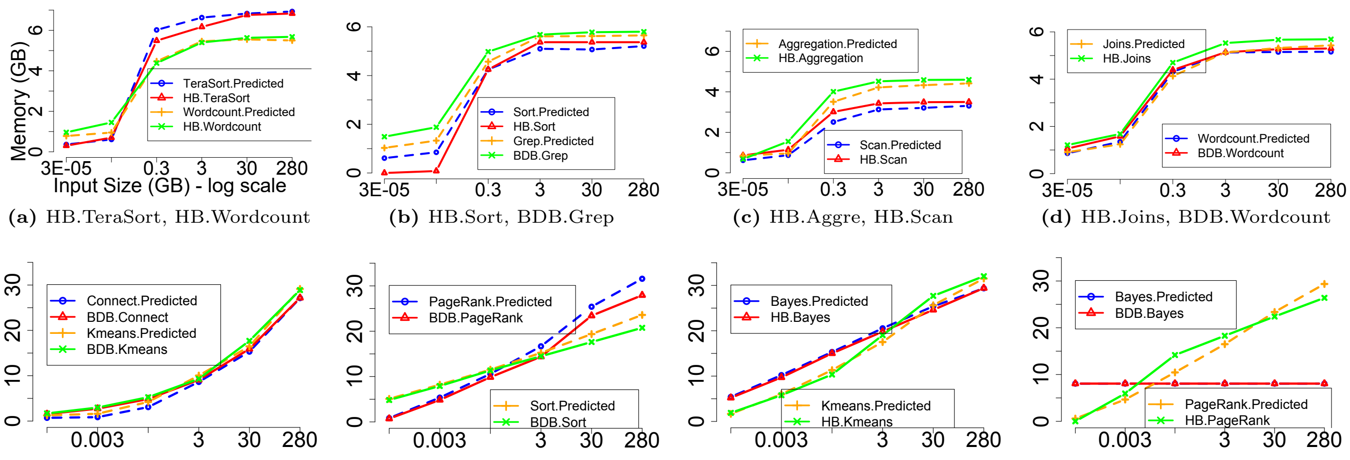
7 Related Work

Our work lies at the intersection between big data workload tuning and machine learning based system optimization.

7.1 Optimizing Big Data Workloads

Domain-specific Optimization. There exists a large body of work focusing on optimizing a single application using domain-specific knowledge. Prior work in domain-specific optimizations for single big data applications includes query optimization [4, 8, 45], graph or data flow optimization [5, 15, 26, 38], task tuning [7] and personal assistant and deep learning services [20]. By contrast, we target resource modeling of Spark applications and demonstrate that this technique is useful for scheduling multiple application tasks.

Memory Management. Numerous techniques have been proposed to manage memory resources of big data applications [40]. Many of the prior works require using dedicated APIs to rewrite the application [14, 34]. Fang *et al.* introduce Interruptible Tasks, a parallel data task that can be interrupted upon memory pressure. Their work aims to solve the out-of-memory problem when processing large amounts of data on a single host [14]. This work is thus orthogonal to our work and can be used to address the problem of occasional over-subscription of memory resources. MemTune is a recent work on heap management for Spark applications [46]. It detects memory contention and dynamically adjusts the memory partitions between Spark processes, but it does not address the problem of how much memory is needed for a given input.



(a) HB.TeraSort, HB.Wordcount **(b)** HB.Sort, BDB.Grep **(c)** HB.Aggre, HB.Scan **(d)** HB.Joins, BDB.Wordcount

(e) BDB.Con.Comp, BDB.Kmeans **(f)** BDB.Sort, BDB.PageRank **(g)** HB.Bayes, HB.Kmeans **(h)** BDB.N.Bayes, HB.PageRank

Figure 18. Comparisons of the predicted memory footprint to the measured value.

Application Scheduling. Verma *et al.* use profiling information to schedule jobs within a MapReduce application [43]. Mashayekhy *et al.* develop energy-aware heuristics to map tasks of a big data application to servers to minimize energy usage [33]. Unlike our work, all these works target scheduling jobs within a single application, and allocate all physical memory of a machine to one single application. Other work looks at mapping parallelism by determining the number of cores and process time to be allocated to an application [19]. Our method promotes memory utilization on a local host, allowing the system to perform more tasks than previously allowed with current methods. Consequently, higher multitasking levels may lead to an increase in non-local data accesses within each task; the scheduling framework in [19] is therefore complementary to our work.

Task Co-location. Prior studies in task co-location include Bubble-Flux [47], Quasar [10], Tetris [17] and Cooper [29], which co-locate tasks across machines. Other studies schedule workloads on multi-core processors [28, 50]. All the approaches mentioned above employ a single monolithic function to model the resource requirement of application tasks. There is little ability to examine whether the function fits the application under the current runtime scenario. Other fine-grained scheduling frameworks, like Mesos [21], rely on the user to provide the resource requirement of the application [21]. By contrast, we develop an extensive framework that uses multiple modeling techniques to automatically estimate the resource requirement. Our approach allows new models to be added over time to target a wider range of applications. Experimental results show that our approach yields better performance than a single model based approach. On the other hand, the co-location policies developed in these prior works for determining which two applications should co-locate are complementary to our work.

7.2 Predictive Modeling

Recent studies have shown that machine learning based predictive modeling is effective in code optimization [39] and processor resource scheduling [44]. In [12], a mixture-of-experts approach is proposed to schedule OpenMP programs on multi-cores. Their approach uses multiple linear regression models to predict the optimal number of threads to use for a given program on a single machine. Our approach differs from [12] in two aspects. First, we target a different problem (determining the memory footprint vs the number of threads) and a different scale (multiple vs a single node). Secondly, we use different modeling techniques, both linear and non-linear, to capture the memory behaviors of different applications. No work so far has used predictive modeling to model an application's memory requirement to co-locate big data application tasks. This work is the first to do so.

8 Conclusions

This paper has presented a novel scheme based on a mixture-of-experts approach to estimate the memory footprint of a Spark applications for a given dataset. Our approach determines at runtime, which of the off-line learned functions should be used to model the application's memory resource demand. One of the advantages of our approach is that it provides a mechanism to gracefully add additional expertise knowledge to target a wider range of applications.

We combine our resource prediction framework with a runtime task scheduler to co-locate latency-insensitive Spark applications. Using the accurate prediction given by our framework, a runtime task scheduler can efficiently dispatch multiple applications to run concurrently on a single host to improve the system's throughput and at the same time to ensure the total memory consumption does not exceed the physical memory of the host. Our approach is applied to 44 representative big data applications built upon Apache Spark. On a 40-node cluster, our approach achieves, on average, 86% and 94.6% of the ORACLE performance on system throughput and turnaround time, respectively.

References

[1] Ethem Alpaydin. 2010. *Introduction to Machine Learning* (2nd ed.). The MIT Press.

[2] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. 2015. Performance characterization of in-memory data analytics on a modern cloud server. In *Big Data and Cloud Computing (BDCloud), 2015 IEEE Fifth International Conference on*. IEEE, 1–8.

[3] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[4] Carsten Binnig, Norman May, and Tobias Mindnich. 2013. SQLScript: Efficiently analyzing big enterprise data in SAP HANA. In *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)*, Vol. P-214. 363–382.

[5] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 1151–1162.

[6] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. 2007. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 185–197.

[7] Dazhao Cheng, Jia Rao, Yanfei Guo, and Xiaobo Zhou. 2014. Improving MapReduce Performance in Heterogeneous Environments with Adaptive Task Tuning. In *Proceedings of the 15th International Middleware Conference (Middleware '14)*. ACM, New York, NY, USA, 97–108.

[8] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce Online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*. USENIX Association, Berkeley, CA, USA, 21–21.

[9] Databricks. 2016. Spark-Perf. (2016). https://github.com/databricks/spark-perf

[10] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 127–144.

[11] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, Grigori Fursin, and Michael F. P. O'Boyle. 2009. Portable Compiler Optimisation Across Embedded Programs and Microarchitectures Using Machine Learning. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 78–88.

[12] Murali Krishna Emani and Michael Boyle. 2015. Celebrating Diversity: A Mixture of Experts Approach for Runtime Mapping in Dynamic Environments. *SIGPLAN Not.* 50, 6 (jun 2015), 499–508.

[13] Stijn Eyerman and Lieven Eeckhout. 2010. Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling. *SIGPLAN Not.* 45, 3 (March 2010), 91–102.

[14] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. 2015. Interruptible Tasks: Treating Memory Pressure As Interrupts for Highly Scalable Data-parallel Programs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 394–409.

[15] Achille Fokoue, Oktie Hassanzadeh, Mohammad Sadoghi, and Ping Zhang. 2016. Predicting Drug-Drug Interactions Through Similarity-Based Link Prediction Over Web Data. In *Proceedings of the 25th International Conference Companion on World Wide Web (WWW '16 Companion)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 175–178.

[16] Wanling Gao, Yuqing Zhu, Zhen Jia, Chunjie Luo, and Lei Wang. 2013. Bigdatabench: a big data benchmark suite from web search engines. 1–7.

[17] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource Packing for Cluster Schedulers, In Proceedings of the 2014 ACM Conference on SIGCOMM. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 455–466.

[18] Dominik Grewe, Zheng Wang, and Michael FP OBoyle. 2013. OpenCL task partitioning in the presence of GPU contention. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 87–101.

[19] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. 2015. DjiNN and Tonic: DNN As a Service and Its Implications for Future Warehouse Scale Computers, In ISCA '15. *SIGARCH Comput. Archit. News* 43, 3 (June 2015), 27–40.

[20] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G. Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. 2015. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers, In ASPLOS '15. *SIGPLAN Not.* 50, 4 (March 2015), 223–238.

[21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. 295–308.

[22] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2011. The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In *New Frontiers in Information and Software as Services: Service and Application Design Challenges in the Cloud*, Divyakant Agrawal, K. Selçuk Candan, and Wen-Syan Li (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–228.

[23] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. 1991. Adaptive Mixtures of Local Experts. *Neural Comput.* 3, 1 (March 1991), 79–87.

[24] Tao Jiang, Qianlong Zhang, Rui Hou, Lin Chai, Sally A Mckee, Zhen Jia, and Ninghui Sun. 2014. Understanding the behavior of in-memory computing workloads. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on Workload Characterization*. IEEE, 22–30.

[25] James M. Keller and Michael R. Gray. 1985. A Fuzzy K-Nearest Neighbor Algorithm. *IEEE Transactions on Systems, Man and Cybernetics* (1985).

[26] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 31–46.

[27] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. 2015. SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers (CF '15)*. ACM, New York, NY, USA, Article 53, 8 pages.

[28] Ming Liu and Tao Li. 2014. Optimizing Virtual Machine Consolidation Performance on NUMA Server Architecture for Cloud Workloads. *SIGARCH Comput. Archit. News* 42, 3 (jun 2014), 325–336.

[29] Qiuyun Llull, Songchun Fan, Seyed Majid Zahedi, and Benjamin C Lee. 2017. Cooper: Task Colocation with Cooperative Games. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 421–432.

[30] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 450–462.

[31] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 45–55.

[32] Bryan FJ Manly. 2004. *Multivariate statistical methods: a primer*. CRC Press.

[33] Lena et al. Mashayekhy. 2015. Energy-Aware Scheduling of MapReduce Jobs for Big Data Applications. *IEEE TPDS* (2015).

[34] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 675–690.

[35] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 293–307.

[36] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, Washington, DC, USA, 1–10.

[37] Karan Singh, Major Bhadauria, and Sally A. McKee. 2009. Real Time Power Estimation and Thread Scheduling via Performance Counters. *SIGARCH Comput. Archit. News* 37, 2 (jul 2009), 46–55.

[38] Evan R Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph Gonzalez, Michael J Franklin, Michael I Jordan, and Tim Kraska. 2013. MLI: An API for distributed machine learning. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*. IEEE, 1187–1192.

[39] Kevin Stock, Louis-Noël Pouchet, and P. Sadayappan. 2012. Using Machine Learning to Improve Automatic Vectorization. *ACM Trans. Archit. Code Optim.* 8, 4, Article 50 (jan 2012), 23 pages.

[40] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. 2011. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. *SIGARCH Comput. Archit. News* 39, 3 (jun 2011), 283–294.

[41] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.* 2, 2 (aug 2009), 1626–1629.

[42] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 5, 16 pages.

[43] A. Verma, L. Cherkasova, and R. H. Campbell. 2012. Two Sides of a Coin: Optimizing the Schedule of MapReduce Jobs to Minimize Their Makespan and Improve Cluster Performance. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 11–18.

[44] Yuan Wen, Zheng Wang, and Michael FP O'Boyle. 2014. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *High Performance Computing (HiPC), 2014 21st International Conference on*. IEEE, 1–10.

[45] Cong Xu, Brendan Saltaformaggio, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. 2015. vRead: Efficient Data Access for Hadoop in Virtualized Clouds. In *Proceedings of the 16th Annual Middleware Conference (Middleware '15)*. ACM, New York, NY, USA, 125–136.

[46] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu. 2016. MEMTUNE: Dynamic Memory Management for In-Memory Data Analytic Platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 383–392.

[47] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 607–618.

[48] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2015. Computer Performance Microscopy with Shim. *SIGARCH Comput. Archit. News* 43, 3 (jun 2015), 170–184.

[49] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10.

[50] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing Shared Resource Contention in Multicore Processors via Scheduling. *SIGPLAN Not.* 45, 3 (mar 2010), 129–142.