RECOMMENDING OPTIMAL CLOUD CONFIGURATION BASED ON BENCHMARKING IN BLACK-BOX CLOUDS

Gueyoung Jung, Naveen Sharma, Tridib Mukherjee, Frank Goetz, and Julien Bourdaillet
Xerox Research Center
{gueyoung.jung, naveen.sharma, tridib.mukherjee, frank.goetz, julien.bourdaillet}@xerox.com

Abstract
This paper focuses on recommending optimal cloud configuration for deploying complex user workloads. Such recommendation has become imperative in recent times for cloud users to reduce complexity in selecting a configuration. The number of different configuration options has increased many-fold due to the proliferation of cloud providers and different non-standardized offerings from these providers. Furthermore, the performance and price implications are unknown beforehand, when a cloud user deploys her workload into these different configurations. The problem gets exacerbated since cloud providers generally keep the underlying infrastructures and technologies non-transparent to users. In this paper, we present (i) a benchmark-based modeling approach to accurately estimate the performance of workloads on target black-box cloud configurations; and (ii) a search algorithm that first generates a capability vector consisting of relative performance scores of resource types (e.g., CPU, memory, and disk) for each configuration and then, identifies a near optimal cloud configuration based on the capability vectors. Experiments show that our approach accurately estimates the performance capability, and performs efficient search for the near optimal cloud configuration that has the minimum price while meeting the throughput goal.

Keywords: cloud, benchmarking, estimation, recommendation

1. INTRODUCTION

Recently, identifying optimal cloud configuration from various options has become critical but difficult problem to cloud users. We have seen a proliferation of cloud providers with increasing popularity of cost-effective deployment of complex applications such as multi-tier web transactions and parallel data mining on the cloud. Indeed, there are more than hundred cloud providers in the current market1. Further, each provider offers many different virtual machines (VMs). These VMs typically vary in terms of their types (e.g., small, medium, large, and extra large) and prices, which can be determined by resource capacities (e.g., the number of vCPUs, disk space, and memory size). Such VMs can have different performance among cloud providers, even if they offer the same VM type, since these VMs can be configured with different customized software (e.g., guest operating system), and deployed on different underlying virtualization technologies and infrastructures. Therefore, cloud users typically face with numerous options on different VMs and their combinations to build a cloud configuration for their workloads.

It is non-trivial for cloud users to identify the best cloud configuration (i.e., a best VM or a combination of VMs to achieve optimal performance and cost for a given workload). A cloud user can be overwhelmed by a number of cloud configurations, when exploring clouds for her workload deployment. Moreover, the performance (e.g. throughput) and cost implications of choosing a cloud configuration for the workload is typically unknown to cloud users. The problem gets exacerbated since clouds are typically black-boxes to cloud users. Cloud providers generally keep the underlying infrastructure and technology details (e.g., server, cluster, storage, and network structures and how VMs are managed) non-transparent (Voorsluys, 2011), but mainly open up the list of pre-defined VM types along with their corresponding prices. Additionally, the cloud providers continually integrate new hardware and software artifacts into clouds.

A user may provision a large number of high-end VMs to avoid the risk of not meeting her performance objectives. This may lead to over-provisioning and unnecessarily high cost. Meanwhile, a cost-concerned user may want low-end VMs to save cost. This however leads to under-provisioning and undesirably low performance. The user may further find an optimal cloud configuration by blindly exploring different cloud configurations and evaluating them to see whether they meet her throughput goal. However, this trial-and-test approach will be very expensive and time-consuming, since, as mentioned earlier, she will find numerous different cloud configuration options. Different workloads and configurations have different performance characteristics (Jayasinghe, 2011). Moreover, the deployment of a workload for testing is typically very complicated (Jayasinghe, 2012). Although a user can figure out which VM has the fastest CPU, disk I/O, or memory separately using existing online services 2, this is not sufficient for the user to understand the performance.

---

1 Refer to CloudHarmony at http://cloudharmony.com


doi: 10.29268/stcc.2013.1.2.2
implications of the VM capabilities on her complex workloads. This is because resource types are usually interdependent while dealing with workloads. This is also because a resource can be bottlenecked for certain amount of loads, and the bottleneck can be migrated between resources as load changes (Malkowski, 2009).

1.1 CONTRIBUTIONS

This paper describes an approach to efficiently estimate performance capabilities of black-box clouds for complex workloads, and to identify and recommend the near optimal cloud configuration that minimizes the price, while meeting user throughput goal. Specifically, our contributions are as follows.

- Estimating cloud capability requirement for workload and its throughput goal. We develop an approach to characterize the performance of a given complex workload and then, build the performance model for estimating the required capability of each resource type to achieve the throughput goal of the workload. These performance capabilities are then encoded into a capability vector.

- Estimating performance capabilities of target clouds. We develop benchmark-based performance scoring methodology to support the accurate estimation of a target cloud’s capability for a workload via the performance model. The benchmark suite is much simpler to be deployed than the application itself. Thus, the performance modeling can be done in an efficient manner. These performance scores are encoded into relative performance capability vectors, each of which represents the capability (i.e., the maximum throughput) of a building block (i.e., a VM) of a target cloud configuration.

- Efficiently identifying near optimal cloud configuration. Using capability vectors, we cast the search problem into the Knapsack problem (Kellerer, 2004). We develop a heuristic search algorithm to identify a near optimal cloud configuration having a best price. Our approach can reduce the search space and speed up the search procedure.

We have evaluated our approach in public clouds with intensive web transaction workloads. Experiment results show that our approach accurately estimate performance capabilities of cloud configurations. To evaluate the scalability of our search algorithm, we perform an extensive simulation with capability vectors collected from benchmark results.

The remainder of this paper is structured as follows. Section 2 outlines our recommendation procedure. Section 3 and 4 describe the estimation and the search algorithm respectively. Section 5 shows our evaluation results. Section 6 reviews the related work. Section 7 concludes the paper.

2. SYSTEM OVERVIEW

Identifying optimal cloud configuration is a crucial problem in the modern cloud era. In this regard, we are developing a cloud recommender system, referred to as Cloud Advisor. It aims to allow cloud users to explore various different optimal cloud configurations based on their preferences and requirements of their workload deployments. We achieve the goal in two steps. First, we accurately estimate the capability of the target cloud. Second, we enable comparison between potential offerings from different cloud providers, in terms of performance and price for the given workload. Figure 1 outlines the approach of Cloud Advisor that consists of offline modeling (solid arrows) and online recommendation (dashed arrows).

2.1 OFFLINE MODELING

For a given user workload, our system figures out its performance characteristics in terms of the workload’s resource usage patterns in a white-box test-bed (i.e., profiling). For the profiling, we have developed Cloud Meter that can capture the relative contribution of each resource type (e.g., CPU, memory, disk I/O, and network bandwidth) to the workload’s throughput. Based on the profiling, our system can build an abstract performance model (see Section 3.1 and 3.2). Meanwhile, Cloud Meter also captures the performance characteristics of target clouds by benchmarking their VMs to be used as building blocks of target cloud configurations. The performance characteristics of a target VM can be encoded to a capability vector, where each element represents the relative benchmark score of each resource type against the white-box test-bed (see Section 3.3). Note that the benchmarking...
process will be scheduled periodically, since those benchmarking results can dynamically change over the time.

2.2 Online Recommendation

Our system computes a near optimal configuration of each target cloud, in the context of throughput and price. Further, the system interactively adjusts the configuration based on user preferences (e.g., the maximum budget, the throughput goal, estimated hourly usage, and load, as shown in the top right part of Figure 1). Please refer to (Jung, 2013) for the detail implementation of Cloud Advisor interface. To do this, our system (i) estimates a capability vector using the abstract model, where each element in the vector represents the required capacity of each resource type to meet the throughput goal (see Section 4.1), (ii) with the capability vectors of target clouds collected from the offline modeling and the capability vector computed in the previous step, searches an optimal cloud configuration (i.e., combined VMs to run the target workload) until there is no more chance to minimize the price (see Section 4.2), and finally, (iii) provides comparison tables using search results.

Two key components, capability estimation and search for an optimal cloud configuration, to be described in the following sections.

3. Capability Estimation

One of the most important parts in the recommender system is accurately estimating the performance capability of each building block of cloud configuration (i.e., a VM) for a given workload. Here, the performance capability of VM is defined as the approximated maximum throughput that can be achieved using the VM for the workload. We develop Cloud Meter, which estimates such performance capabilities of different VMs in a target cloud. Using Cloud Meter, our approach first builds an abstract performance model based on the resource usage patterns of the workload measured in an in-house test-bed (i.e., a white-box environment). Second, it computes relative performance scores of many different VMs of a target cloud against the in-house cloud using benchmarking technique. Finally, it applies the collected performance scores into the abstract performance model to estimate performance capabilities of those VMs. This approach needs little cost and time to estimate the performance. This is because we do not need to deploy the complex workload itself into all possible VMs and cloud configurations to evaluate their performance capabilities.

3.1 Performance Characterization of Workload

For a given workload, Cloud Meter first characterizes the workload in the context of its resource usage patterns by deploying the workload into an in-house test-bed and computing the correlation of resource usages to workload throughput, while load changes. In our approach, we capture the change rate of resource usage (i.e., slope) of each resource type and the change rate of workload throughput until the capability is reached (i.e., knee point), while load increases.

The usage change rate before the knee point can approximately indicate the degree of contribution of each resource type to the throughput change and the performance capability. These change rates are used as parameters of our performance model that will be described in the following section. Figure 3 shows the change rates of throughput and
two representative resource types, while load increases over the time. In this example, the change rate of CPU is higher than memory usage. It indicates that CPU contributes more to the workload throughput than memory, and CPU can be bottlenecked first on its knee point. Note that the knee points of two resource types can occur at different points, since a bottlenecked resource type (e.g., CPU) can affect usages of the other resource types (e.g., memory usage).

To compute the change rates and then, build a model, we have to identify the knee points. Figure 4 illustrates our approach. At the end of measurement, Cloud Meter generates a linear line that connects the first measurement point to the last point and then, computes its length (i.e., \( z \) in the figure). At each measurement point, Cloud Meter can compute the length of the orthogonal line drawn from the linear line to the measurement point (i.e., the height \( h_k \) in the figure, where \( k \) is each measurement point). To compute the height of each measurement point, it generates two lines and computes their lengths (\( x_k \) and \( y_k \) in the figure). First line is drawn from the first measurement point to the current measurement point, and the second line is from the current measurement point to the last point. Then, using cosine rule and sine rule, it computes the height as following.

\[
h_k = x_k \sin(\cos^{-1}((x_k^2 + z^2 - y_k^2)/2x_kz))
\]

Finally, the knee point among all measurement points has the highest height from the linear line.

A workload simulator of Cloud Meter has been developed for the performance characterization as well. It generates synthetic loads with various data access patterns (e.g., the ratio of database write over read transactions and the ratio of CPU usage over I/O usage). If a historical load is available, the workload simulator can sort and re-play the workload to give systematic stress to the target application. We also consider that the test-bed is highly capable to run any type of workload such as CPU-intensive, memory-intensive, I/O intensive, or network-intensive.

3.2 BUILDING ABSTRACT PERFORMANCE MODEL

Based on the performance characterization, our approach defines performance models for resource types. As shown in Figure 3, throughput increases until the performance capability is reached. The performance capability is determined by some resource types that consume the most of their available capacities (i.e., bottlenecked). Hence, our system defines a quantitative performance model for each individual resource type to identify its correlation to the performance capability. Specifically, for each resource type \( j \), a quantitative performance model can be defined as,

\[
T_j = f(U_j | (C_j = c, \exists j \in R) \land (C_r = \infty, \forall r \in R, r \neq j))
\]

where \( T_j \) is the throughput to be achieved with the normalized usage rate, \( U_j \), over given capacity (i.e., \( C_j = c \)) of a resource type \( j \). \( R \) is a set of resource types, and \( r \) is a resource type in \( R \), where \( r \) is not equal to \( j \). We consider \( r \) has unlimited capacities so that we can compute the correlation of only \( j \) to \( T_j \).

To compute \( T_j \) using function \( f \), our system takes 4 steps. First, the system figures out the relation of load to the usage rate of the resource type. The relation can be defined as a linear function or, generally, as a function that has a curve for a resource type \( j \). Usage rates we consider in this paper are the total CPU that consists of user and system CPU usages, cache, memory, disk I/O, and network usages. More specifically, the function can be as follows.

\[
U_j = s_{ij}(a_j(2L - L_i^p) + \gamma_j) \quad (1)
\]

where \( L \) is the amount of load, \( p \) is used to minimize the square error (a linear function is a special case when \( p = 1 \)). \( a_j \) is the change rate (e.g., a slope in a linear function), and \( \gamma_j \) is an initial resource consumption of the current configuration.

We can obtain \( a_j, \gamma_j \), and \( p \) by calibrating the function to fit into actual curve. As mentioned in the previous section, in this fitting, we use the change rate before knee point. Then, \( s_{ij} \) is computed that is the relative performance score of a target VM \( i \). This will be described in the next section in detail. In the white-box, \( s_{ij} \) is set to 1.

Second, the relation of \( L \) to throughput \( T \) is defined as,

\[
T = \beta (2L - L_i^q) \quad (2)
\]

where \( \beta \) is the change rate of throughput, and \( q \) is used to minimize the square error (a linear when \( q = 1 \)). Similarly, we can obtain \( \beta \) and \( q \) by calibrating the function to fit into actual curve.

Third, we compute the capability based on the correlation of resource type \( j \) to \( L \). We can obtain a theoretical amount of load, when \( j \) reaches the full usage point using Equation 1 (i.e., theoretically extending the curve beyond the knee point with the same change rate \( a_j \) until \( U_j = 1 \)). Then, the obtained amount of load is applied to Equation 2.

Finally, the estimated capability of a target VM can be computed as follow.
3.3 Computing Relative Performance Scores

Although the same workload is deployed, target VMs (and target cloud configurations) have different performances each other due to their various different resource capacities and capabilities. To complete the abstract performance model (i.e., Equation 1) and estimate capabilities of such cloud configurations, we have to capture the performance characteristics of each target VM $i$, in terms of relative performance score $s_{i,j}$ for each resource type $j$. Using Cloud Meter, our approach collects relative performance scores based on resource capability measurements. These scores can be reused for any different workload later. In current implementation, Cloud Meter contains a set of micro-benchmark workloads including Dhrystone, Whetstone, cache capacity, system calls, and context switch that are integrated into UnixBench for CPU benchmarking, CacheBench for memory benchmarking, IOZone for disk I/O benchmarking, and our own network benchmark application. Cloud Meter is very useful when historical workload trace of an application in the target cloud is not available. Once it is available after the application is deployed, the application itself can be a benchmark of Cloud Meter and then, the historical data can be used to compute $s_{i,j}$ of a new workload that has similar performance characteristics.

Using measurements, our approach computes $s_{i,j}$ as, $s_{i,j} = \left( \frac{b_{i,j}}{b_j} \right) \left( \frac{a_{i,j}}{a_j} \right)$, where $a_j$ and $b_j$ are a given allocation of $j$ (e.g., the number of vCPUs or memory size of a VM) and the benchmarking measurement for $j$, respectively, in the white-box cloud configuration. Similarly, $a_{i,j}$ and $b_{i,j}$ are those in the target VM $i$.

By applying $s_{i,j}$ to Equation 1, we can obtain the performance capability of $i$. When we deal with CPU for $s_{i,cpu}$, the system CPU usage for $s_{i,cys}$ and user level CPU usage for $s_{i,sys}$ must be considered separately in the total CPU usage, since the system CPU usage is related to context switch and system call used for interrupts, allocate/free memory, and communicating with file system that can be different among cloud configurations. Thus, we extend $s_{i,j}$ for CPU as $s_{i,cpu} = \left( \frac{\alpha_{i}\alpha_{sys} + \alpha_{sys}}{\alpha_{cpu}} \right) / \alpha_{cpu}$, where $\alpha_{sys}$ is the increase rate of system CPU usage while $\alpha_{cpu}$ is the increase rate of the total CPU usage. These rates are captured from the fitting model step in Equation 1.

4. Search for Optimal Cloud Configuration

Using the performance model, Cloud Advisor identifies a near optimal cloud configuration, which can be composed of multiple VMs, for the workload and its throughput goal. In particular, we focus on parallel workload in this paper. To do this, our approach first generates capability vectors, each of which represents the performance of a target VM. Second, we encode the search problem into Knapsack Problem (Kellerer, 2004) and then, we develop a search algorithm to solve the problem in an efficient way.

4.1 Generating Capability Vectors for Given Workload

The first step to identify an optimal cloud configuration is generating a required capability vector using the performance model in the white-box test-bed. Each element of the required capability vector represents the performance capability value of each resource type to meet a throughput goal.

Specifically, the required performance capability value, remarked as $c_{i,j}$ of a resource type $j$ (i.e., the usage rate $U_j$ required to achieve the throughput goal $T^*$), can be given as $c_{i,j} = (a_j / \beta) T^* + \gamma_j$, when we consider a linear function in Equation 1 and 2. Here, $(a_j / \beta)$ indicates the normalized increase rate of the resource usage to achieve a unit of throughput. Thus, the equation indicates how much resource capability is required to meet $T^*$. Note that if $c_{i,j}$ is more than 1, it indicates that more resource capability is required to meet $T^*$ than currently available in the test-bed configuration. With this equation, the required capability vector for the workload and its throughput goal is a set of such required performance capability values and is represented as a form of $V^* = \{c^*_1, c^*_2, c^*_3, \ldots, c^*_r\}$.

We may obtain the required capability vector of a cloud configuration in a target cloud by directly deploying the given workload into it and perform aforementioned measurements. However, it is expensive and time consuming task, since there are many different cloud configurations to be evaluated, and the workload deployment is typically very complicated. Hence, our approach instead captures the relative performance capability value of each resource type $j$, remarked as $c_{i,j}$ of a target VM $i$ in a cloud, by using the benchmarking measurements that are used for performance score $s_{i,j}$. It is much simpler than deploying the complex workload itself into the target cloud. Moreover, benchmarking measurements can be reused for other workloads once these are continually updated to reflect any change of the cloud.

Specifically, the relative performance capability value $c_{i,j} = \left( \frac{b_{i,j}}{b_j} \right) \left( \frac{a_{i,j}}{a_j} \right) c_j$, where $(b_{i,j} / b_j)$ is the performance...
ratio based on benchmarking, \((a_{ij} / a_j)\) is resource allocation ratio, and \(c_j\) is the maximum resource usage rate for given capacity of \(j\) (typically, \(c_j = 1\)). Then, the capability vector of \(i\) can be represented as a form of \(V_i = [c_{i,1}, c_{i,2}, c_{i,3}, ..., c_{i,j}]\). Finally, our approach computes all target capability vectors from clouds of interests. Note that \(V_i\) is just the relative capability vector of \(i\) against the test-bed, while \(V^*\) is the capability vector required to achieve \(T^*\) and computed in the test-bed.

4.2 SEARCH ALGORITHM

Cloud Advisor explores various different cloud configurations offered from each cloud provider to identify a near optimal cloud configuration having a best price while meeting a throughput goal. If we can obtain the price per unit of resource usage, we can easily compute the total price. However, most cloud providers have pre-determined small cloud configurations with specific prices as VM types that have different CPU, memory, disk and network capacities (Cardosa, 2011). Some cloud providers such as Amazon Web Services use a dynamic pricing scheme for their virtual resources as well. In this paper, we assume a static pricing scheme that is used by many cloud providers. In our current implementation of the search algorithm, we also assume a parallel workload such as clustered parallel database transactions and a parallel data mining using MapReduce (Dean, 2004). For parallel workloads, the cloud configuration can have multiple heterogeneous VMs to handle loads in parallel with a load balancing technique such as one introduced in (Jung, 2012).

The capability vector \(V_i\) of each specific VM type can be computed as described in the prior section. We can also capture \(V^*\) that represents a resource capacity requirement to meet a throughput goal \(T^*\). Then, the search procedure is to fit those numerical capability values of pre-determined VM types into numerical capability values defined in \(V^*\). Especially, the search algorithm identifies a cloud configuration having a minimum price when it fits into the required capability vector \(V^*\).

As shown in Figure 5, we illustrate \(V^*\) with a set of containers, each of which has a specific size (i.e., the numerical capability \(c^*_j\) of resource type \(j\) as described in Section 4.1). Then, we can convert the problem into Knapsack problem (Kellerer, 2004), since the search algorithm tries to fill containers with items (i.e., capability vectors \(V_i's\)) that have different values (i.e., prices) while aiming at a minimal total price. Figure 5 shows a snapshot of the search procedure, when containers are filled with two different VMs. Here, \(V^*\) is \([10.0, 9.8, 7.0, 5.0]\) representing CPU, memory, disk, and network resource types. \(V_1\) and \(V_2\) are \([1.0, 1.2, 0.8, 0.5]\) and \([2.0, 1.2, 0.8, 1.0]\), respectively. After two capability vectors are inserted, \(V^*\) becomes \([7.0, 7.4, 5.4, 3.5]\) and is further filled with more capability vectors.

The problem we tackle is finding such VMs to minimize the cumulated price of VMs, while all containers are completely filled. More specifically, the optimal cloud configuration will have 0 distance for each resource type \(j\) between \(c^*_j\) of \(V^*\) and cumulated capabilities of VMs (i.e., \((c^*_j - \sum_{i=1}^{m} n_i) = 0\), where \(m\) is the number of VM instances in a cloud configuration) while having the minimum cumulated price of VMs. The distance can be computed as follows.

\[
D = \sum_{j=1}^{m} \max ((c^*_j - \sum_{i=1}^{m} n_i), 0) \quad (3)
\]

In this equation, if \((c^*_j - \sum_{i=1}^{m} n_i) < 0\), its distance is considered as 0 since it indicates enough capability. Otherwise, containers have to be filled with more VMs since the throughput goal is not met. Note that all containers must be fully filled even if a container’s size is very small (i.e., the required capability of a resource type is relatively small such as network bandwidth in Figure 5). This is because the bottleneck can occur in any resource that has not enough capability even if the resource type is not critical to achieve the throughput goal.

Identifying such cloud configuration is not trivial, since there are many different combinations of VMs that can meet the throughput goal. We can formulate the problem into Integer Linear Programming (ILP) as follows.

\[
\text{Minimize} \quad \sum_{i=1}^{m} n_i p_i \\
\text{Subject to} \quad \sum_{i=1}^{m} n_i c_{ij} \geq c^*_j \quad \forall j \in R, \ n_i \geq 0, \ p_i \geq 0, \text{ and } c_{ij} \geq 0
\]

, where \(m\) is the number of VM types, \(n_i\) is the required number of VM instances of each VM type \(i\), and \(p_i\) is the price of \(i\). Then, to solve this ILP, we can use generic solutions such as Tabu search (Glover, 1989), Simulated annealing (Granville, 1994), and Hill climbing (Russell, 2003).

In our approach, we develop an efficient search algorithm based on the detection of resource bottlenecks and the best-first search method, rather than blindly exploring the search space. Algorithm 1 shows the basic best-first

---

**Figure 5. Knapsack problem with capability vectors**
The algorithm takes two inputs, the required capability vector \( V^* \) and a set of pre-determined VM types \( \Gamma \). All capability vectors have been computed as described in the previous section. To obtain the optimal combination of VM instances, \( \bar{V}_{\text{curr}} \), our algorithm first takes an instance of each VM type in \( \Gamma \) as a candidate (line 3~5 in Algorithm 1) and then, generates different combination of VM types over the search algorithm. We extend this algorithm for efficient search by reducing the search space.

The overhead of the search procedure is caused by too many candidate combinations to be generated (i.e., a large search space) in Algorithm 1. Hence, we extend the algorithm to significantly reduce the search space using two pruning techniques.

Conservative Gradient Search (CG). Our algorithm checks the gradient of each resource capability (i.e., any improvement toward the goal by adding the capability) in the current cheapest candidate against a previously selected one by replacing the line 8 of Algorithm 1 with Algorithm 2.

For the current cheapest combination \( \bar{V}_{\text{curr}} \), CG checks its size (i.e., the number of VM instances combined into it), and chooses it as the new search starting point if the size is less than the size of the candidate chosen in the prior iteration, \( \bar{V}_{\text{prior}} \). This is because the size and price of newly generated candidate have been increased in Algorithm 1, so that \((\bar{V}_{\text{curr}} \prec \bar{V}_{\text{prior}})\) means that \( \bar{V}_{\text{curr}} \) has been generated earlier, but it does not have a chance to be explored yet. If it is not the case, our algorithm further checks if \( \bar{V}_{\text{curr}} \) has any resource type still required to be filled into \( V^* \) (i.e., \( \bar{V}_{\text{curr}} \cap V^* \neq \emptyset \)) and reducing the gap more than \( \bar{V}_{\text{prior}} \) has done (i.e., \( \bar{V}_{\text{curr}} \backslash V^* \geq \bar{V}_{\text{prior}} \backslash V^* \)). Alternatively, we can check this condition for all resource types or bottlenecked one (see below), instead of any resource type meeting the condition. However, this alternative method sometimes significantly affects the optimality of our algorithm, while it speeds up the search further. Therefore, we choose the conservative method to find the optimal combination.

Bottleneck-aware Proactive Search (BP). Our algorithm explores the search space along with biased paths in the search space that have more remaining distances (i.e., focuses on resource types to be potentially bottlenecked). To do this, we insert the following two functions after line 11 in Algorithm 1.

The first function is to figure out the bottlenecked resource type \( j^* \) and compute the remaining distance \( g^* \) for \( j^* \). \( j^* \) is the resource type of \( \bar{V}_{\text{curr}} \) that has the maximum distance between \( V^* \) and \( \bar{V}_{\text{curr}} \) (i.e., \( V^*, e^* - \bar{V}_{\text{curr}}, c_j \)). The second function is to select top \( K \) VM types based on \( j^* \) and \( g^* \) from the set of all VM types \( \Gamma \) and store the selected top \( K \) VM types into \( I_{\text{sort}} \) to be used for generating a set of new candidate combinations. Then, our algorithm uses smaller candidate types (i.e., \( I_{\text{sort}} \)) instead of \( \Gamma \) in line 12 of Algorithm 1. The top \( K \) VM types should fill the distance of the bottlenecked resource with less amount of price than the other configuration types in \( \Gamma \). Thus, this function computes the potential price when using each VM type in \( \Gamma \) to fill the distance of the bottlenecked resource as \( upper(g^*, V_j, c_j) V_i, p \), where \( V_i \uparrow \Gamma \) is the type, and \( V_j, p \) is the price of the
type. This means that the potential price is computed by considering the required number of VM instances to fill $g^*$. Then, the function sorts all VM types in $f$ by their potential prices. Note that our algorithm checks remaining distance of resource types and figures out a bottlenecks type resource every time in the iteration, since the potential bottleneck keeps migrating while adding resources. The number of VM types $K$ in the algorithm can be determined empirically or dynamically over the iteration (i.e., reducing the number as the distance decreases). The higher $K$ is, the better chance there is to increase the optimality, but the slower the search would be. In Section 5, we show the impact of the choice of $K$ on the search speed and the optimality.

Using these techniques (i.e., CG and BP) on the best-first search (i.e., Algorithm 1), we can reduce the search space by pruning out some candidate combinations that can achieve the throughput goal, but can be more expensive than the optimal combination. Hence, we can compute the near optimal combination (i.e., a cloud configuration) much more efficiently compared to blind exploration.

5. EXPERIMENTAL EVALUATION

5.1 EXPERIMENTAL SETUP

To evaluate our approach, we have used an online auction web transaction workload, RUBiS\(^3\), that is deployed on servlet server (Tomcat) and back-end database server (MySQL). The workload provided by RUBiS package consists of 26 different transaction types. In our experiments, we focus on clustering of database servers so that we have modified the original workload to intentionally place loads on the database servers. This has been done by reducing simple HTML transactions that lightly place loads on the servlet server, and by increasing the rate of database query transactions. Then, we have created two different workloads by changing the rate of database write transactions in the original workload. The light-write workload has 5% write transactions, while the heavy-write workload has 50% write transactions out of all transactions in the workload.

We have prepared a VM type in our test-bed cloud that is configured with 2 vCPUs, 4 GB memory, and 40 GB disk. Ubuntu 10.04 operating system is loaded in the VM. Instances of this VM type have been deployed into our Intel blade with KVM virtualization. We have used this VM type to build our simple white-box cloud configuration (i.e., a single VM instance) and clustered white-box cloud configuration (i.e., two VM instances to configure a database cluster). We call these configurations as W-2-4-K (i.e., White-box having 2 vCPUs and 4 GB memory with KVM virtualization) and W-4-8-K for the simple and the clustered configurations, respectively.

For evaluating the capability estimation (Section 5.2), we have set up a VM type as a black-box. An instance of this VM type is called B-4-2-X (i.e., Black-box having 4 vCPUs and 2 GB memory with Xen virtualization). It has also 80 GB disk space running on AMD server. We have set up another small black-box VM type to be compared. An instance of this small VM type is called B-1-2-K (i.e., Black-box having 1 vCPU and 2 GB memory with KVM). It has disk space of 40 GB and runs on Intel server.

We have prepared other VM types obtained from Rackspace that is a well-known cloud infrastructure provider. There are various VM types in Rackspace configured with different number of vCPUs, memory size, and disk size. We have used 6 VM types, where they vary from 1 to 8 vCPUs, 1 to 30 GB memory, 40 GB to 1.2 TB disk space, and 30 to 300 Mbps network. Their prices are from $0.06 to $1.20 per hour. Ubuntu 10.04 has been installed in all these VMs. These VMs have been used in our experiments for the search algorithm (Section 5.3).

Table 1 outlines these VMs.

<table>
<thead>
<tr>
<th>Name</th>
<th>vCPU (No.)</th>
<th>Mem (GB)</th>
<th>Disk (GB)</th>
<th>Network (Mbps)</th>
<th>Price ($/hr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-1-1-X</td>
<td>1</td>
<td>1</td>
<td>40</td>
<td>60</td>
<td>0.06</td>
</tr>
<tr>
<td>B-2-2-X</td>
<td>2</td>
<td>2</td>
<td>80</td>
<td>120</td>
<td>0.12</td>
</tr>
<tr>
<td>B-2-4-X</td>
<td>2</td>
<td>4</td>
<td>160</td>
<td>200</td>
<td>0.24</td>
</tr>
<tr>
<td>B-4-8-X</td>
<td>4</td>
<td>8</td>
<td>320</td>
<td>300</td>
<td>0.48</td>
</tr>
<tr>
<td>B-6-15-X</td>
<td>6</td>
<td>15</td>
<td>620</td>
<td>400</td>
<td>0.90</td>
</tr>
<tr>
<td>B-8-30-X</td>
<td>8</td>
<td>30</td>
<td>1200</td>
<td>600</td>
<td>1.20</td>
</tr>
</tbody>
</table>

Table 1. Black-box configurations in Rackspace

5.2 PERFORMANCE CAPABILITY ESTIMATION

To evaluate the capability estimation, we have deployed two RUBiS workloads into W-2-4-K and built the abstract performance models (i.e., Equation 1 and 2 in Section 3.2). In this section, we show that our model can accurately estimate the maximum throughput of the target VM for any workload using these two workloads that have different performance characteristics from each other.

Heavy-Write Workload. As shown in Figure 6, our workload generator records throughput (i.e., the number of responses per 3 minutes) while increasing the number of concurrent users by 50 every 3 minutes. The throughput change and the maximum throughput of W-2-4-K are shown in the figure. Figure 6 also shows the maximum throughputs of the other two target configurations, B-1-2-K and B-4-2-X. Our estimation approach will estimate these maximum throughputs. We note that change rates of all three configurations are almost identical in low load. This is because they have enough capability to deal with such load. However, their maximum throughputs are different due to their different capabilities.

To compute such capabilities, the resource usage patterns (i.e., usage changes) of 5 resource types in W-2-4-K have been plotted as shown in Figure 7. We can observe that CPU user, CPU system, and memory have notably affected the throughput of the heavy-write workload since

\(3\) RUBiS is available at http://rubis.ow2.org
their usage changes are much higher than the other 2 resource types (i.e., disk I/O and network I/O) in this configuration.

We have captured the abstract performance model of the heavy-write workload into Table 2. To compute the change rates and the parameters of the model in the table (i.e., $\alpha$, $p$, and $\gamma$), our approach has first figured out the knee points of these trajectories (i.e., the red circles in Figure 6 and 7) using the technique described in Section 3.1. Note that the knee points of memory, disk I/O, and network I/O are located at the last measurement points. This is because there are no obvious knee points of these resource types. Similarly, our approach has figured out the knee point of throughput graph, and captured $\beta = 5.7$ and $q = 0.98$ used in Equation 2, as the throughput increase rate and square error, respectively.

Cloud Meter has been deployed into W-2-4-K, B-1-2-K, and B-4-2-X configurations and performed benchmarking of resource types to compute performance scores of these configurations as described in Section 3.3. Figure 8 shows the performance scores of 6 different resource types. Note that the scores of B-4-2-X are quite different from other two configurations since the ratio of CPU and memory allocations is different, and B-4-2-K is configured with different type of architecture (i.e., AMD) and virtualization (i.e., Xen).

By applying these scores to the abstract performance model, we can estimate capabilities of B-1-2-K and B-4-2-X as shown in Table 3.

Based on our approach described in Section 3.2, CPU is bottlenecked in B-1-2-K (i.e., $T_{cpu}$ is the minimum in the column of Table 3), similar to W-2-4-K as shown in Figure 7. In B-4-2-X, it however turns out memory is bottlenecked (different from other two configurations). This is because it has enough CPU allocation, but the bottleneck is migrated to memory in this case. Compared to the measured maximum throughputs, the error rate is less than 10% (for B-1-2-K, it is 8.75%, and for B-4-2-X, 6.98%).

When we see the resource usage patterns in Figure 7, it seems that CPU is bottlenecked in W-2-4-K, but it turns out that memory can be bottlenecked in different configurations. Additionally, using B-4-2-X instances can be over-provisioning to achieve a little capability increase, when we compare to W-2-4-K. Finally, we can also find the bottleneck can be migrated between resources as adding more allocation into bottlenecked resources. Hence, the recommender system must accurately capture such resource usage patterns and bottleneck migrations for workload.

**Light-Write Workload.** Similarly, we have measured the throughput and resource usage patterns in W-2-4-K and then, estimated the capabilities of the other two black-box configurations for this workload. Figure 9 shows the knee point and the throughput change rate, while Figure 10 shows the knee points and usage rates of 5 resource types.

We note here that this workload has different performance characteristics from the heavy-write workload. As shown in Figure 9, the maximum throughput of W-2-4-K is higher than one of B-4-2-X in this workload. As shown

![Figure 6. Throughputs of three configurations (VMs) in heavy-write workload](image6)

![Figure 7. Resource usages of heavy-write workload](image7)

**Table 2. Parameters of abstract performance models of heavy-write workload**

<table>
<thead>
<tr>
<th>Resources</th>
<th>$\alpha$</th>
<th>$p$</th>
<th>$\gamma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>0.08</td>
<td>1.01</td>
<td>8</td>
</tr>
<tr>
<td>CPU User</td>
<td>0.05</td>
<td>1.02</td>
<td>5</td>
</tr>
<tr>
<td>CPU Sys</td>
<td>0.03</td>
<td>0.98</td>
<td>3</td>
</tr>
<tr>
<td>Memory</td>
<td>0.02</td>
<td>1.01</td>
<td>21</td>
</tr>
</tbody>
</table>

**Table 3. Capability estimates of heavy-write workload**

<table>
<thead>
<tr>
<th></th>
<th>W-2-4-K</th>
<th>B-1-2-K</th>
<th>B-4-2-X</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{cpu}$</td>
<td>3241.02</td>
<td>13681.25</td>
<td>7586.55</td>
</tr>
<tr>
<td>$T_{mem}$</td>
<td>8165.41</td>
<td>7586.55</td>
<td>3483818.44</td>
</tr>
<tr>
<td>$T_{disk}$</td>
<td>4178010.09</td>
<td>1715384.62</td>
<td></td>
</tr>
<tr>
<td>$T_{network}$</td>
<td>2097002.99</td>
<td>1715384.62</td>
<td></td>
</tr>
</tbody>
</table>
in Figure 10, this may be caused by that the usage change rate of memory resource in this workload is higher than in the previous workload, and W-2-4-K has more memory capacity than B-4-2-X, although B-4-2-X has more CPU capacity than W-2-4-K. However, we need to further analyze the resource usage patterns since system CPU obviously has higher change rate than user CPU, and B-4-2-X has relatively better system CPU score than user CPU score as shown in Figure 8.

To analyze this situation and estimate the capabilities, we have captured the performance model in Table 4 with $\beta = 5.0$ and $\gamma = 0.99$. Then, we have computed the estimates of the maximum throughputs. Table 5 summarizes the results. Note that we can reuse the performance scores used in the heavy-write workload for this estimation.

Although Table 4 indicates that the system CPU usage change has the highest rate in this workload, Table 5 shows that the bottleneck of B-4-2-X is still memory resource, while the bottleneck of B-1-2-K is CPU resource. Compared to the measured maximum throughputs, the error rate is less than 10% in this workload.

Experiment results indicate that our performance modeling approach is accurate enough to be applied to the search for the optimal cloud configuration in our recommender system.

### 5.3 Scalability of the Search Algorithm

We have obtained all capability vectors of 6 VM types of Rackspace and then, evaluated the accuracy of our search algorithm using the light-write and heavy-write workloads. The resulting configuration (i.e., database cluster using multiple VMs) computed by our search algorithm (i.e., Bottleneck-aware Proactive Search (BP) described in Section 4.2) has been compared with the configuration computed by a brute-force search (i.e., exhaustive searching for the best configuration among all possible VM combinations). We have used a relatively low throughput goal (i.e., 20K) so that the best resulting configuration has consisted of only 5 low-end VM instances. We have deployed a simple load balancer in a separate VM that forwards user requests to the VM cluster based on their performance capabilities. We can see that BP returns the exactly same configuration (i.e., 5 low-end VM instances)
with the brute-force search in this small setup. Then, we have deployed the result configuration into Rackspace infrastructure to see if it can meet the throughput goal (i.e., 20K). Although the VM cluster has been a little over-provisioned (i.e., 21.3K) against the goal, the error is still around 5% in light-write workload. For the heavy-write workload, the resulting VM cluster has been under-provisioned (i.e., 18.2K) with around 10% error rate. We have analyzed the cause of under-provisioning further with our clustered white-box configuration (i.e., W-4-8-K), and figured out that the network resource has been consumed a little more than we have estimated, to synchronize database writings among servers. In our current on-going work, we are improving the accuracy of our recommender system by considering such database synchronization.

We have conducted an extensive simulation to evaluate the potential scalability and optimality of our search algorithm. Currently, we plan to deploy a large-scale MapReduce cluster with a parallel data mining workload that may need up to several hundreds of VMs in the cluster. Thus, the search algorithm has to be scalable to deal with such large workload. In experiments, we have run our search algorithm with all capability vectors of 6 VMs and W-1-2-K to compute the configuration and its total price. We have increased the throughput goal from 60K to 240K while measuring the duration of our search algorithm and the total price of resulting configuration. The higher throughput goal is, the more VMs are combined into the configuration, and the longer the search algorithm runs. Three different search algorithms have been compared in this evaluation.

- **Naive Best-First Search (NBF):** It uses the basic best-first search algorithm as shown in Algorithm 1 of Section 4.2. This algorithm is not so scalable since it explores the cheapest candidate step by step while generating all possible candidates in the iteration. We have used this algorithm as a baseline.

- **Conservative-Gradient Search (CG):** It integrates the conservative gradient search (described in Section 4.2) only on NBF.

- **Bottleneck-aware Proactive Search (BP):** It integrates the bottleneck-aware search to prune the search space with CG on NBF as described in Section 4.2. To show the impact of the pruning parameter $K$ on the optimality and the scalability of our algorithm, We have used the different $K$ values (i.e., $K = 1, 3, 5$ indicating the number of top $K$ VM types out of the total 6 VM types to generate new candidate combinations).

Experiment results show that our BP algorithm with $K = 3$ is scalable (Figure 11), while having reasonable optimality (Figure 12). NBF in Figure 11 shows an obvious exponential increase as the throughput goal increases. Although CG has better scalability than NBF, it still shows the exponential increase. This is mainly because capabilities per unit price of given VM types are not so different in our experimental setup so that the pruning rate is very low. When we have set $K$ is to 5 in our BP algorithm, we cannot obtain the good scalability since it has still generated numerous candidates in the queue to be evaluated later in the search procedure. However, BP algorithm with $K = 3$ starts to aggressively prune the search space based on potential bottlenecks. Hence, we can achieve a good scalability that is close to BP with $K = 1$ (i.e., the case that increases the search space linearly). Meanwhile, BP with $K = 3$ shows only a little loss of optimality. As shown in Figure 12, BP with $K = 3$ returns at most 4% more price than the price computed by NBF. CG and BP with $K = 5$ is almost identical to NBF since it explores the most of candidates that are explored by NBF. However, BP with $K = 1$ shows the significant loss of the optimality caused by ignoring some candidates that can lead to the optimal configuration.

6. RELATED WORK

Cloud has gathered pace, as most enterprises are moving toward the agile hosting of applications in public clouds. In this regard, many researchers have focused on three
different directions: (i) updating application architecture to move from legacy systems to clouds (Chauhan, 2011), (ii) evaluating different clouds’ functional and non-functional attributes for informed decision on which cloud to host applications (Jayasinghe, 2011; Huang, 2010; Calheiros, 2011; Jayasinghe, 2011; Li, 2012; Cunha, 2013), and (iii) efficient orchestration of virtual appliances in a cloud (Keahey, 2008), which may also include negotiations with users (Venugopal, 2009). This paper complements these different directions by enabling recommendation of cloud configuration to suit application requirements. In this regard, cloud capability estimation methodology has been developed using benchmark-based profiling. Previous work has principally focused on comparing rudimentary cloud capabilities using benchmarks (Jayasinghe, 2011; Huang, 2010) and automated performance testing (Malkowski, 2009). This paper focuses on detailed characterization of cloud capabilities for user workload in various target clouds.

Recently, many cloud providers such as (Calheiros, 2011; Li, 2012) have further offered testing paradigm to enable the evaluation of various workload models on different resource management models. However, scaling such offerings requires identifying various cloud capability models (Gao, 2011). This paper fills the gap by developing a generic methodology to characterize and model cloud capabilities. The methodology is applicable to black-box clouds since the modeling of cloud capabilities are based on experiments on clouds only using externally observable characteristics. The model is used to estimate performance of applications in target clouds and then, such estimates are used to efficiently search near optimal cloud configurations.

Building performance models and diagnosing performance bottlenecks for Hadoop clusters have been explored in (Gupta, 2012). This paper further develops methodology to recommend cloud configurations to meet user demands. Orchestration of virtual appliances in a cloud (Keahey, 2008) and efficient allocation of instances in cloud (Venugopal, 2009) have been addressed previously to meet user demands. However, such approaches are typically non-transparent to users. This paper, on the other hand, makes the recommendation transparent by allowing users to make choices accordingly based on cloud capability estimation.

The importance of estimating cloud configuration has been mentioned in (Cunha, 2013) as well, and they have proposed an approach to collect the estimates from public clouds. However, in their approach, users must need a help from application expert to deploy the application and collect estimates. Our approach can estimate the cloud configuration using simple benchmark workloads, rather than deploying the complex user application itself.

7. Conclusions

This paper has aimed to address one major barrier given to recommender system that identifies an optimal cloud configuration for complex user workload based on benchmark-based approximation. Especially, we have shown that such estimation and recommendation can be done even under black-box environments. To achieve this, our system generates the capability vector that consists of relative performance scores of resource types and then, our search algorithm identifies a near optimal cloud configuration based on these capability vectors. Our experiments show our approach estimates the near optimal cloud configuration for workload within 10% error, and our search algorithm is scalable enough to apply for a large scale workload deployment. We currently work on applying our approach to a large-scale MapReduce job by extending the current approach. In this case, we are considering the time factor (e.g., the number of hours to be used for each VM) into Integer Linear Programming problem, defined in Section 4.2, for so-called bag-of-tasks as introduced in (Gutierrez-Garcia, 2012).

B. References


Jung, G., Gnanasambandam, N., & Mukherjee, T. (2012). Synchronous parallel processing of big-data analytics to optimize performance in...


Authors

**Gueyoung Jung** received his Ph.D. in Computer Science from Georgia Institute of Technology. His research background is in Distributed Computing, Autonomic Computing, and Services Computing. In particular, his research interests are in designing systems and analytical modeling to predict and optimize QoS in large-scale, data-intensive distributed systems. He has authored numerous papers in prestigious conferences, journals, and book chapters. Before starting his graduate program, he was a system developer for five years. Since he joined Xerox Research Center in 2010, he has participated in some cutting-edge research projects in broad areas. His project portfolio includes Optimal Cloud Deployment, Services Composition, and Big Data Analytics. He has filed more than 20 US patent applications.

**Tridib Mukherjee** is a research scientist at Xerox Research Center, India and works in the broad areas of Distributed Computing, Cloud Computing, Green Computing, Sensor Networks, and Services Computing. Tridib has been involved in cutting edge research in these areas since joining the Xerox Innovation Group in 2011. He has published numerous articles in reputed journals, conferences, book chapters and books. He has also filed many US patent applications. Tridib has been serving in the technical program committee of many top conferences. He has also co-presented tutorials and involved in organizing workshops in prestigious conferences. Tridib received his Ph.D. in Computer Sc. from Arizona State University. Tridib was a finalist of the IBM Ph.D. fellowship award in 2007.

**Naveen Sharma** directs the Computing and Information Services Lab (CISL) at the Xerox Research Center Webster. His technical leadership in distributed and cloud computing and data intensive computing played a significant role in the formation of CISL. Naveen joined the Xerox Research Center in 1994. He has extensive experience in technology transfer and the delivery of product ready code to Xerox business groups. Prior to joining Xerox, Naveen was a research assistant professor at the Institute of Computational Science, Kent State University in Ohio. Naveen has a doctorate in computer science. He holds more than 25 U.S. patents, and has published more than 20 research papers.

**Frank Goetz** is responsible for overseeing research, development and transition of technology options targeted for our Information Technology business. His early years were in product application software development for Xerox production. As a member of the Research Center, Frank led a team working on advanced distributed systems architecture and technology for enterprise output management. He has contributed to architecture and technology for Xerox products. Frank received his BS degree in Electrical Engineering from the University of Illinois at Urbana-Champaign and his MS degree in Computer Science from the Rochester Institute of Technology.

**Julien Bourdailet** is a research scientist at Xerox Research. His research interests include artificial intelligence, social media, cloud computing, and natural language processing. He filed more than 10 patent applications and published more than 20 papers in prestigious conferences and journals. He received his PhD from University Pierre and Marie Curie, Paris, France. He held a post-doctoral position at the University of Montreal, Canada; and worked in several cutting-edge startups.