This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems

Automatic Cloud I/O Configurator for I/O Intensive Parallel Applications

Jidong Zhai, Mingliang Liu, Ye Jin, Xiaosong Ma and Wenguang Chen

Abstract—As the cloud platform becomes a promising alternative to traditional HPC (High Performance Computing) centers or in-house clusters, the I/O bottleneck problem is highlighted in this new environment, typically with top-of-the-line compute instances but sub-par communication and I/O facilities. It has been observed that changing the cloud I/O system configurations, such as choices of file systems, number of I/O servers and their placement strategies, etc., will lead to a considerable variation in the performance and cost efficiency of I/O intensive parallel applications. However, storage system configuration is tedious and error-prone to do manually, even for expert users, leading to solutions that are grossly over-provisioned (low cost inefficiency), substantially under-performing (poor performance) or, in the worst case, both.

This paper proposes ACIC, a system which automatically searches for optimized I/O system configurations from many candidates for each individual application running on a given cloud platform. ACIC takes advantage of machine learning models to perform performance/cost predictions. To tackle the high-dimensional parameter exploration space, we enable affordable, reusable, and incremental training on cloud platforms, guided by the Plackett and Burman Matrices for experiment design. Our evaluation results with four representative parallel applications indicate that ACIC consistently identifies optimal or near-optimal configurations among a large group of candidate settings. The top ACIC-recommended configuration is capable of improving the applications' performance by a factor of up to 10.5 (3.1 on average), and cost saving of up to 89% (51% on average), compared with a commonly used baseline I/O configuration. In addition, we carried out a small-scale user study for one of the test applications, which found that ACIC consistently beat the user and even the application's developer, often by a significant margin, in selecting optimized configurations.

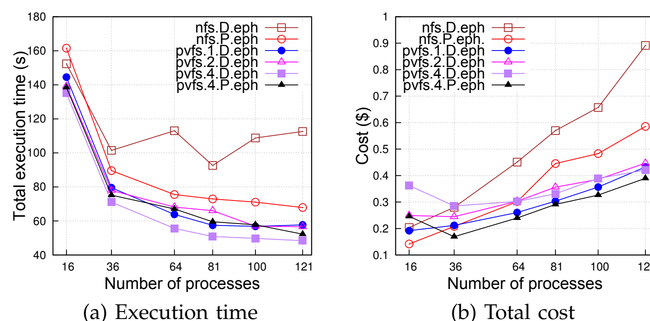
Index Terms—Performance Tool, Cloud Computing, Storage Configuration, Parallel Applications.

1 INTRODUCTION

More and more HPC users today are beginning to explore running their applications in the cloud [1], [2], [3], [4]. Emerging cloud resources targeting HPC usage, such as the Amazon CCIs (Cluster Compute Instances) [2], have largely improved the outlook for HPC in the cloud. Clouds offer many advantages over traditional HPC platforms: elastic resource allocation, elimination of queue waiting, no up-front hardware investment or hosting/maintenance/upgrades, and convenient pay-as-you-go pricing models. By closing on the performance gap between cloud instances vs. in-house clusters [4], public clouds have become a cost-effective choice to many scientific application users and developers.

Unfortunately, cloud platforms amplify the growing performance gap between the I/O subsystem and other system components long existing in conventional HPC environments [5]. Leading cloud platforms such as Amazon interconnect the compute instances with commodity networks instead of dedicated high-speed interconnection, such as InfiniBand. Also, multi-tenant cloud resources deliver inferior and sometimes highly variable performance [6].

On the flip side, clouds empower users with full, a-la-carte configuration of the I/O subsystem, which is impossible on traditional HPC clusters. For example,



(a) Execution time (b) Total cost

Fig. 1: The execution time and monetary cost of BTIO under selected I/O system configurations, in terms of file system type (nfs vs. pvfs), number of I/O servers (1, 2, 4), placement strategy (Part-time vs. Dedicated), and disk device (ephemeral).

users can choose important I/O parameters such as the file system type, the number of I/O servers, the type and number of I/O devices to use, etc. Previous study revealed that the in-cloud performance of representative HPC applications is highly sensitive to such I/O system configurations [7]. Figure 1 demonstrates this impact on both performance and monetary cost of running the NPB BTIO application (more information in Section 5), shown to vary dramatically with different I/O system configurations. It also shows that even for a single HPC application, its performance/cost behavior across different I/O configurations varies with different problem/job sizes, and no single configuration excels in all cases. The cloud enables users to setup optimized

- *J. Zhai, M. Liu, and W. Chen are with the Department of Computer Sciences and Technology, Tsinghua University, Beijing, China.*
- *Y. Jin and X. Ma are with the North Carolina State University, USA.*

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems

I/O configurations for *individual* application upon its execution, instead of forcing all applications to use a pre-configured solution.

However, taking advantage of this uniquely available configurability and deriving optimized per-application I/O configuration are very challenging and potentially very expensive. Several factors, including the lack of one-size-fits-all parameter choices, the complexity from both the system and the application side, and the obscureness of I/O system hardware/software details due to virtualization, make white-box modeling and analysis unrealistic. Meanwhile, the high-dimensional cloud I/O configuration parameter space makes learning-based, black-box approaches quite costly, in terms of both time and monetary overhead. Furthermore, as I/O configuration has been shown to be application- and even scale-dependent, knowledge and training data obtained from one application may not apply to another.

There are many tools that evaluate and configure storage systems for traditional clusters [8], [9], [10], [11] (more discussions in Section 7). However, some of them [10], [11] focus on the storage devices only and hence are not able to address the complex, high-dimensional cloud I/O configuration problem. Some others (such as Minerva [8]) are extremely complicated for non-expert users, requiring expertise with advanced tools and a large number of experiments. Moreover, none of them covers the complicated cost-performance tradeoff unique to the cloud.

To address this problem, we propose ACIC (**A**utomatic **C**loud **I**/O **C**onfigurator), the first tool to optimize the I/O system for parallel applications in the cloud. Given a parallel application to run on a given cloud platform, ACIC automatically searches for optimized I/O system configurations from many candidate settings. Our approach takes advantage of a black-box model to learn the relationship between influential I/O system configurations and the optimization objective (cost or performance). After training the model on the target cloud platform, ACIC automatically extracts the given parallel application's I/O characteristics, evaluates candidate I/O configurations, and recommends an optimized configuration according to user's selected objective.

Though learning-based performance modeling/prediction has long been explored, including for parallel applications [12], [13], ACIC's originality lies in the cost-saving mechanisms that make such approaches affordable on clouds:

1) We explore a crowdsourcing service model for automatic, per-application cloud system configuration, where community members build and share a public performance/cost database. The service may not rely on, but can benefit from continuous training data contributions, which improve its configuration accuracy, as well as its adaptivity to system upgrades. We describe our proof-of-concept ACIC tool using parallel I/O as a case study, yet the service model applies to other configurable systems.
2) Rather than case-by-case learning/prediction, we enable *reusable training* by adopting a generic synthetic I/O benchmark and systematically sampling the parameter space.
3) To tackle the large training space that renders the model training prohibitively expensive, we perform dimension reduction by evaluating parameters' impact on performance using PB matrices [14].
4) We employ two types of machine learning models in current ACIC, regression-based and ranking-based, to better identify the optimal cloud I/O configuration with limited training data.

We implemented ACIC, trained it with the synthetic yet expressive parallel I/O benchmark IOR (Interleaved Or Random) [15] on Amazon EC2, and evaluated it with four real-world data-intensive parallel applications. Our results indicate that ACIC consistently provides optimized configurations that improve performance (total execution time) by a factor of 3.1 on average and the cost saving of 51% on average under the baseline configuration (see Section 5).

We have recently released the ACIC tool, plus all our training data collected from EC2 [16]. Currently, users can download the shared training data, build the prediction model, use our provided tool to obtain I/O characteristics from their applications, run the prediction, and configure EC2 to deploy the recommended I/O configuration with our provided scripts. A preliminary version of this work has been published in SC'13 [17].

The rest of the paper is organized as follows. Section 2 introduces the exploration space of the I/O system optimization. Section 3 is the overview of the ACIC system. Section 4 illustrates the predicting and ranking model. Section 5 shows the experimental results. Section 6 describes the user study. Section 7 discusses the related work and Section 8 concludes the paper.

2 Exploration Space

2.1 I/O Configuration Options

Layer		
Application	**Application file interface** (MPI-IO vs. POSIX)	
File System	**File system internal parameters** (Stripe Size: 64KB/4MB)	
	I/O server number (1/2/4)	**I/O server placement** (Dedicated vs. Parttime)
	File system (NFS vs. PVFS2)	
Storage Device	**Software RAID** (RAID 0 vs. No RAID)	**Device number** (1/2)
	Cloud storage device type (EBS vs. Ephemeral)	

Fig. 2: Configurable I/O system stack in the cloud.

Figure 2 depicts the configurable I/O system stack in the cloud, using Amazon EC2 terms, from I/O library all the way to storage device hardware. In contrast, on traditional shared parallel platforms users typically can only configure the top layer. Therefore, in this paper we focus on the layers opened up by cloud platforms.

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems

Below we briefly describe the I/O configurations found relevant to parallel applications' performance/cost in the cloud [7].

Storage device and organization Cloud platforms typically provide multiple storage choices, with different levels of abstraction and access interfaces. E.g., with EC2 CCIs, applications have access to: *1*) the local block storage ("ephemeral") with 4×840GB capacity, where user data does not persist across instance reservations, 2) off-instance, persistent Elastic Block Store (EBS), and 3) SSD disks. Apart from data persistence, the ephemeral and EBS devices possess different performance characteristics, usage constraints, and pricing policies. Finally, a cloud HPC user can easily scale up the aggregate I/O capacity and bandwidth, e.g., by aggregating multiple disks into a software RAID 0 partition.

File system selection and configuration Typically, supercomputers or large clusters have parallel file systems such as Lustre, GPFS, and PVFS, while smaller clusters tend to choose shared file systems such as NFS. Unlike traditional HPC users, cloud users can choose between the two categories based on individual applications' demands, and switch between selections quite easily and quickly. Once selected, a parallel/shared file system itself has many internal knobs and is non-trivial to configure.

In this proof-of-concept work, we focus on two important and highly application-dependent parameters, the number and placement of I/O servers. The number of I/O servers can significantly affect the performance of I/O-intensive applications. There are two types of I/O server placement, *dedicated* and *part-time*. With the former, I/O servers run on dedicated cloud instances, while with the latter, they share cloud instances with a subset of the computing instances. Due to the obvious impact on both performance and cost, it is important to optimize such server placement for better resource utilization and cost-effectiveness.

2.2 Application I/O Characteristics

I/O workload characterization has remained an active problem [5]. Meanwhile, though applications have varying concrete I/O patterns, they also share high-level I/O behaviors common to most HPC scientific codes, especially the periodic checkpoint/restart output activities.

To enable reusable training, ACIC chooses to measure cloud I/O performance with sampled system configurations using synthetic benchmarks created via IOR [15]. IOR is a flexible and expressive parallel I/O benchmark that can be configured to mimic different applications' I/O behavior. Also, its open-source nature allows easy extension to test additional I/O features when the need arises.

Currently ACIC considers the following I/O characteristics parameters in creating IOR test cases. Note that among these parameters, 4 parameters are unique for parallel applications, such as *number of processes*, *number of I/O processes*, *collective*, and *file sharing*. The range of parameters is selected based on the real-world applications used in our evaluation, and can be expanded with additional training, without invalidating the collected data.

- **Number of processes:** total number of processes running the application in parallel
- **Number of I/O processes:** number of processes performing the I/O operations simultaneously
- **I/O interface:** POSIX, MPI-IO, or high-level libraries such as HDF5 and netCDF
- **I/O iteration count:** number of I/O iterations within the application execution
- **Data size:** amount of data each I/O process reads and writes within each I/O iteration (e.g., the size of the 3-D array partition assigned to each process)
- **Request size:** amount of data transferred in each I/O function call (I/O request size)
- **Read and/or write:** I/O operation type
- **Collective:** whether I/O processes adopt collective I/O [18] to cooperatively read/write shared files
- **File sharing:** whether the I/O processes access a single shared file, or per-process private files

Although IOR covers most important aspects of HPC I/O parameters, it does make certain simplifications. For example, the request sizes for different variables that a parallel simulation writes out may not be uniform. In our future work, we plan to assess the impact of such simplification on our model prediction accuracy and investigate ways to allow more detailed characteristics specification if necessary.

Parameter Name	Values
Disk device	{EBS, ephemeral}
File system	{NFS, PVFS2}
Instance type	{cc1.4xlarge, cc2.8xlarge}
I/O server number	{1, 2, 4}
Placement	{part-time, dedicated}
Stripe size	{64KB, 4MB}
Num. of all processes	{32, 64, 128, 256}
Num. of I/O processes	{32, 64, 128, 256}
I/O interface	{POSIX, MPI-IO}
I/O iteration count	{1, 10, 100}
Data size	{1, 4, 16, 32, 128, 512 (MB)}
Request size	{256KB, 4MB, 16MB, 128MB}
Read and/or write	{read, write}
Collective	{yes, no}
File sharing	{share, individual}

TABLE 1: Sample variables affecting performance and cost. The top 6 variables are cloud I/O system options, others workload characteristics.

To extract parameters representing application's I/O characteristics, one can use existing profiling/tracing tools [19], [20] to instrument I/O primitives of the application, followed by trace collection/analysis. We include a simple tool for collecting ACIC-relevant application I/O characteristics encompassing a tracing library and scripts for parsing and statistically summarizing I/O traces [16].

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems

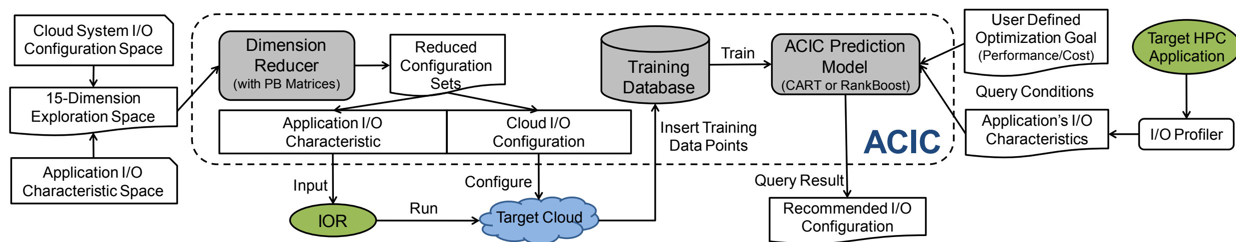


Fig. 3: ACIC architecture.

2.3 Exploration Space and Challenges

Exploration Space Table 1 summarizes the system I/O configurations and application I/O characteristics considered in this ACIC prototype. We set the range of the values according to our real-world application test cases with different job scales (32 to 256). For each parameter, we sample its value range in our training. For example, the compute-node-to-I/O-server ratio typically varies between $4:1$ and $64:1$ on a HPC cluster, which differs a lot from distributed file systems. Since there are at most 16 instances in our testbed, we select 1, 2 and 4 as sampled values of the "I/O server number" parameter. For continuous (numerical) domain parameters, such as *data size* and *request size*, we select samples from their value ranges that form evenly spaced vectors in log space. Such training is used in our study to bootstrap ACIC's auto-configuration. Again, this design allows users to constantly contribute training data points to the ACIC training database.

Challenges As shown in Table 1, although we have left out a number of parameters and sampled the numerical parameter space rather sparsely, the concatenated exploration space combining system configurations and application characteristics is still daunting. Even considering that not all sample parameter value combinations are valid (e.g., NFS does not have *stripe size*; *request size* cannot be greater than *data size*), the 15 parameter dimensions create roughly ***a million valid training data points***.[1] In Section 4 we will present how ACIC tackles this high-dimensional training space challenge.

3 Approach Overview

Figure 3 illustrates the ACIC architecture. Its central component is a black-box prediction model, which can be bootstrapped with a limited amount of initial training data. ACIC takes both the cloud system I/O configuration parameters (such as file system type, storage device type, number of I/O servers, etc.) and application I/O characteristics (such as major operation type, read/write block size, read/write count, etc.). Concatenated together, these parameters constitute a 15-D exploration space for ACIC's training and prediction. To reduce the time overhead and monetary cost associated with training, ACIC employs a dimension reducer using Plackett-Burman (PB) matrices [14], with more details discussed in Section 4.1.

1. 2 * 2 * 2 * 3 * 2 * 2 * 4 * 4 * 2 * 3 * 6 * 4 * 2 * 2 * 2 = 1,769,472.

Generally, there are several ways to collect the training data, such as application case studies, benchmarks, and trace replays. ACIC chooses the IOR [15] synthetic benchmark as it is generic, highly configurable, and open-source. It carries out the initial training by running the synthetic IOR benchmarks on the target cloud system, systematically sampling the concatenated parameter space across the dimensions selected through PB matrices. For each training run, ACIC collects the performance (cost) metric with the candidate cloud I/O configurations. With the sampled data points fed into a training database, ACIC can use different machine learning algorithms to train its prediction model. In ACIC, we employ both classification and regression trees (CART) [21] and RankBoost techniques [22].

Given a target parallel application, users can either directly provide values of relevant I/O characteristics, or use a simple profiling tool (included as part of ACIC) to extract such application-specific parameters. Both approaches are feasible, as HPC applications, especially parallel simulations, are known to have periodic, relatively well-defined I/O behavior [15]. Based on the user-specified optimization goal, currently either the performance (application execution time) or the monetary cost of execution, ACIC outputs the predicted optimal I/O configuration. Since current cloud providers normally charge users according to usage time regardless of the system load, we take the final performance and cost as our optimization objective. Note that the monetary cost of a certain application execution is not proportional to the execution time here, as I/O servers can be placed at dedicated instances or part-time ones.

One major advantage of ACIC is its reusability. It is worth pointing out that even with its dimension reducer, the initial training of ACIC may cost dozens to hundreds of hours (and dollars). However, we argue that such expense is reasonable considering that the application-independent IOR training results can be reused. Therefore, the training cost is to be amortized over many different applications and different executions of the same applications.

Another chief advantage of ACIC is its expandability. First, it benefits from continuous, incremental training. With more user-contributed IOR training data points, ACIC achieves higher prediction accuracy. This allows it to bootstrap with sparse sampling in its initial training. The additional training may even come at no extra monetary cost, as public clouds like Amazon EC2 typi-

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems

cally charge users at a hourly billing granularity. Users can fit one or more short IOR training runs into the ”residual” time allocation, after completing their application runs. Second, with continuous, incremental training, the ACIC training database can effortlessly deal with cloud hardware/software upgrades with common data aging methods. Third, ACIC can easily handle new I/O configurations or characteristic parameters by adding more dimensions into its prediction model, though the open-source IOR benchmark may need to be expanded if an application has I/O features that it does not test.

Finally, although the training and prediction are cloud-dependent, ACIC makes no assumptions on the cloud I/O configurations or application I/O characteristics and can be applied to any platform-application combinations.

4 Performance/Cost Prediction

4.1 Exploration Space Reduction

To tackle the aforementioned high-dimensional parameter space, ACIC employs a statistical technique, called *Plackett and Burman (PB) design* [14]. We choose PB design for exploration space reduction from a large number of dimensional reduction techniques due to the following reasons.

First, PB design is an important type of fractional factorial experiment designs and is good at identifying or screening important parameters (factors) when the number of factors is too large to evaluate higher-order effects [23]. Based on the previous study [24], due to “*sparsity of effects*” principle, the system performance is normally dominated by the main effects and their low-order interactions. Therefore, PB design is more suitable for analyzing the importance of various parameters on clouds. Second, PB design makes it feasible to enable a fast (though less accurate) training to bootstrap the ACIC prediction. For N parameters design, it only require N' *runs*, where N' is the smallest multiple of 4 above to N. For example, if $N = 7$, then $N' = 8$. Its computational complexity is much lower than the full factorial design ($O(N)$) vs. $O(2^N)$), such as ANOVA [25]. Third, PB design allows us to provide a crowdsourcing service model in ACIC. With the ranking order of parameters, we can explore the most influential parameters first and gradually expand our training data collection to the lower ranked dimensions for more accurate prediction.

A complete PB design includes the following steps. (1) **Building the PB design matrix:** The value of each element in the PB design matrix is either “+1” or “-1”, where “+1” denotes the parameter’s high value and “-1” denotes the parameter’s low value. The first row of the PB matrix is given in [14]. The next $N' - 2$ rows are generated through performing a circular right shift on the preceding row and the last row of the matrix is a row of minus one. (2) **Selecting the parameter’s low and high values:** The high or low value for each parameter represents a value that is the highest or the lowest of normal values for that parameter. However, the high and low values are not only restricted to numerical values. (3) **Performing the experiments using the acquired PB matrix:** For each run, the value of each parameter is set according to one row of the PB matrix. After the N' runs are completed, the performance for each run is collected. (4) **Calculating the effect for each parameter:** The effect of each parameter is calculated by multiplying the values in the PB matrix by the collected performance for each run and summing the products across all rows. The sign of the effect is meaningless and we rank the parameters according to the absolute value of the effect.

Row	*The PB Design Matrix*							Perf.
	A	**B**	**C**	**D**	**E**	**F**	**G**	
1	+1	+1	+1	-1	+1	-1	-1	19
2	-1	+1	+1	+1	-1	+1	-1	12
3	-1	-1	+1	+1	+1	-1	+1	22
4	+1	-1	-1	+1	+1	+1	-1	13
5	-1	+1	-1	-1	+1	+1	+1	50
6	+1	-1	+1	-1	-1	+1	+1	35
7	+1	+1	-1	+1	-1	-1	+1	18
8	-1	-1	-1	-1	-1	-1	-1	80
9	-1	-1	-1	+1	-1	+1	+1	10
10	+1	-1	-1	-1	+1	-1	+1	20
11	+1	+1	-1	-1	-1	+1	-1	6
12	-1	+1	+1	-1	-1	-1	+1	30
13	+1	-1	+1	+1	-1	-1	-1	17
14	-1	+1	-1	+1	+1	-1	-1	43
15	-1	-1	+1	-1	+1	+1	-1	16
16	+1	+1	+1	+1	+1	+1	+1	26
Effect	-109	-9	-63	-95	1	-81	5	
Rank	**1**	**5**	**4**	**2**	**7**	**3**	**6**	

TABLE 2: The PB design matrix with foldover ($N = 7$).

Like in the prior work by Yi et al. [24], we adopted the improved variation called *foldover PB design* [25] in ACIC. Foldover PB design is able to quantify the effects that not only single parameters have but also two-parameter interactions have on the final performance. Foldover PB design adds N' additional rows in the matrix. The signs of the values in these additional rows are the opposite of the corresponding entries in the original matrix. Table 2 shows a complete foldover PB design matrix for 7 parameters ($N = 7$). For example, the effect of the parameter A is calculated as follows: Effect$_A$ = (1×19) + (-1×12) + ... + (-1×16) + (1×26) = -109.

Parameter Name	Low Value	High Value	Rank
Data size	1MB	512MB	1
Read and/or write	read	write	2
I/O server number	1	4	3
Num. of I/O processes	32	256	4
File system	NFS	PVFS2	5
Stripe size	64KB	4MB	6
Placement	part-time	dedicated	7
Request size	256KB	128MB	8
I/O interface	POSIX	MPI-IO	9
Disk device	EBS	ephemeral	10
Collective	no	yes	11
Instance type	cc1.4xlarge	cc2.8xlarge	12
I/O iteration count	1	100	13
Num. of all processes	32	256	14
File sharing	Share	Individual	15

TABLE 3: The Plackett and Burman values and their ranking results.

In this proof-of-concept study, we built the ACIC

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems

foldover PB matrix ($N = 15$) for the 15-dimensional exploration space (in Table 1). Table 3 shows the PB values used in ACIC. We select these values based on the following rules. For non-binary (numerical) parameter value ranges, we select a slightly higher or lower value from all surveyed applications. This is because that a relatively large range for parameter values ensures that that parameter will have an effect on the output performance [24]. For a binary parameter, we use all the values. We carried out the 32 test runs with IOR on the cloud storage system configured according to the values in the PB matrix.

Table 3 shows the *relative importance ranking* determined by the PB design. The results show that the most important three parameters are "data size", "I/O operation type", and "I/O server number", while the least important ones are "whether file sharing is on", "number of all processes", and "I/O iteration count". Such ranking enables ACIC to explore the most influential parameters first and make a fast (though less accurate) training to bootstrap the ACIC prediction. This way, ACIC populates its training database by sampling the top-ranked parameters first (adopting default settings for the other parameters), then gradually expands training data collection to the lower-ranked dimensions.

4.2 Performance/Cost Prediction Models

Given the data points collected from IOR training runs guided by PB design, ACIC can then employ different black-box prediction methods. Many machine learning algorithms can help ACIC learn the mapping between I/O system/application parameters and the optimization goal.This problem falls under the general scope of *supervised learning*. In current ACIC, we employ two types of models to assess the feasibility of ACIC's reusable training. One is a regression-based model, the other is ranking-based. Users can flexibly choose one or both of them to combine their advantages. Meanwhile, ACIC is implemented in a way that different learning algorithms can be easily plugged in.

4.2.1 CART-based Prediction Model

The first technique used in ACIC is CART (Classification and Regression Trees) [21] for its simplicity, flexibility, and interpretability. It is a regression-based prediction approach, requiring no knowledge about the prediction target, with trees built top-down recursively. At each stop in the recursion, the CART algorithm determines which predictor parameter in the training data best splits the current node into leaf nodes, then continues recursively within each subtree. The optimal split minimizes the difference (e.g., root mean square) among the samples in the leaf nodes. The error for each sample is the difference between it and the average of all samples in the leaf node. Therefore, each internal node contains a "best" predictor, while each leaf node gives a predicted target result. Eventually, the optimal decision tree is pruned to avoid over-fitting. To make a prediction, the tree takes a set of parameter values as input, and outputs the predicted target value dictated by the destination leaf node as it follows the path dictated by a sequence of internal nodes.

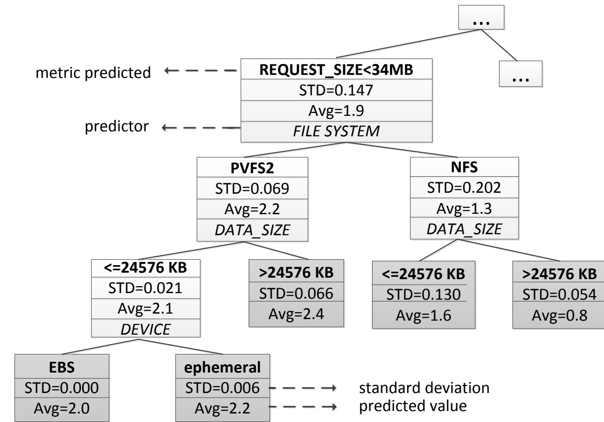


Fig. 4: Sample tree built by ACIC using CART.

Figure 4 shows a portion of the tree that models the I/O operation cost built by ACIC. The light-shaded nodes are internal nodes while the darker ones are leaves. Each level of the tree (composed by nodes with the same depth) examines the value of one dimension in the parameter space. For internal nodes, the first field contains the current-level parameter value range (such as "<=24576KB"), automatically calculated by CART to guide the decision making given the input parameter. The second field contains the standard deviation of the target value of all of its children and the third field contains the average value. The last field indicates the next-level parameter for branching its children into two subtrees. The leaf nodes report the predicted target value (both average and standard deviation).

Note that CART also arranges the ordering of parameters, by placing the ones it considers more "important" to decision making higher up (closer to the root). However, this is not redundant with the PB design generated ranking, as the former can only create such ranking based on collected training data, while the latter gives direction to training data collection itself.

With ACIC, we face the problem of performance reporting mismatch between IOR and the target application requesting I/O configuration optimization. It is unrealistic to assume that the applications can be modified to report I/O performance in a way consistent with IOR. We solve this problem by adopting *performance/cost improvement* (over a *baseline configuration*) as the predicted target rather than using absolute values. The idea is similar to the "relative" notion in storage performance modeling [11]. In our implementation and experiments, we set the baseline configuration as "single dedicated NFS server, mounting two EBS disks with a software RAID-0", which is indeed the cloud version of a highly common shared storage setup with small- to medium-scale clusters [1], [3], [4].

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems

Algorithm 1: CART-based prediction algorithm.

```
input: A CART regression tree built based on training data;
input: A parallel application and a user-specified optimization
goal (performance or cost);
output: A recommended cloud I/O configuration;
begin
    Collect I/O characteristics for the given parallel application
    A, WC^A;
    Predict the application performance or cost on a baseline
    configuration with the CART tree,
    P_{A,base} = M(WC^A, IC^{base});
    for IC^J ∈ ℂ do
        Predict the application performance or cost on the
        candidate configuration, P_{A,J} = M(WC^A, IC^J);
        Compute the relative improvement against the baseline
        configuration R_J = P_{A,base}/P_{A,J};
    end
    Sort all the relative improvement R_J;
    Output the cloud I/O configuration IC^J with the maximal
    R_J;
end
```

Algorithm 1 presents CART-based prediction algorithm in ACIC. In our cloud storage configuration context, given the target application, ACIC joins the application's I/O characteristics WC^A with all candidate I/O system configurations $\mathbb{C}$, as the input to the CART model. We skip invalid parameter combinations for performance/cost prediction. As the prediction overhead is negligible compared to the training data collection cost, a full exploration of system configuration space is affordable here. We use $P_{A,J}$ to denote the execution time or total cost of the parallel application A with an I/O configuration IC^J, where $P_{A,J} = M(WC^A, IC^J)$. M is a mapping function of the execution time, or total cost, of that I/O characteristics and cloud configurations. The candidate configurations are then sorted by their relative improvement over the baseline configuration, based on the CART prediction. ACIC can be configured to report the top k predicted optimized candidates. When $k > 1$, the application user has a better opportunity to identify an optimal or near-optimal solution, at the cost of more benchmarking runs trying out the top k configurations.

4.2.2 Ranking-based Prediction Model

To better identify the optimal target configuration with limited training data, ACIC also employs a representative Learning-to-Rank technique called RankBoost [22]. RankBoost has been widely used in information retrieval and data mining, such as web pages ranking and recommendation systems [26].

RankBoost is a ranking-based technique, extending the boosting algorithm [27]. Giving a partial order or total order for a set of input objects, it has two main advantages over regression-based methods. First, instead of predicting absolute values (such as execution time or total cost in ACIC) for input objects, it only computes a ranking list for them. RankBoost has been validated much more effectively for problems requiring a ranking order of input objects [22]. Second, pairwise-based RankBoost algorithms convert n training data points into C_n^2 pairs, thus resulting in a larger training set than regression methods ($n(n-1)/2$ vs. n), which can effectively improve prediction accuracy with limited training data.

RankBoost is also a supervised learning task and thus has training and testing phases. In training, we use χ to denote the object set to be ranked. Each object x_i of χ is a vector through concatenating application I/O characteristics WC^A and cloud I/O configurations IC^J after reduction with PB matrices in ACIC, $\chi = \{x_i | x_i = (WC^A, IC^J), i \in [1,n], A \in [1,m], J \in [1,k]\}$. Each concatenation x_i is associated with a response value y_i, which is execution time or total cost in ACIC. So, the training set can be represented as $S = \{(x_i, y_i)\}_{i=1}^n$. RankBoost will convert it into C_n^2 concatenation pairs, (x_i, x_j), where $x_i \prec x_j$. The ranking order of each concatenation pair (x_i, x_j) is defined as, if $y_i < y_j$, then $x_i \prec x_j$.

Like all boosting algorithms [27], RankBoost is an iterative algorithm. In each iteration, it searches for an optimal *weak learner* $h_t(x)$ that maximizes the ranking accuracy of concatenation pairs. A *weak learner* is a simple learning model which is defined in RankBoost [22] as the following functions.

$$h(x) = \begin{cases} 1 & \text{if } f_i(x) > \theta \\ 0 & \text{if } f_i(x) \le \theta \end{cases} \quad (1)$$

where $f_i(x)$ means the ith parameter in the concatenation vector x, θ is a threshold value. For example, in ACIC, a weak learner $h(x)$ maybe $h(x) = 1$ if *I/O server number* > 2 and $h(x) = 0$ otherwise.

To select optimal weak learners for each iteration, RankBoost uses a weight distribution D_t (the tth iteration distribution) to emphasize those concatenation pairs that are hard to be ranked. Initially, RankBoost sets equal weights ($1/C_n^2$) for each concatenation pair, (x_i, x_j). In each iteration, if a concatenation pair receives an incorrect ranking order with the selected weak learner, its weight will increase according to the formula in Algorithm 2, so that the next iteration's weak learner will focus on correcting this.

Algorithm 2: RankBoost training algorithm used in ACIC.

```
input: Training data S = {(x_i, y_i)}_{i=1}^n and iteration number T;
output: A ranking function F(x);
initialize: ∀(x_i, x_j) ∈ χ × χ, set D_1(x_i, x_j) = 1/C_n^2;
begin
    for t = 1 to T do
        Search for the optimal weak learner h_t(x) based on
        weighted distribution D_t in training data S;
        Choose step size α_t;
        // α_t is set according to the error of h_t(x) [22];
        Update distribution weights: ∀(x_i, x_j) ∈ χ × χ,
        D_{t+1}(x_i, x_j) = D_t(x_i, x_j) exp α_t(h_t(x_i) − h_t(x_j))/Z_t.
        // Z_t is a normalization factor;
    end
    Output the final ranking function: F(x) = Σ_{t=1}^T (α_t h_t(x)).
end
```

Finally, RankBoost outputs a ranking model F by

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems

linearly combining the weak rankers from each iteration, $F(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$, where $h_t(x)$ is the weak learner selected during each iteration, α_t is computed according to the training error of $h_t(x)$ [22], and T is the number of iterations. Figure 5 shows the statistics of RankBoost training for 300 iterations in ACIC. The training error of $h_t(x)$ changes little after 100 iterations. So, we set $T = 200$ in ACIC. Algorithm 2 presents the workflow of RankBoost used in ACIC.

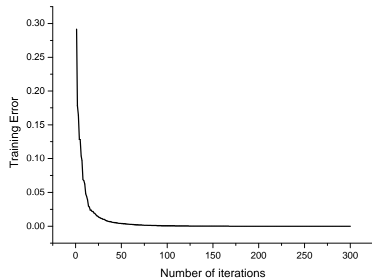


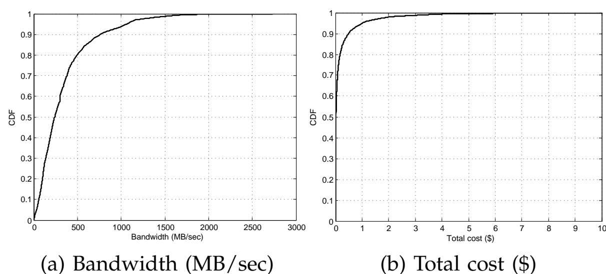
Fig. 5: Statistics of RankBoost training in ACIC.

In testing, for a given target application, we input all concatenations of the application's I/O characteristics with candidate cloud I/O configurations into $F(x)$. Then, an optimal cloud I/O configuration with the maximum value of $F(x)$ is output.

5 EVALUATION

5.1 Experiment Setup

Platform: All our experiments are performed on Amazon EC2 Cluster Computing Instances (CCIs), with node type $cc2.8xlarge$ [2]. Each such instance has two 8-core Intel Xeon processors and 60.5GB of memory. The CCIs are inter-connected with 10-Gigabit Ethernet. Regarding OS and system software, we use the Amazon Linux OS 201202, Intel compiler 11.1.072 and Intel MPI 4.0.1. The compiler optimization level is $O3$.



(a) Bandwidth (MB/sec) (b) Total cost ($)

Fig. 6: Cumulative distribution functions (CDF) of bandwidth and total cost for training samples.

Training Data: According to the PB design experimental results, the first 10 parameters listed in Table 3 are used in our training. We collect about 10K data points from EC2 platform. In order to show the distribution of training samples, we draw the cumulative distribution curves for bandwidth and total cost in Figure 6. The bandwidth of training samples generated with IOR benchmark is from 2MB/sec to about 3GB/sec. The corresponding total cost for each run is from a few cents to about $10.

Applications: It is highly time and money consuming to run I/O-intensive parallel applications to evaluate ACIC. This is not due to ACIC's own overhead, but the fact that we perform *exhaustive* evaluation of all candidate configuration settings to evaluate its optimization effectiveness. In addition, we run each experiment several times, with cache content cleared in between. Given such time/cost constraints, we select four representative applications with different I/O characteristics, from different scientific computing domains.

Name	Field	CPU	Comm.	R/W	API
BTIO	Physics	H	H	W	MPI-IO
FLASHIO	Astro	L	L	W	MPI-IO
mpiBLAST	Biology	M	M	R	POSIX
MADbench2	Cosmology	L	M	RW	MPI-IO

TABLE 4: Test applications' resource usage and I/O type (H=High, M=Medium, L=Low, R=Read, W=Write).

Table 4 shows their major I/O characteristics and computation/communication intensity levels. BTIO is an I/O-enabled version of the BT benchmark in NPB suite [28]. The problem size used in our experiment is class C, with collective I/O turned on. FLASHIO is an I/O kernel derived from the full parallel FLASH simulation, a modular adaptive mesh astrophysics code [29]. mpiBLAST [30] is a parallel implementation of the NCBI BLAST tool, for protein or DNA sequence search. In our tests, the 84GB *wgs* database is partitioned into 32 segments and there are around $1K$ query sequences sampled from itself. MADBench2 is a "stripped-down" version of the MADspec code, used in analyzing the Cosmic Microwave Background (CMB) radiation datasets [31]. In our experiments, the output file is up to 32GB, accessed four times throughout the execution.

Application	NP	Device	P/D	FS	IOS	SS
Optimal Performance Configurations						
BTIO	64	EBS	P	NFS	1	NA
	256	eph.	P	PVFS2	4	4MB
FLASHIO	64	eph.	D	NFS	1	NA
	256	eph.	P	NFS	1	NA
mpiBLAST	32	eph.	P	PVFS2	4	64KB
	64	eph.	D	PVFS2	4	4MB
	128	eph.	D	PVFS2	4	4MB
MADbench2	64	eph.	D	PVFS2	4	4MB
	256	EBS	D	PVFS2	4	4MB
Optimal Cost Configurations						
BTIO	64	EBS	P	NFS	1	NA
	256	eph.	P	PVFS2	4	4MB
FLASHIO	64	EBS	P	NFS	1	NA
	256	eph.	P	NFS	1	NA
mpiBLAST	32	eph.	P	PVFS2	4	64KB
	64	eph.	P	PVFS2	4	4MB
	128	eph.	P	PVFS2	4	4MB
MADbench2	64	EBS	P	PVFS2	4	4MB
	256	eph.	P	PVFS2	4	64KB

TABLE 5: Optimal performance and cost configurations for different applications with different scales. Column names: **NP** - Number of I/O processes; **Device** - Disk device; **P/D** - I/O server placement, part-time (P) or dedicated (D); **FS** - File system; **IOS** - Number of I/O servers; **SS** - Stripe size for PVFS2; eph. - ephemeral.

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems

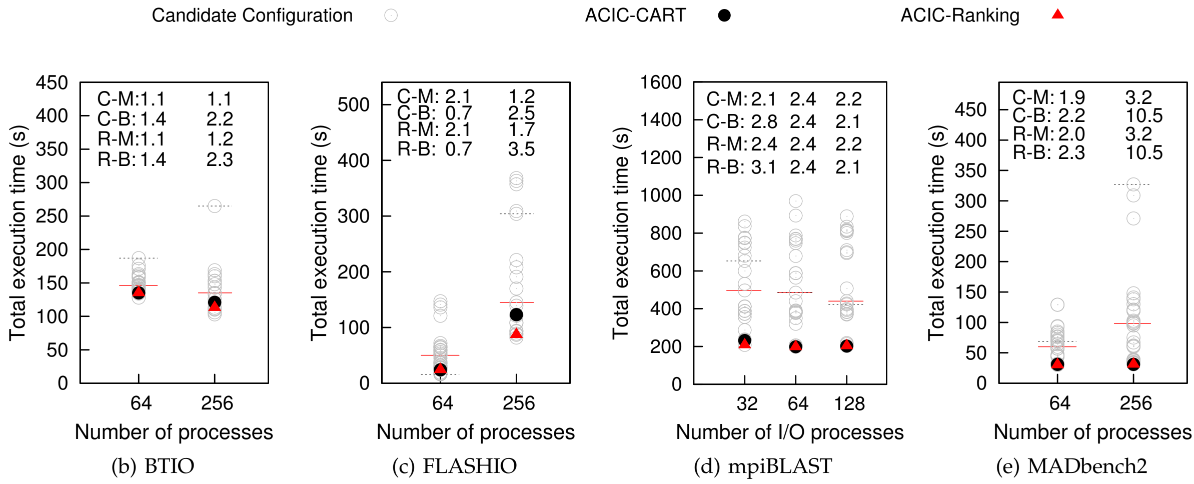


Fig. 7: Total execution time of test applications. In each set of application run, the black dot (the CART-based model) and the red dot (the Ranking-based model) indicate the ACIC predicted best configuration's performance and the gray circles indicate performance of all candidate configurations. The solid (red) line marks the median (M) performance among all configuration candidates, while the dashed (black) line marks the performance of the baseline (B) I/O configuration. Speedup ratios achieved by ACIC over the median and baseline performance are shown at the top of each figure (C means CART, R means Ranking, M means median, B means baseline).

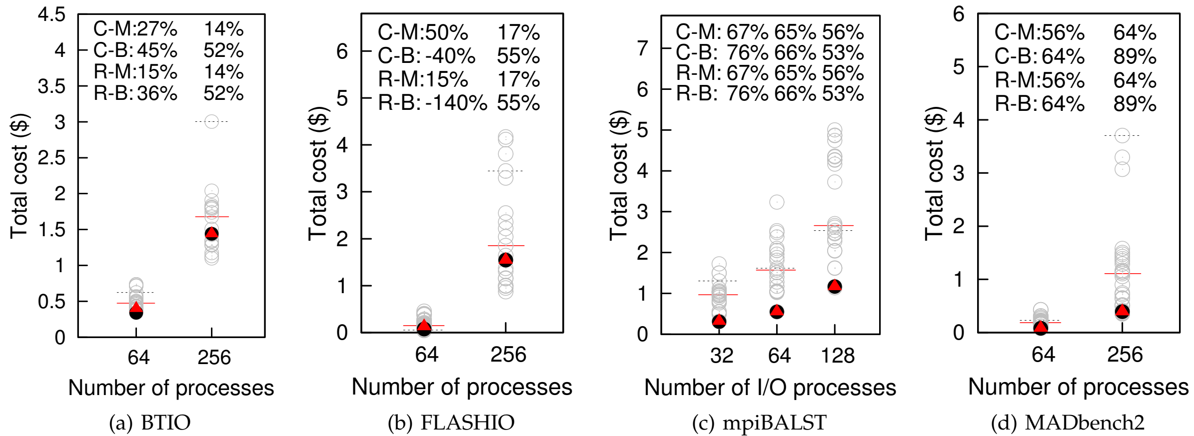


Fig. 8: Total cost of running the test applications. Cost saving percentages are listed at the top of each figure.

5.2 Optimal I/O Configurations

To evaluate ACIC, we need to actually measure the above applications' performance (depicted with total execution time) and monetary cost running on EC2, using each of the candidate I/O configurations. Table 5 shows the optimal I/O configurations we found, with performance (overall execution time) and total cost as the optimization goal. The results for performance optimization showcase the lack of one-size-fits-all I/O configurations, with 7 unique optimal I/O configurations for 9 application runs. This means that even for the same application, different job sizes (numbers of processes) will call for different I/O system settings. Taking mpiBLAST as an example, the optimal performance configuration for 32-process runs adopts part-time I/O servers, while the one for 128-process runs adopts dedicated. One possible reason is that with a smaller number of processes, the locality effect brought by the part-time I/O servers outweighs other I/O system options. This is less likely to happen on today's in-house clusters, whose interconnect often use dedicated high performance network like InfiniBand. Even with a moderate 5-D configuration space, it is hard for users to manually explore the impact of parameter values and their interplay, as demonstrated in our user study (Section 6). The configuration results for cost optimization show similar behavior and in many cases the best configuration for cost does not agree with that for performance optimization.

5.3 ACIC Auto-Configuration Effectiveness

Figure 7 and Figure 8 show the execution time and cost distribution respectively, for the evaluated 9 application executions. The monetary cost for each execution is:

$$cost = execution_time \times num_instances \times unit_price \quad (2)$$

As mentioned earlier, we exhaustively tested *all* candidate configurations, each indicated by a gray circle, whose vertical span depicts the range of perfor-

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems

mance/cost measurement for the entire configuration space. The lowest dot in each figure is the measured optimal configuration. The black and the red points highlight the target measurement achieved under the ACIC recommended I/O configuration with the CART-based model and the Ranking-based model. Note that some gray circles may be covered by the black point. For example, in Figure 7b, some gray circles are covered by the black point, so the solid (red) line does not seem to be at the median. When ACIC gives several configurations as co-champions, we report the median results using these configurations. For each application setting, the solid (red) line indicates the median performing I/O configuration's position among the gray circles and the dashed (black) line marks the performance of the baseline I/O configuration. As described Section 4.2.1, the baseline we used is "dedicated NFS server mounting two EBS disks with a software RAID-0", a configuration similar to the baseline setup of many small- to medium-sized in-house clusters.

First, these figures clearly demonstrate the potentially large difference, caused by different I/O system configurations, in *overall execution time* (not total I/O time) and monetary cost of running data-intensive applications in the cloud. More specifically, we see the performance difference ranging between $1.4\times$ and $10.5\times$, and cost difference between $2.2\times$ and $10.5\times$. Second, at a glimpse, ACIC is able to identify near-optimal I/O configurations in almost all situations, as the black and red points are located near the bottom of the gray "spectrum". At the top of each chart, we note the improvement achieved by the ACIC-recommended configuration over the median ("M"/solid line) and the baseline configuration ("B"/dashed line). For performance, we used *speedup*, calculated as:

$$speed_up = \frac{time_{baseline/median}}{time_{ACIC}}. \quad (3)$$

For cost, we report

$$cost_saving = \frac{cost_{baseline/median} - cost_{ACIC}}{cost_{baseline/median}} \times 100\% \quad (4)$$

In all cases, the ACIC-recommended configuration outperforms the median configuration, by a factor of 1.1-3.2 for both the CART-based model and the Ranking-based model in execution time, while delivering a cost saving of 14%-67%. It also beats the baseline configuration most of the time. There is an exception of FLASHIO using 64 processes, where the baseline configuration happens to be near-optimal itself. Moreover, the absolute values of execution time (and hence cost) are relatively small, leading to a substantial negative cost saving in this case.

Figure 9 presents the performance comparison for the CART-based prediction model and the Ranking-based model over the median (M) and baseline (B) performance. The Ranking-based model presents better performance than the CART-based for most of the programs.

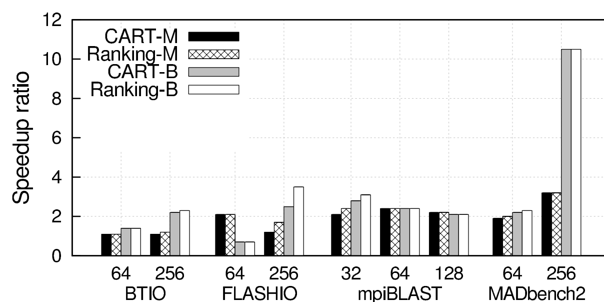
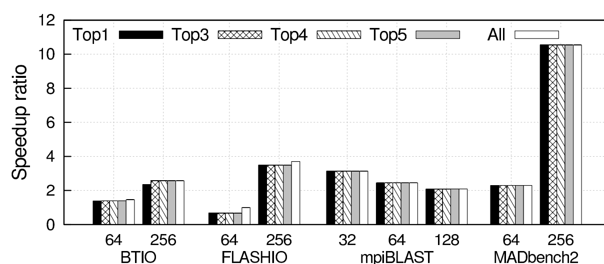


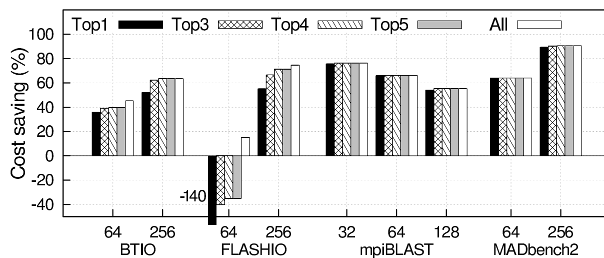
Fig. 9: Performance comparison for the CART-based model and the Ranking-based model (M means over the median performance, B means over the baseline performance).

The Ranking-based model is almost able to identify near-optimal I/O configurations in all situations as well as with smaller co-champions. For instance, the CART-based method fails to identify optimal configurations for FLASHIO and mpiBLAST. In the prediction of the total cost, the Ranking-based model presents consistent results with the CART-based.

Since the Ranking-based model converts a limited training data points into a much larger training set than the CART-based model ($n(n-1)/2$ vs. n), the Ranking-based model usually can obtain much higher prediction accuracy with limited training data. In general, the Ranking-based model is more suitable for problems requiring a ranking order for input objects. However, the CART-based model can give a prediction of execution time or total cost for a given application-platform combination. The CART tree acquired with the training set can also help users to understand the relative importance of input parameters. Users can flexibly choose one model or both of them to combine their advantages in ACIC.



(a) Execution time (over baseline)



(b) Total cost (over baseline)

Fig. 10: Accuracy enhancement from examining top-k ACIC recommendations using the Ranking-based method.

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems

Next, we examine the potential difference made by verifying a larger ACIC recommendation set, an optional effort users can make by running their applications with not one, but the top-k recommendations. As mentioned earlier, users may have "residual resource" left from their hourly cloud instance rentals and can piggy-back verification runs at no extra cost. Figure 10 shows the execution time and cost improvement achieved by the best configuration among the top 1, 3, 4, and 5 recommendations and eventually all I/O configurations (the true optimal) using the Ranking-based model (Due to space limit, we omit the results for the CART-based model which shows similar behavior). The results reveal that actually the top recommendation (median if there are co-champions) works fairly well, though considering more top candidates does help with several cases (eg. 64-process FLASHIO). In particular, in almost all cases, little further gain can be achieved by checking beyond the top 3 recommendations.

5.4 Training Cost Analysis

The ACIC overhead includes three types of cost, caused by its profiling, training data collection, the actual prediction. Among them, the most significant item is definitely the training data collection through IOR runs on the cloud, which incurs time overhead larger than the other two by orders of magnitude, and could be expensive money wise. More training data points, however, typically lead to higher prediction accuracy. To investigate this tradeoff, we experimented with CART-based prediction using different numbers of configuration parameters (dimensions), as guided by PB design results.

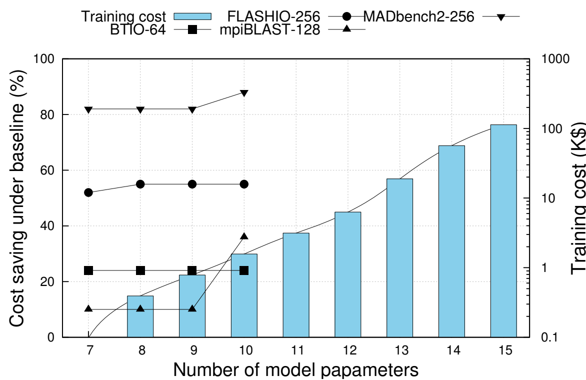


Fig. 11: Impact on prediction performance using different numbers of top ranking model parameters.

Figure 11 presents the results of this sensitivity study using four sample runs, one for each application. The x axis indicates the number of top ranking parameters used in model training as ordered by PB matrices. For each parameter count, the y axis on the left measures the performance of the ACIC top recommendation in terms of cost saving under the baseline, while the y axis on the right measures the cost of training data collection. Note that the left axis is linear scale and the right is log scale. When using 10 parameters, the total training data collection cost is around $1K.

The results here show that we can still achieve considerable cloud application execution cost saving, with only the top 7 parameters (which requires a training data collection cost of only $108). Meanwhile, we do observe higher optimization effectiveness when considering more parameters (by collecting more training data points), though the gain appears to be heavily application-dependent. As expected, the estimated training data collection cost continues to grow exponentially beyond 10 parameters, reaching $100K when exploring the full 15-D space. Due to time/funding constraints, we did not perform more training than the top 10 dimensions, and do not expect such additional exploration will bring significant gain, as shown in Figure 10.

5.5 Comparison with PB Space Walking

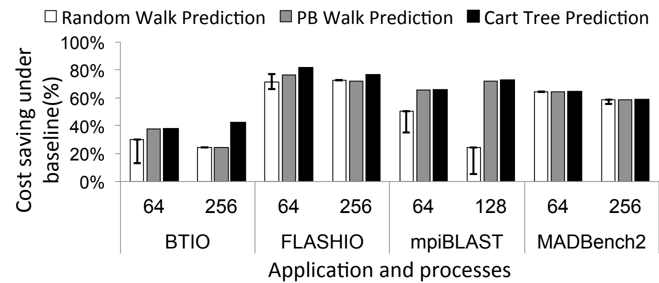


Fig. 12: Comparing alternative prediction approaches.

Finally, we compare the auto-configuration capability of the CART-based and the PB-guided space walking prediction [17], again in terms of cost saving over the baseline configuration. Here we compare three prediction methodologies. The first is *random walk*, which randomly selects the ordering of the I/O configuration parameters in its dimension-by-dimension training and prediction. For this approach, we report the average results from 10 predictions with different random parameter orderings, with the y error bars depicting the range of cost saving distribution. The second is the PB-guided walk method [17]. The third is the CART-based prediction.

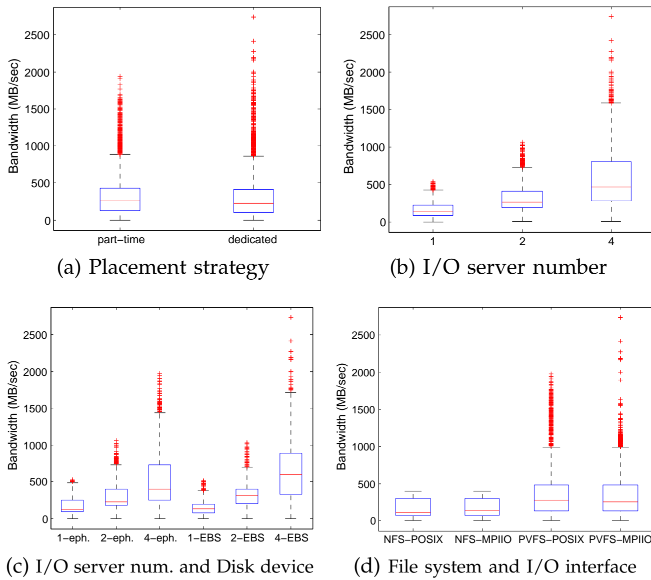
Figure 12 shows the CART-based prediction delivers the best optimization results consistently. The PB-guided space walking closely follows in most cases, benefiting from the guidance of PB designs and application-specific training. The random walking approach, on the other hand, generates significantly inferior as well as less predictable optimization performance in half of the cases. The results confirm that PB-guided walking is an appealing approach when the ACIC training database has not been sufficiently populated.

5.6 Observations From Training Experience

In addition to releasing our ACIC tool, here we share the major observations based on our extensive initial

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems



(a) Placement strategy

(b) I/O server number

(c) I/O server num. and Disk device

(d) File system and I/O interface

Fig. 13: Observations from training data.

training with roughly 10K data points from EC2. The distribution for different training data is shown with boxplots in Figure 13.

1) It is more cost-effective to use part-time than dedicated I/O servers for applications with I/O aggregators, where each communication group has a root process that collects data and writes them locally. In particular, data locality can be much enhanced when placing the part-time I/O servers on the same physical instances as the aggregators. However, as shown in Figure 13a, there is no significant difference between these two placement strategies.
2) As shown in Figure 13b, for parallel file system like PVFS2, having more I/O servers can improve performance of time perspective significantly. Across all four applications, we found few cases where one PVFS2 I/O server performs better than four ones. In Figure 13c, EBS disks usually perform better than ephemeral disks when there are more than one I/O servers deployed.
3) As shown in Figure 13d, PVFS works better than NFS in most cases, but NFS works better for applications performing small amounts of I/O using POSIX API on EC2 platform.
4) It is important to tolerate server connection failures on a cloud platform for production runs. We experienced lost connections to the I/O server, causing data corruption, in around 1‰ of experiments during training.

6 User Study

To further verify ACIC's benefit on automated I/O configuration optimization, we performed a small-scale subject study. We used one of our test applications, mpiBLAST, as we obtained consent from one of its core developers [30] ("Dev"), plus one of its skilled users [32] ("User"), to participate in our evaluation. It is challenging to do a larger study due to the difficulty in finding (expert) users/developers of I/O-intensive parallel applications, simultaneously with cloud execution experience and time to participate. We provided the participants with sufficient information regarding the executions (such as input and job scale) and the platform (such as pricing policy and device performance). Based on their knowledge and experience, the participants each gave the optimal configurations manually selected. E.g., the user gave a configuration of "Eph.-P-NFS-1-4MB" for cost minimization of 32-process runs, while the developer gave a configuration of "Eph.-D-PVFS2-2-4MB" for performance optimization of 64-process runs.

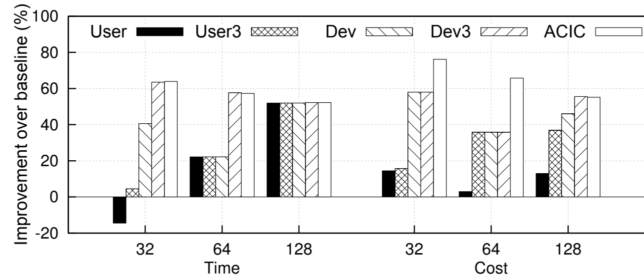


Fig. 14: Comparing manual configurations with ACIC.

Figure 14 shows the improvement of ACIC's predicted configuration and the manually selected ones. Across all execution scales and both optimization goals, ACIC consistently provides better suggestion than the experienced human participants, beating the user by an average of 37.43% and the developer by 17.8%. In addition, both developer and user agree with each other in three out of the six test groups, confirming the impact of common knowledge. However, in two of the rest three test groups, their selections generate highly contrasting results, indicating the limitation and unreliability of manual configurations. We also invited them to give 3 configurations for each test group provided with the insights in Section 5.6, and then compared the ACIC with the top-3 manual configurations (denoted as "Dev3" and "User3"). While the execution time of the top-3 manual configurations by the developer can match the ACIC performance, the manual top-3 configurations visibly lag behind ACIC (36% for user and 17% for developer on average).

7 Related Work

Parameter space reduction: There are many dimensional reduction techniques, such as Principal Component Analysis (PCA) [33] and factor analysis [34]. PCA reduces the data dimension through finding a few orthogonal linear combinations of the original variables. Factor analysis investigates variable relationships by collapsing a large number of variables into a few interpretable underlying factors. However, the above techniques need a large number of input data to perform dimension reduction. In contrast, PB design [14] can

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems

identify the importance of a number of independent parameters with a limited number of experiments. It has been applied to computer system's analysis. For example, Yi et al. [24] employed it to identify key processor parameters for massive simulations. The novelty in this paper, however, lies in the combination of PB-based space reduction with multiple machine learning approaches to enable cost-effective, reusable model training for black-box performance/cost prediction.

Machine learning algorithms: Learning-based performance analysis has long been explored [35], [36]. For example, Joshi [35] employed machine learning techniques to search the application behavior space to automatically construct benchmarks for evaluating a computer architecture design. Hassan et al. [36] applied both regularized regression and random forests in mining bioprocess data and also compared them with multiple linear regression. Actually, CART models have also been used as attribute filters to prune the similarity search space [37]. However, ACIC's originality lies in the cost-saving mechanisms that make such approaches affordable on clouds.

Cloud system configuration: Recently several approaches have been developed to optimize cloud platform configurations [1], [38], [39]. For instance, Gideon et al. [1] study the impact of different data sharing options for scientific workflows on Amazon EC2. Herodotou et al. [38] propose a system, *Elastisizer*, which can select the proper cluster size and instance types for MapReduce workloads running in the cloud. DOT [39] is a model analyzing large data analytic software and offering optimization guidelines. Most of these efforts assume certain knowledge on the application/middleware internals, while ACIC is based on black-box prediction and can assist many applications with diverse I/O behaviors. Also, ACIC offers the flexibility and expandability that allow it to work across cloud platforms and across hardware updates. The recently proposed *Scalia* [40] is a cloud storage brokerage solution that focuses on cross-cloud optimization and estimates cost using longer-term access statistics. In contrast, ACIC, while capable of multi-cloud optimization, is designed specifically to address the high-dimensional space optimization problem for individual application.

Storage provisioning tools: There are tools aiming at reducing the human effects involved in storage system provisioning and management. For instance, MINERVA [8] and Hippodrome [9] perform automatic block-level cluster storage tuning. Wang et al. [41] used CART models to predict storage device performance. *scc* [42] automates cluster storage configuration based on formal specifications of application behavior and hardware properties. Our work complements such prior work by addressing the unique storage system configuration space opened up by cloud and the training cost challenge brought by the high-dimensional configuration space.

Prediction models: Many studies exist on performance modeling for I/O systems [43], [44]. For instance, Nikolaus et al. [43] demonstrated that the Palladio Component Model can predict the performance of industry workload with system using virtual storage. Eno Thereska et al. [44] proposed a robust performance model, called IRONModel, to localize system-model inconsistencies. Osogami et al. [45] optimized web system performance by heuristically searching the configuration space to automatically predict the performance based on the model measured similar configurations. In addition, *Pesto* [10] is a unified storage performance management system that automatically constructs approximate black-box performance models of storage devices. Compared to these studies, our work focuses on the unique high-dimensional black-box modeling of cloud performance and the associated training cost challenge.

8 CONCLUSIONS

In this paper, we demonstrate that cloud I/O system configurations have considerable impacts on both the performance and cost efficiency of I/O intensive parallel applications. We further propose ACIC, an automatic cloud I/O system configuration tool for parallel applications. ACIC combines several statistical and machine learning techniques to enable application-dependent, incremental model training and black-box performance/cost prediction. In particular, we have found that the PB design approach, which effectively trims the parameter exploration space and reduces the high-dimensional model training to a feasible task, works well in conjunction with regression tree, learning-to-rank, and space walking. Our evaluation results demonstrate that accurate I/O configuration can be predicted with a significantly reduced exploration dimension, without requesting users to perform application-specific manual tuning or benchmarking.

REFERENCES

[1] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling, "Data Sharing Options for Scientific Workflows on Amazon EC2," in *SC*, 2010.

[2] Amazon Inc., "High Performance Computing (HPC)," http://aws.amazon.com/ec2/hpc-applications/, 2011.

[3] C. Evangelinos and C. Hill, "Cloud Computing for Parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2." *Ratio*, vol. 2, no. 2.40, pp. 2–34, 2008.

[4] Y. Zhai, M. Liu, J. Zhai, X. Ma, and W. Chen, "Cloud Versus Inhouse Cluster: Evaluating Amazon Cluster Compute Instances for Running MPI Applications," in *SC*, 2011.

[5] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O Performance Challenges at Leadership Scale," in *SC*, 2009.

[6] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson *et al.*, "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[7] M. Liu, J. Zhai, Y. Zhai, X. Ma, and W. Chen, "One Optimized I/O Configuration per HPC Application: Leveraging The Configurability of Cloud," in *APSys*, 2011.

[8] G. Alvarez, E. Borowsky, S. Go *et al.*, "Minerva: An Automated Resource Provisioning Tool for Large-scale Storage Systems," *TOCS*, vol. 19, no. 4, pp. 483–518, 2001.

[9] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch, "Hippodrome: Running Circles Around Storage Administration," in *FAST*, 2002.

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2378277, IEEE Transactions on Parallel and Distributed Systems

[10] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal, "Pesto: Online Storage Performance Management in Virtualized Datacenters," in *SOCC*, 2011.
[11] M. Mesnier, M. Wachs, R. Sambasivan, A. Zheng, and G. Ganger, "Modeling the Relative Fitness of Storage," in *SIGMETRICS*, 2007.
[12] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee, "An Approach to Performance Prediction for Parallel Applications," in *Euro-Par*, 2005.
[13] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of Inference and Learning for Performance Modeling of Parallel Applications," in *PPoPP*, 2007.
[14] R. Plackett and J. Burman, "The Design of Optimum Multifactorial Experiments," *Biometrika*, vol. 33, no. 4, pp. 305–325, 1946.
[15] H. Shan, K. Antypas, and J. Shalf, "Characterizing and Predicting the I/O Performance of HPC Applications Using a Parameterized Synthetic Benchmark," in *SC*, 2008.
[16] M. Liu, Y. Jin, J. Zhai, Y. Z. Q. Shi, X. Ma, and W. Chen, "ACIC Homepage," http://hpc.cs.tsinghua.edu.cn/ACIC, 2013.
[17] M. Liu, Y. Jin, J. Zhai, Y. Zhai, Q. Shi *et al.*, "ACIC: automatic cloud I/O configurator for HPC applications," in *SC*, 2013.
[18] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *FRONTIERS*, 1999.
[19] A. Konwinski, J. Bent, J. Nunez, and M. Quist, "Towards An I/O Tracing Framework Taxonomy," in *PDSW*, 2007.
[20] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel I/O Prefetching Using MPI File Caching and I/O Signatures," in *SC*, 2008.
[21] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.
[22] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer, "An Efficient Boosting Algorithm for Combining Preferences," *J. Mach. Learn. Res.*, vol. 4, 2003.
[23] T. W. Simpson, J. Peplinski, P. N. Koch, and J. K. Allen, "On the Use of Statistics in Design and the Implications for Deterministic Computer Experiments," *Design Theory and Methodology*, pp. 14–17, 1997.
[24] J. Yi, D. Lilja, and D. Hawkins, "A Statistically Rigorous Approach for Improving Simulation Methodology," in *HPCA*, 2003.
[25] D. Montgomery, *Design and analysis of experiments*. John Wiley & Sons Inc, 1991.
[26] T.-Y. Liu, "Learning to Rank for Information Retrieval," *Found. Trends Inf. Retr.*, vol. 3, no. 3, pp. 225–331, 2009.
[27] Y. Freund and R. E. Schapire, "A Decision-theoretic Generalization of On-line Learning and an Application to Boosting," *J. Comput. Syst. Sci.*, vol. 55, Aug. 1997.
[28] P. Wong and R. der Wijngaart, "NAS Parallel Benchmarks I/O Version 2.4," *NASA Ames Research Center Tech. Rep.*, 2003.
[29] M. Zingale, "FLASH I/O Benchmark Routine Parallel HDF5," http://www.ucolick.org/~zingale, 2001.
[30] H. Lin, X. Ma, W. Feng, and N. Samatova, "Coordinating Computation and I/O in Massively Parallel Sequence Search," *TPDS*, vol. 22, no. 4, pp. 529–543, 2011.
[31] Computational Research Division, "Madbench2," http://crd-legacy.lbl.gov/~borrill/MADbench2/.
[32] R. Xue, W. Chen, and W. Zheng, "CprFS: A User-level File System to Support Consistent File States for Checkpoint and Restart," in *ICS*, 2008.
[33] I. Jolliffe, *Principal Component Analysis*. Springer Verlag, 1986.
[34] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer New York Inc., 2001.
[35] A. M. Joshi, *Constructing Adaptable and Scalable Synthetic Benchmarks for Microprocessor Performance Evaluation*. ProQuest, 2007.
[36] S. Hassan, M. Farhan, R. Mangayil, H. Huttunen, and T. Aho, "Bioprocess Data Mining Using Regularized Regression and Random Forests," *BMC Systems Biology*, vol. 7, no. S-1, p. S5, 2013.
[37] E. Thereska, B. Doebel, A. Zheng, and P. Nobel, "Practical Performance Models for Complex, Popular Applications," in *SIGMETRICS*, 2010.
[38] H. Herodotou, F. Dong, and S. Babu, "No One (cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics," in *SOCC*, 2011.
[39] Y. Huai, R. Lee, S. Zhang, C. H. Xia, and X. Zhang, "DOT: A Matrix Model for Analyzing, Optimizing And Deploying Software for Big Data Analytics in Distributed Systems," in *SOCC*, 2011.
[40] T. Papaioannou, N. Bonvin, and K. Aberer, "Scalia: An Adaptive Scheme for Efficient Multi-Cloud Storage," in *SC*, 2012.
[41] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. Ganger, "Storage Device Performance Prediction with CART Models," in *MASCOTS*, 2004.
[42] H. Madhyastha, J. McCullough, G. Porter, R. Kapoor, S. Savage *et al.*, "SCC: Cluster Storage Provisioning Informed by Application Characteristics and SLAs," in *FAST*, 2012.
[43] N. Huber, S. Becker, C. Rathfelder, J. Schweflinghaus, and R. H. Reussner, "Performance Modeling in Industry: A Case Study on Storage Virtualization," in *ICSE*, 2010.
[44] E. Thereska and G. R. Ganger, "Ironmodel: Robust Performance Models in the Wild," in *SIGMETRICS*, 2008.
[45] T. Osogami and S. Kato, "Optimizing System Configurations Quickly by Guessing at The Performance," in *SIGMETRICS*, 2007.

Jidong Zhai received the BS degree in computer science from University of Electronic Science and Technology of China in 2003, and PhD degree in computer science from Tsinghua University in 2010. He is now an assistant professor in Department of Computer Science and Technology, Tsinghua University. His research interests include high performance computing, compilers and performance analysis and optimization of parallel applications.

Mingliang Liu received the BS degree in information management from China Three Gorges University in 2006. He has been a PhD student in computer science of Tsinghua University since 2007. He is now an research associate at Qatar Computing Research Institute. His research interests include high performance computing, compilers and storage systems.

Ye Jin received the BS degree in computer science from Beijing Normal University in 2007. He has been a PhD student in computer science department of North Carolina State University since 2010. He is now an research associate at Qatar Computing Research Institute. His research interests include high performance computing, data management and data compression.

Xiaosong Ma received the BS degree in computer science from Peking University, China, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 2003. She is currently an associate professor in the Department of Computer Science at North Carolina State University. She is also a joint faculty in the Computer Science and Mathematics Division at the Oak Ridge National Laboratory. Her research interests include the areas of storage systems, parallel I/O, high-performance parallel applications, and self-configurable performance optimization. She received the US Department of Energy (DOE) Early Career Principal Investigator Award in 2005, the US National Science Foundation (NSF) CAREER Award in 2006, and the IBM Faculty Award in 2009.

Wenguang Chen received the BS and PhD degrees in computer science from Tsinghua University in 1995 and 2000 respectively. He was the CTO of Opportunity International Inc. from 2000-2002. Since January 2003, he joined Tsinghua University. He is now a professor and associate head in Department of Computer Science and Technology, Tsinghua University. His research interest is in parallel and distributed computing and programming model.

1045-9219 (c) 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.