Guided Bayesian Optimization to AutoTune Memory-based Analytics

Mayuresh Kunjir
Duke University
Email: mayuresh@cs.duke.edu

***Abstract*—There is a lot of interest today in building autonomous (or, self-driving) data processing systems. An emerging school of thought is to leverage the "black box" algorithm of Bayesian Optimization for problems of this flavor both due to its wider applicability and theoretical guarantees on the quality of results produced. The black-box approach, however, could be time and labor-intensive; or otherwise get stuck in a local minima. We study an important problem of auto-tuning the memory allocation for applications running on modern distributed data processing systems. A simple "white-box" model is developed which can quickly separate good configurations from bad ones. To combine the benefits of the two approaches to tuning, we build a framework called Guided Bayesian Optimization (GBO) that uses the white-box model as a *guide* during the Bayesian Optimization exploration process. An evaluation carried out on Apache Spark using industry-standard benchmark applications shows that GBO consistently provides performance speedups across the application workload with the magnitude of savings being close to 2x.**

***Index Terms*—data analytics; auto tuning**

I. Introduction

A. AutoTuning Techniques

Data analytics systems, such as Spark, Flink, and Tez, support a variety of computational patterns such as Map-Reduce or Iterative Machine Learning. Each of these systems provide multiple tunable configuration options to optimize each use case. The problem of finding the best configuration of such a tunable system can be specified by the following maximization (or minimization) objective f:

$$\mathbf{x}^* = \arg\max_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x}) \qquad (1)$$

where $\mathcal{X}$ is the space of features/configuration options.

If the optimization function f is known apriori, e.g. a linear combination of features, regression-based optimization can be used to solve the problem. Such functions are referred to as *white-box* optimization models. DB2 tuning advisor [1] is a classical example of auto-tuners using white-box models. Modern white-box auto-tuners use feedback-based techniques to recalibrate and revalidate the model in order to support dynamic system changes, e.g. Starfish [2] and Ernest [3]. However, building parametric objective functions is often non-trivial in presence of complex interactions among configuration options and performance metrics [4].

Black-box optimizers are employed when the objective function is not known apriori. Black-box models learn the objective function by sampling and observing points from the configuration space. Bayesian optimization [5] is a powerful black-box technique that is applied to varied designs including Database systems [6], [7], Big data analytics [8], [9], Storage systems [10], and Cloud infrastructures [11], [12]. Bayesian Optimization (BO) first prescribes a prior belief on the probability distribution of f (e.g. Gaussian). It then sequentially updates its belief by learning the posterior distribution from observing samples from configuration space $\mathcal{X}$. In each iteration, the optimizer suggests the next sample to probe using an *acquisition function* which balances exploration (i.e. acquiring new knowledge) and exploitation (i.e. using existing knowledge in decision making). This way, BO provides a theoretically justified means of searching for optimal configuration making it an attractive choice for auto-tuning.

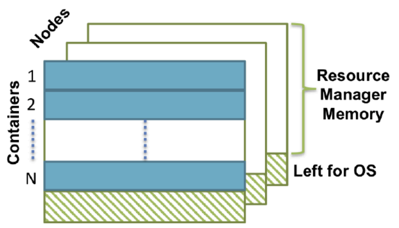

Fig. 1: Node memory managed by Resource Manager

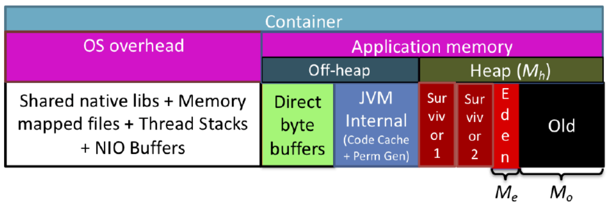

Fig. 2: Container memory managed by JVM

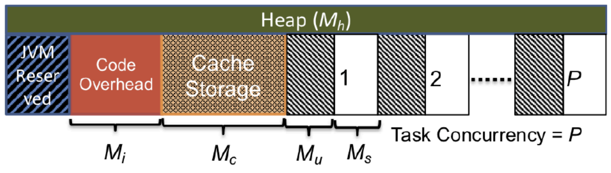

Fig. 3: Heap managed by application framework

B. Memory-based Analytics

Data analytics systems employ a resource manager, such as Yarn, to allocate system resources in the form of *containers* to requesting applications. A container is simply a slice of physi-

TABLE I: Parameters controlling memory pools across multiple levels: Container, Application Framework, and JVM displayed in order from top to bottom.

Parameter	Description	Pool(s) controlled
Heap Size	Heap size in a container	Heap (M_h)
Cache Capacity	Cache storage as a fraction of Heap	Cache Storage (M_c)
Shuffle Capacity	Shuffle memory as a fraction of Heap	Task Shuffle (M_s)
Task Concurrency	Number of tasks running concurrently	Task Unmanaged (M_u)
NewRatio	Ratio of Old capacity to Young capacity	Old (M_o)
SurvivorRatio	Ratio of Eden capacity to Survivor space	Eden (M_e)

cal resources carved out of a node allocated exclusively to the application. Fig. 1 shows the cluster memory organization.

Many popular data analytics systems (e.g., Spark, Flink, and Tez) use a JVM-based architecture for memory management. For applications running on these systems, a JVM process is executed inside each allocated container. As shown in Fig. 2, the container memory is divided into two parts: (a) Memory available to the JVM process, and (b) An overhead space used by the operating system for process management. The JVM further divides its allocation into a heap space and an off-heap space. All objects, except native byte buffers, created by the application code are allocated on Heap and are managed by the JVM's generational heap management.

Fig. 3 shows how Heap is organized into different pools from the application's perspective. Memory used by an application can be broadly categorized into three pools:

1) *Code Overhead*: Memory required for application code objects. Treated as a constant overhead.
2) *Cache Storage*: Memory used to store the data cached by application. In particular, storing intermediate results in memory is beneficial during iterative computations.
3) *Task Memory*: The rest of the memory is used by application tasks. The number of tasks running concurrently is set as a configuration parameter, Task Concurrency, which determines the share of memory each task gets to use. A task uses its allocation for two purposes: (a) Memory for shuffle processing tasks such as sort and aggregation (M_s), (b) Memory for input data objects and serialization/deserialization buffers (M_u).

Users of data analytics systems expect to achieve the best possible latency (wall clock duration) for their applications. For periodically running applications, the reliability of performance is also an important factor. An application can be tuned at multiple levels: (a) while allocating resources from the resource managers; (b) while setting options provided by the application framework relating to the internal memory pools management; and (c) while configuring JVM parameters related to garbage collection of heap. TABLE I provides a summary of parameters controlling usage of memory pools in—and effectively impacting the performance of—memory-based analytics systems.

Depending on the computations involved or the type of data processed, best settings at each level of memory management can differ a great deal. In a recent work [13], we showed how the robust default settings provided by systems [14], [15] do not always work well. In the same work, we developed an empirically-driven rule-based auto-tuner for memory pools, called RelM. RelM uses a set of analytical models driven by low-level statistics to recommend a configuration expected to provide a near-optimal and reliable performance. The statistics used by RelM correspond to memory requirements for various internal pools within an application which makes it robust to changes in computational patterns and input data design. It is shown to produce high quality of results within a fraction of time taken by state-of-the-art Bayesian Optimizers. However, RelM, like the white-box models discussed earlier, is reliant on the accuracy of the statistics collected and can produce results far off from optimal in certain rare events.

C. Our Contributions

We have learned from our experiencing of building RelM [13] that a simple white-box tuner can *quickly* produce decent results. The black-box approach of Bayesian Optimization (BO), however, offers other benefits including wider applicability and theoretically-guaranteed convergence to the optimal settings. Motivated by this, we develop a framework called 'Guided Bayesian Optimization' (GBO) combining the benefits of the two. GBO supplements a Bayesian Optimizer (BO) with an approximate white-box model capable of separating good configurations from bad ones in quick time. The BO in GBO is modeled as a Gaussian Process (GP) [16]. The model is bootstrapped with a small number of pre-executed samples taken from Latin Hypercube Sampling [17]. The same set of samples is used to bootstrap a white-box model with required low-level statistics. During an iteration of sequential tuning, GBO recommends a configuration to probe next which both maximizes the acquisition function over the current posterior of the GP and is expected to perform well according to our white-box model as well. Section II details the GBO framework. Following that, Section III discusses the white-box model we have used in our system prototype.

D. Related Work

One approach towards auto-tuning has been to assume a shape of optimization objective, e.g. linear or quadratic. Using this approach, some researchers have used regression models [18], while others have used hill-climbing techniques [19]. Bayesian optimization works without a need of such assumptions and is, therefore, more versatile. We use Gaussian Process (GP) Regression [16] as our candidate BO. There are alternative techniques in literature, such as ensemble random forest [20], which have been used for systems employing BO [12]. However, unlike GP, none of them provide an estimate of the variance of its predictions.

The problem of speeding up a black-box bayesian optimizer with white-box models is fairly recent. Dalibard et. al. [21]

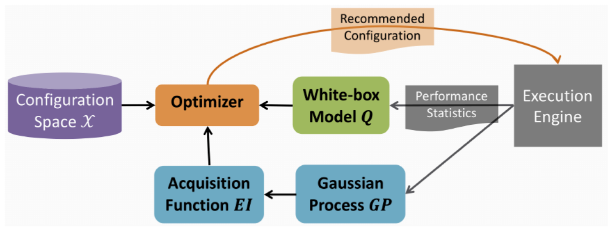

Fig. 4: Workflow of Guided Bayesian Optimization (GBO)

propose *Structured Bayesian Optimization* (SBO) which lets system developers add structure to the optimizer by means of bespoke probabilistic models including simple parametric models inferred from low-level performance metrics observed during a tuning run. The combination of non-parametric bayesian optimizer and evolving parametric models help with a faster system convergence. SBO needs considerable system expertise to design a combination model for auto-tuning. Our focus, instead, is restricting exploration of bayesian optimizer to a smaller space of configurations identified by simple white-box models.

Another recent work specifically targeted at finding best VM configurations, Arrow [12], augments a bayesian optimizer with not only VM characteristics but also low-level performance metrics. Providing additional features to BO, however, increases the dimensionality of the problem. In smaller dimensional configuration spaces such as ours, the benefits of adding structure by means of low level information are far outweighed by the extra exploration necessitated by the higher dimensionality. Our approach, on the other hand, can help exploit system knowledge without extra exploration overheads.

The idea of combining the black-box acquisition function with a white-box model is similar to an approach of managing a *portfolio* of acquisition functions for BO [22]. While the motivation behind the portfolio management is to find a schedule of acquisition functions that maximizes rewards during a tuning run, our motivation is to incorporate knowledge of system behavior through white-box models.

II. Guided Bayesian Optimization

We have described the Guided Bayesian Optimization (GBO) framework in brief in Section I. Fig. 4 outlines the workflow of tuning process. The configuration space used during optimization comes from the parameters listed in TABLE I. We first describe the BO model used in our framework before outlining the process of selecting next configuration to probe.

BO model requires two building blocks: (a) *Bayesian prior* prescribes a prior belief over the possible objective functions, and (b) *Bayesian posterior* provides a mechanism to sequentially update the belief by learning from new observations. Since the objective function is unknown, we need to use a non-parametric model. Gaussian Process [16] is a popular choice because of its salient features such as support for noisy observations and ability to use gradient-based methods [23]. Using Gaussian Process, the prior belief is modeled as $f(\mathbf{x}) \sim GP(\mu_0, k)$, where $\mu_0 : \mathcal{X} \to \mathbb{R}$ denotes the prior mean function and $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ denotes the covariance function. Given n sampled points $\mathbf{x}_{1:n}$ and noisy observations $y_{1:n}$ (σ^2 denoting a constant observation noise), the unknown function values $\mathbf{f} := f_{1:n}$ are assumed to be jointly Gaussian, i.e. $\mathbf{f}|\mathbf{x} \sim \mathcal{N}(\mathbf{m}, \mathbf{K})$, and the observations $\mathbf{y} := y_{1:n}$ are normally distributed given $\mathbf{f}$, i.e. $\mathbf{y}|\mathbf{f}, \sigma^2 \sim \mathcal{N}(\mathbf{f}, \sigma^2\mathbf{I})$. The posterior mean and variance are then given by the following:

$$\begin{aligned} \mu_n(\mathbf{x}) &= \mu_0(\mathbf{x}) + \mathbf{k}(\mathbf{x})^\top(\mathbf{K} + \sigma^2\mathbf{I})^{-1}(\mathbf{y} - \mathbf{m}) \\ \sigma_n^2(\mathbf{x}) &= k(\mathbf{x}, \mathbf{x}) - \mathbf{k}(\mathbf{x})^\top(\mathbf{K} + \sigma^2\mathbf{I})^{-1}\mathbf{k}(\mathbf{x}) \end{aligned} \tag{2}$$

where $\mathbf{k}(\mathbf{x})$ is a vector of covariance between $\mathbf{x}$ and $\mathbf{x}_{1:n}$.

An acquisition function provided by BO suggests the next sample to probe based on the posterior distribution. We use one of the most popular acquisition functions, Expected Improvement (EI), given below:

$$EI(\mathbf{x}; \mathbf{x}_{1:n}, y_{1:n}) = (\tau - \mu_n(\mathbf{x}))\Phi(Z) + \sigma_n(\mathbf{x})\phi(Z) \tag{3}$$

Here, τ denotes the current best observation, $Z = (\tau - \mu_n(\mathbf{x}))/\sigma_n(\mathbf{x})$, and Φ and ϕ are standard normal cumulative distribution and density functions respectively. The next sample will be either picked from a region where uncertainty is high, captured by $\sigma_n(\mathbf{x})$, or from a region close to the current best, captured by $(\tau - \mu_n(\mathbf{x}))$, thus balancing the *exploration* and the *exploitation*. The GP suggests next sample where the expected improvement is the highest.

GBO uses a white-box model $Q : \mathcal{X} \to \mathbb{R}$ which is driven by low-level performance statistics and outputs a utility score that helps rank the configurations in terms of expected performance. Q does not necessarily model the exact system behavior because of the inaccuracy in statistics as well as a possible mismatch between the assumed function and the true interactions. As part of its initialization, GBO observes a small number of samples taken using Latin Hypercube Sampling (LHS) [17] over the domain space of configuration options. LHS is an efficient technique to generate near-random samples from a multidimensional space providing a good coverage. These samples are used to bootstrap both the Gaussian process and the white-box model used as a guide. Both models can improve themselves as more samples are observed during a tuning run.

GBO modifies exploration for the next best point: Using Q, it prunes out the configurations expected to produce poor performance. The change is given by the equation below.

$$\mathbf{x}_{n+1} = \arg\max_{\mathbf{x} \in \mathbf{X}} EI(\mathbf{x}, \mathcal{D}_n).I(\mathbf{x}) \tag{4}$$

where $\mathcal{D}_n$ is a database of n samples including the configurations and the corresponding observations. The optimizer uses a small number of uniform random samples and a few invocations of quasi-Newton hill climbers (e.g. L-BFGS [24]) to explore the space of unseen configurations ($\mathbf{X}$).

$I(\mathbf{x}) \sim \{0, 1\}$ is a boolean function telling whether the configuration is worthy of exploration or not. Algorithm 1

Algorithm 1 Configuration Pruner

```
    Input: White box function $Q$
    Input: Configurations to be explored $\mathbf{X}$
    Input: Query configuration $\mathbf{x}$
    Output: $\{0, 1\}$
1: Set $low = \min_{\mathbf{x}' \in \mathbf{X}} Q(\mathbf{x}')$
2: Set $high = \max_{\mathbf{x}' \in \mathbf{X}} Q(\mathbf{x}')$
3: if $Q(\mathbf{x}) \geq U([low, high])$ then
4:     Return 1
5: else
6:     Return 0
7: end if
```

details how white box function Q is used in decision making. Q is evaluated on each of the configurations chosen for exploration by the optimizer. The configurations with high Q values are made more likely to be considered by putting a high probability mass on them. The idea behind probabilistically picking (or pruning) a configuration is to have a lesser dependence on accuracy of white-box model. The other option, of picking configurations with Q scores above certain value (say 70 percentile), makes the model completely reliant on white-box model. GBO makes a conscious choice of primarily depending on black-box acquisition function because it makes a more informed decision as it explores more samples; whereas the predictions of white-box function may remain inaccurate.

III. White-box Model for Memory Pools

We build a closed-form prototype utility model by understanding memory pool management in data analytics systems. The understanding is based on a systematic empirical study carried out in project RelM [13]. The utility model relies on the statistics generated from profiles of applications sampled apriori as part of GBO bootstrapping. Although, the statistics could be updated as more samples are observed, we do not consider this approach here. Section IV talks of merits/demerits of this choice through evaluation. The utility model we build have two characteristics:
(a) It assigns a high utility to the memory pool allocations meeting application requirements, and
(b) It penalizes the memory allocations that are either expected to result in *out-of-memory* errors or lead to high garbage collection costs.

A. Statistics Generation

We use Thoth [25] framework to obtain a profile of the application which includes JVM logs, application event logs, and resource monitors. TABLE II lists the statistics derived from an application profile. The first two entries correspond to the configuration of Container used to run the profiled application. The requirement for Code Overhead pool (M_i) is obtained by looking up heap usage value at the instance of the first task submission to the container. This value corresponds to the memory required for application code objects and is expected to be occupied through the lifetime of the container.

TABLE II: Statistics derived from an application profile

Notation	Description	Example
N	Containers per Node	1
M_h	Heap size	4404MB
M_i	Code Overhead 90%ile value	130MB
M_c	Cache Storage 90%ile value	2300MB
M_s	Task Shuffle 90%ile value	0MB
M_u	Task Unmanaged 90%ile value	70MB
P	Task Concurrency	2
H	Cache Hit Ratio (the fraction of cached data partitions actually read from cache)	0.7
S	Data Spillage Fraction (the fraction of shuffle data spilled to disk)	0

The values obtained from multiple containers in an application profile could have some variance, so we use a 90th percentile value as a stability against outliers. The memory used by Cache Storage (M_c) is computed by looking up the maximum cache usage value from the profile. This cache usage value may not necessarily correspond to the actual requirement because the application could possibly be configured with an under-sized cache. We record Cache Hit Ratio (H) from application logs in order to evaluate the actual requirement.

While both M_i and M_c are considered long term memory requirements of a container, the memory used for task execution ($M_s + M_u$), corresponds to short-term memory requirements. We assume that each task running concurrently equally contributes to the total task memory. This assumption helps us derive the value of M_s. Like in the case of M_c, the shuffle memory usage value does not necessarily correspond to the actual requirement of the application since the shuffle data could possibly have been spilled because of capacity constraints. Data spillage fraction (S) allows us to estimate the actual memory requirement. The M_u value is the hardest to obtain among the statistics presented in Table II since the application does not track this memory pool. We use JVM instrumentation to get a good estimate as explained next.

As described in Section I, JVM uses two garbage collection processes, namely, *young GC* and *full GC*, to collect any unreferenced objects from Heap. The *full GC* event cleans up garbage both from young generation and old generation pools. Monitoring heap usage right after a *full GC*, therefore, gives us a more accurate picture of task memory requirements. With this idea, we monitor every *full GC* event during the runtime of the application to obtain an estimate of M_u. More details on this and other statistics generation methodology are available in our technical report [13].

Example. *Statistics for* K-means *benchmark application executed on a Spark cluster are listed in the third column of TABLE II. It can be noticed that the application has a high Cache Storage requirement compared to Task Memory requirements.*

B. Utility Evaluation

Given a test configuration $\mathbf{x}$ and the statistics derived by bootstrapping process, we evaluate the utility of $\mathbf{x}$ using an analytical model given next. We use upperscript $\mathbf{x}$ to denote

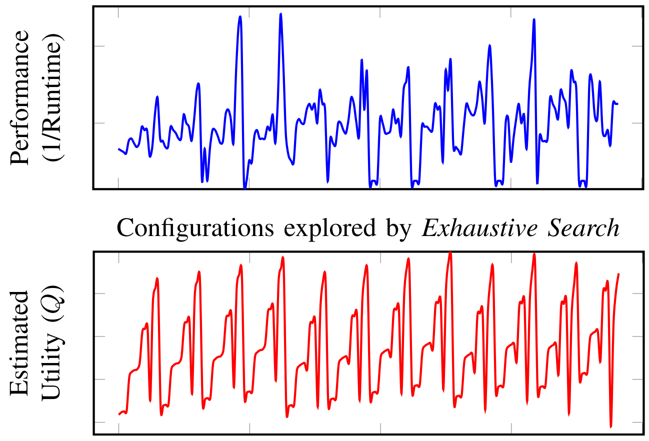

Fig. 5: Accuracy of White-box model estimates for K-means. Configurations explored by gridding the configuration space and running them exhaustively (see TABLE V) are presented on X-axis.

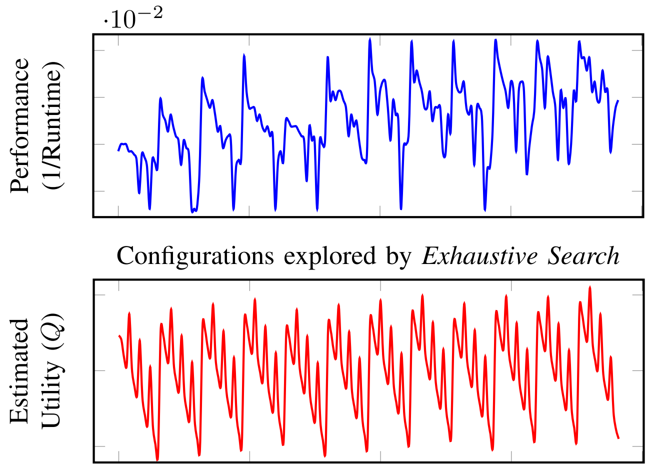

Fig. 6: Accuracy of White-box model estimates for Sort. Configurations explored by gridding the configuration space and running them exhaustively (see TABLE V) are presented on X-axis.

either the parameter values or the functions evaluated for configuration $\mathbf{x}$.

$$M_{cr} = M_c/H$$
$$M_{sr} = M_s/(1-S/P)$$
$$R^{\mathbf{x}} = \frac{M_i + P^{\mathbf{x}} * M_u + min(M_c^{\mathbf{x}}, M_{cr}) + min(M_s^{\mathbf{x}}, M_{sr})}{M_h^{\mathbf{x}}}$$
$$P1^{\mathbf{x}} = c_1 * min(0, R^{\mathbf{x}} - 1)$$
$$P2^{\mathbf{x}} = c_2 * min\left(0, \frac{M_i + min(M_c^{\mathbf{x}}, M_{cr}) - M_o^{\mathbf{x}}}{M_h^{\mathbf{x}}}\right)$$
$$P3^{\mathbf{x}} = c_3 * min\left(0, \frac{P^{\mathbf{x}} * M_u + min(M_s^{\mathbf{x}}, M_{sr}) - M_e^{\mathbf{x}}}{M_h^{\mathbf{x}}}\right)$$
$$Q^{\mathbf{x}} = R^{\mathbf{x}} - P1^{\mathbf{x}} - P2^{\mathbf{x}} - P3^{\mathbf{x}}$$

We first evaluate the requirements for Cache Storage (M_{cr}) and Task Shuffle (M_{sr}) memory pools using the statistics obtained. Here, each concurrently running task is assumed to contribute equally to data spillage. Next, we compute the total utilization of the application level memory pools as a fraction of heap size ($R^{\mathbf{x}}$). This value could exceed 1 if the memory is over-allocated. We penalize such configurations using function $P1$. Two more penalty functions, viz. $P2$ and $P3$ are used to penalize the long term memory usage exceeding JVM's Old generation pool capacity and the short term memory usage exceeding JVM's Eden pool capacity. Both the functions correspond to the garbage collection overheads. Finally, function $Q^{\mathbf{x}}$ outputs the utility of the configuration $\mathbf{x}$.

Penalty factors c_1, c_2, and c_3 correspond to the actual magnitude of the penalty. Inferring these factors accurately is an equally hard problem to the auto-tuning problem at hand. However, since the white box model is only used as a heuristic in GBO, it is sufficient to set penalty factors that could only distinguish good configurations from bad ones without accurately modeling the performance. We set each of the factors to 2 in our evaluation. Fig. 5 and Fig. 6 present the utilities estimated by the white box model compared with the actual performance on exhaustively searched configurations on two applications: K-means and Sort. Visual inspection shows that the model is capable of distinguishing the best performing and the worst performing configurations apart. Furthermore, the graphs show that the shape of the objective function can be widely varied across applications substantiating the main motivation behind our work.

IV. Evaluation

A. Setup

We carry our evaluation on a Spark cluster configured as listed in TABLE III. We use five benchmark applications for evaluation which represent Map and Reduce computations, machine learning, distributed graph processing, and SQL processing use cases. The test suite including input data sources is provided in TABLE IV.

Configuration Space: The configuration options we tune correspond to the parameters controlling memory pools listed in TABLE I. The maximum heap available for allocation per node is 4404MB. We allow it to be distributed equally among 1, 2, 3, or 4 Containers. We limit the number of concurrently running tasks on a node to the number of physical CPU cores (=8). Therefore, the Task Concurrency can range from 1 to the ratio of the number of physical CPU cores to the number of containers on the node. For example, if 2 containers are launched on a node, Task Concurrency on each container ranges from 1 to 4. Cache Capacity and Shuffle Capacity values are set as a fraction (ranging from 0 to 1) of Heap size. As Spark provides a unified memory pool [26] combining both Cache Storage and Task Shuffle, we set the capacity of the unified pool to the sum of Cache Capacity and Shuffle Capacity. The lowest possible value for NewRatio is 1. The maximum, while unbounded in theory, is limited to 9 (i.e. at

TABLE III: Test cluster setup

Number of worker nodes	8
Memory per worker	6GB
CPU cores per worker	8
Compute Framework	Spark-2.0.1
Resource Manager	Yarn-2.7.2
JVM Framework	OpenJDK-1.8.0

TABLE IV: Test suite used in evaluation

Application	Category	Dataset
K-means	Machine Learning	HiBench [28] Huge (100M samples, 20 dimensions, 5 clusters)
Sort	Map and Reduce	Hadoop RandomTextWriter (30GB)
WordCount	Map and Reduce	Hadoop RandomTextWriter (50GB)
SVM	Machine Learning	HiBench huge (100M examples, 10 features)
PageRank	Graph	LiveJournal dataset [29] (69M edges, 5M vertices)

TABLE V: Sample space used in *Exhaustive Search*.

Containers per Node	[1, 2, 3, 4]
Concurrent tasks per node[1]	[2, 4, 6, 8]
Cache Capacity/ Shuffle Capacity	[.2, .4, .6, .8]
NewRatio	[1, 3, 5, 7]

least 10% of the heap is available to the young generation) in our setup. We keep the SurvivorRatio to its default value.

Exhaustive Search: In order to find an optimal configuration, we carry out an exhaustive search by gridding the configuration space. The domain of each parameter is discretized into 4 values as listed in Table V. We use only one of Cache Capacity and Shuffle Capacity depending on the dominant requirement of the application under test in order to speed up the process. The minor memory pool capacity is kept constant at 10% of the heap.

Bayesian Optimization (BO): As detailed in Section II, we use Gaussian Process (GP) Regression as our candidate for black-box tuning. The GP is implemented using *scikit-learn* library in Python [27]. Like in the case of *Exhaustive Search*, we use only the dominant memory pool between Cache Storage and Task Shuffle during optimization. The objective function is set to minimize latency, or alternately, maximize the inverse of the runtime. If a run is aborted due to errors, we set the objective value for the sample to twice the worst runtime obtained on the samples explored so far.

White-box Optimization (WO): The white-box model we have detailed in Section III is used for exploration in this policy. During each iteration, a small random subset of configurations (about 10%) is evaluated using the white-box model Q. Configuration with the best score is chosen for exploration. We do not use inference from previously executed configurations to improve the model.

Guided Bayesian Optimization (GBO): The GBO policy detailed in Section II is used with the white-box model Q set as a guide for exploration.

[1]Task Concurrency will be determined by dividing this value by the number of Containers per Node.

B. Convergence

We evaluate how various tuning policies fare in terms of the speed of convergence to the optimal configuration. Exhaustive Search takes one to two orders of magnitude longer to converge compared to the other three policies. We do not include its results in the graphs in order to focus on our models. The rest of the policies adaptively sample up to 50 configurations in a run; 5 such runs are carried out for each application. We analyze how much of a training overhead would a policy require if it were to carry on until we find a configuration providing a performance within top 5 percentile of the exhaustively searched configurations. Although this stopping criteria is not practical because the optimal performance is not known apriori, we use it in order to compare how long do different policies take to converge to a good configuration.

Fig. 7 plots the training time and the number of iterations required for each tuning approach. Training times are normalized to the time taken by BO policy on the same application. It can be noticed that the WO policy shows a wide variation in performance: While it provides quick tuning for some applications, e.g. K-means; it leads to huge performance degradations on some others, e.g. SVM. On the other hand, GBO policy consistently lowers training time across applications compared to BO. On average, GBO reduces the number of iterations to close to half the iterations needed by the black-box policy.

To dive deep into how the policies function, we provide convergence plots for three applications, viz. K-means, SVM, and PageRank, in Fig. 8. The plot on K-means shows that the BO policy takes 8 iterations on average to find a configuration that runs within 11 minutes (top 5 percentile barrier). The exploration until then often results in configurations with expensive runtimes reflected by its high training time. The WO and GBO policies avoid such configurations and, therefore, can find a top configurations with low overheads. The white-box function Q closely mimics the performance on K-means, as seen in Fig. 5 earlier, which helps both the policies.

The SVM application throws a curious result where WO takes more than twice the time required for BO. This is a case of white-box model Q bootstrapped with incorrect statistics. We expand on this issue in the next subsection. The important thing to note here is that, despite the substandard white-box model, GBO policy provides a performance similar to the black-box tuning policy. This is a direct consequence of our design decision to rely on acquisition function used in BO for ranking the configurations under exploration.

The PageRank application presents a case wherein WO takes longer to converge despite the white-box model using correct statistics. In this case, many configurations fail with out-of-memory errors due to very high task memory requirements of the application. Although, the white-box model Q penalizes such configurations, it cannot correctly attribute the magnitude of this penalty. Due to this inaccuracy, the WO tuner is able to find a configuration within top 10

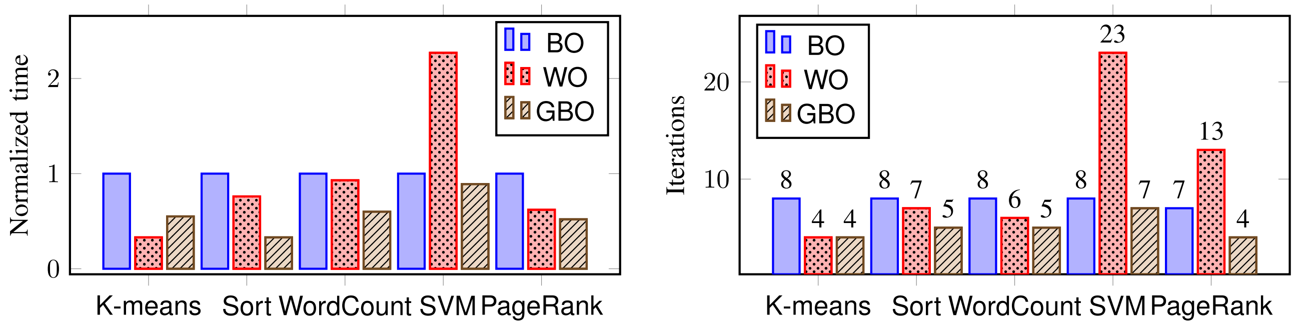

Fig. 7: Analyzing training overheads for various tuners when the tuning is set to stop as soon as a configuration is found having performance within top 5 percentile of the configurations explored by *Exhaustive Search*. The first plot shows training time normalized to time taken by BO. The second plot shows the number of iterations required.

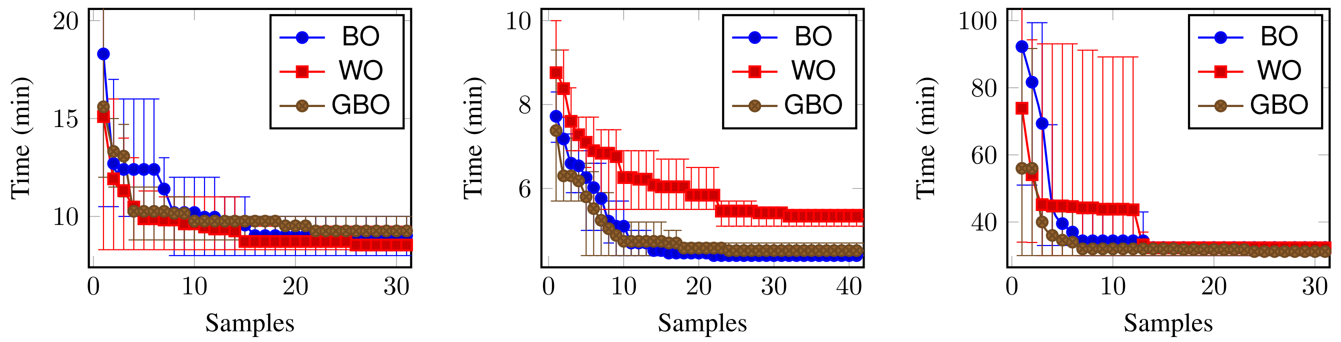

Fig. 8: Convergence of tuning policies for K-means, SVM, and PageRank from left to right. Each tuner is run 5 times; the mean, min, and max values for the lowest runtime on the samples observed so far are plotted on Y-axis.

percentile within 2-3 iterations, but struggles to find a better configuration post that. On the positive side, it avoids exploring very expensive configurations reflected by its lower training time compared to BO. The GBO policy, like in the case of SVM, brings the best of both worlds to produce a desired configuration within 4 iterations on average.

C. Quality of White-box Model

We showed the quality of our white-box model by comparing the estimated utilities to the actual performance observed on the exhaustively searched configurations on two applications: K-means (Fig. 5) and Sort (Fig. 6). Q is able to distinguish good configurations from bad ones in both cases correctly. In case of SVM, however, the white-box model Q makes wrong predictions. This happens due to inaccurate estimation of task memory requirements during bootstrapping. Recalling the statistics generation methodology from Section III, the 'Task Unmanaged' memory pool requirement is estimated by monitoring JVM's *full GC* events. The configurations considered during bootstrapping happen to contain very few such events which leads to an over-estimation of the task memory requirement.

We plot the utility estimated with incorrect statistics along with the actual performance for comparison in Fig. 9. It can be seen that, due to over-estimation of task memory requirements, the model incorrectly suggests a performance degradation with increasing Cache Capacity. When we gather better statistics after observing more configurations, the model correctly predicts the actual performance pattern. This evaluation showcases a need to update the white-box model using feedback. We plan to work on a systematic feedback mechanism to improve tuning at runtime in future.

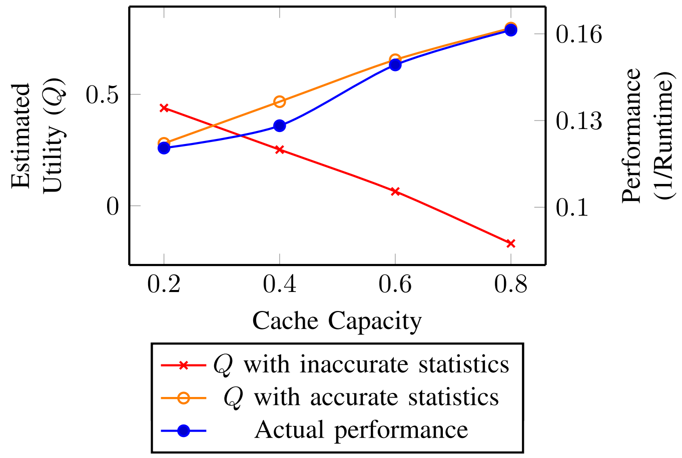

Fig. 9: Graph showing that the white-box function Q modeled with accurate statistics mimics the true performance better. Configuration option 'Cache Capacity' alone is varied on X-axis keeping other configuration options constant.

V. Discussion and Future Work

We showed how black-box auto-tuner can be sped up by using simple white-box models built using low-level performance metrics to guide the exploration. We identified an important problem of auto-tuning memory pool configurations in data analytics systems to demonstrate our work. The GBO framework we developed is flexible enough to incorporate white-box models of varying degree of accuracy anytime during the tuning process. As part of the future work, we would like to work on improving the black-box optimization by means of hyperparameter tuning of the Gaussian Process model we have used in GBO. In addition, we would like to explore other system tuning applications suitable for GBO framework.

Acknowledgments

The author would like to thank Dr. Shivnath Babu for his useful feedback on system design.

References

[1] E. Kwan, S. Lightstone, K. B. Schiefer, A. J. Storm, and L. Wu, "Automatic database configuration for db2 universal database: Compressing years of performance expertise into seconds of execution," in *BTW*, 2003.

[2] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics," in *In CIDR*, pp. 261–272, 2011.

[3] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, (Santa Clara, CA), pp. 363–378, USENIX Association, 2016.

[4] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback, "Self-tuning database technology and information services: From wishful thinking to viable engineering," in *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pp. 20–31, VLDB Endowment, 2002.

[5] J. Mockus, *Bayesian approach to global optimization: theory and applications*. Mathematics and its applications (Kluwer Academic Publishers).: Soviet series, Kluwer Academic, 1989.

[6] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned," *PVLDB*, vol. 2, no. 1, pp. 1246–1257, 2009.

[7] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pp. 1009–1024, 2017.

[8] P. Jamshidi and G. Casale, "An uncertainty-aware approach to optimal configuration of stream processing systems," *CoRR*, vol. abs/1606.06543, 2016.

[9] L. Bao, X. Liu, and W. Chen, "Learning-based automatic parameter tuning for big data analytics frameworks," *CoRR*, vol. abs/1808.06008, 2018.

[10] Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok, "Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, (Berkeley, CA, USA), pp. 893–907, USENIX Association, 2018.

[11] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, (Boston, MA), pp. 469–482, USENIX Association, 2017.

[12] C. Hsu, V. Nair, V. W. Freeh, and T. Menzies, "Arrow: Low-level augmented bayesian optimization for finding the best cloud VM," in *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*, pp. 660–670, IEEE Computer Society, 2018.

[13] "RelM Report. https://users.cs.duke.edu/~mayuresh/relm-report.pdf."

[14] "Spark configuration. https://bit.ly/2rXR4NK."

[15] "Amazon EMR Documentation. https://amzn.to/2zrpNtt."

[16] C. E. Rasmussen, "Gaussian processes for machine learning," MIT Press, 2006.

[17] C. Ireland, "Fundamental concepts in the design of experiments," *Technometrics*, vol. 7, no. 4, pp. 652–653, 1965.

[18] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. H. Tung, "A new approach to dynamic self-tuning of database buffers," *Trans. Storage*, vol. 4, pp. 3:1–3:25, May 2008.

[19] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, "A smart hill-climbing algorithm for application server configuration," in *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, (New York, NY, USA), pp. 287–296, ACM, 2004.

[20] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, pp. 5–32, Oct. 2001.

[21] V. Dalibard, M. Schaarschmidt, and E. Yoneki, "BOAT: Building auto-tuners with structured bayesian optimization," in *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, (Republic and Canton of Geneva, Switzerland), pp. 479–488, International World Wide Web Conferences Steering Committee, 2017.

[22] M. Hoffman, E. Brochu, and N. de Freitas, "Portfolio allocation for bayesian optimization," in *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, UAI'11, (Arlington, Virginia, United States), pp. 327–336, AUAI Press, 2011.

[23] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, pp. 148–175, 2016.

[24] R. H. Byrd, P. Lu, and J. Nocedal, "A limited-memory algorithm for bound constrained optimization," *SIAM Journal on Scientific Computing*, 1994.

[25] M. Kunjir and S. Babu, "Thoth in action: Memory management in modern data analytics," *Proc. VLDB Endow.*, vol. 10, pp. 1917–1920, Aug. 2017.

[26] "Deep dive: Apache spark memory management https://bit.ly/2C2x1YH."

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[28] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pp. 41–51, March 2010.

[29] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.