Memory Scaling of Cloud-based Big Data Systems: A Hybrid Approach

Xinying Wang, Cong Xu, Ke Wang, Feng Yan, Dongfang Zhao

Abstract—When deploying applications with dynamic and intensive memory footprint to big data systems on public clouds, one important yet challenging question to answer is how to select a specific instance type whose memory capacity is large enough to prevent out-of-memory errors while the cost is minimized without violating performance requirements. The state-of-the-practice solution is trial and error, causing both performance overhead and additional monetary cost. This paper investigates two memory scaling mechanisms in public clouds: physical memory (good performance and high cost) and virtual memory (degraded performance and no additional cost). In order to analyze the trade-off between performance and cost of the two scaling options, a performance-cost model is developed that is driven by a lightweight analytic prediction approach through a compact representation of the memory footprint. In addition, for those scenarios when the footprint is unavailable, a meta-model-based prediction method is proposed using just-in-time migration mechanisms. The proposed techniques have been extensively evaluated with various benchmarks and real-world applications on Amazon Web Services: the performance-cost model is highly accurate and the proposed just-in-time migration approach reduces the monetary cost by up to 66%.

1 Introduction

While increasingly more modern applications are turning from compute-centric to data-centric, many systems have emerged to overcome the new challenges brought by the so-called big data. Representative big data systems include Hadoop [14], Spark [3], TensorFlow [38], Myria [31], and SciDB [34], all of which share the same paradigm—data parallelism, assuming the underlying infrastructure is a shared-nothing cluster. Although these systems had greatly lowered the technical barrier for parallel processing (comparing to, for instance, OpenMP [33], MPI [29]), our prior work [28] showed that new challenges are emerging from those big data systems such as application migration, memory management, among many others.

- *X. Wang and F. Yan are with University of Nevada, Reno, United States. Email: xinyingw@nevada.unr.edu, fyan@unr.edu*
- *C. Xu is with HP Labs, United States. Email: cong.xu@hpe.com*
- *K. Wang is with Mircosoft Azure, United States. Email: ke.wang@microsoft.com*
- *D. Zhao is with University of Nevada, Reno and University of California, Davis, United States. Email: dzhao@unr.edu*

As a concrete example, when deploying an astronomical application, namely Large Synoptic Survey Telescope (LSST [23]), to a big data management systems Myria [31] on the Amazon Web Services (AWS) public cloud [2], the converted code scales up to 8 visits of sky surveys but then starts to experience out-of-memory (OOM) errors for 12 or more visits because the allocated data to each node exceeds the physical memory capacity allocated popular big data systems such as Spark and SciDB [28]. We had to spend considerable time to completely rewrite the original application using multi-query to avoid memory errors.

Data intensive applications usually consume lots of memory, which can be quite expensive if not carefully planned. More importantly, many big data applications generate a highly dynamic amount of intermediate data that needs to be persisted in memory for performance. The OOM error in big data applications is often deemed as one of the most frustrating errors as it usually implies that the developers would need to spend a lot of time in further splitting the already complex application with finer granularity, reconfiguring the underlying big data system, redeploying the big data application, and recomputing many time-consuming results. Despite all such efforts, the application is not guaranteed to work without OOM errors—possibly making all the time and resource investment worthless. Part of the challenge comes from the unpredictability of memory footprint when the input data size changes. An intuitive solution would be to estimate the memory footprint based on different input sizes, which, unfortunately, is also challenging because in the real world the relationship between the two could be nonlinear in one of our more recent works [20]. The number of streamlines represents the input size for an Magnetic Resonance Imaging (MRI) application detailed in [28].

One conventional way to scale memory is using operating system's virtual memory (e.g., swap in Unix-like systems). Compared to scaling up physical memory (e.g., selecting a more powerful instance[1]), virtual memory usually yields degraded performance (thus potentially longer running time and higher cost). Therefore, an effective way

1. For memory-intensive applications, memory is the bottleneck, so a more powerful instance with better other resources (e.g., CPU, networking) would not necessarily improve the runtime performance. In the applications studied in this paper, we observed less than 100% utilization in resources other than memory capacity, such as CPU cycles and memory access bandwidth.

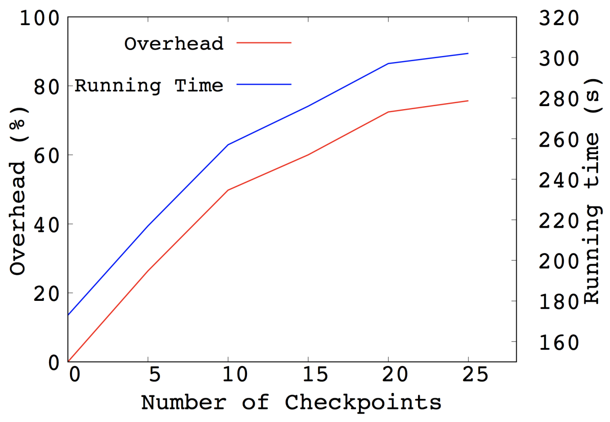


Fig. 1: Checkpointing Overhead

to determine the performance and cost impact when using different amount of virtual memory is highly desirable for users to make decisions. To quantitatively study the trade-off between performance and cost caused by virtual memory, the first goal of this work aims at building a performance-cost model that can accurately estimate the performance and cost using virtual memory.

Public cloud vendors provide a rich selection of scaling options (e.g., different instance types with various memory capacities) and a flexible pay-as-you-go pricing scheme. Unfortunately, preventing OOM errors in the real-world is even more challenging because we need to consider the monetary cost not applicable to conventional clusters. One native approach is simply choosing the most powerful instances, which usually incurs resource under-utilization and unnecessary monetary cost. The current state-of-the-practice approach in industry is trial and error using checkpoint: we launch an instance with small memory and periodically checkpoint runtime memory states; upon crash, we launch a new instance with larger memory capacity to, hopefully, meet the memory demands. Trial-and-error incurs significant performance overhead and monetary cost as saving and restoring large amounts of memory states can be highly expensive. As a concrete example, Figure 1 shows both the normalized overhead[2] in % and the actual running time in seconds when different numbers of checkpointing are applied to a real-world application used in [28] using a popular checkpointing tool CRIU [9]. We observe a fast-growing trend of performance overhead incurred by checkpointing, which motivates us to develop a just-in-time migration mechanism that only performs checkpointing just before the OOM occurs—the second objective of this work.

This paper aims to answer the following research questions: *how could we prevent data-intensive applications from experiencing OOM errors while retaining high resource utilization and low monetary cost.* To this end, we propose two techniques, one building on virtual memory of the operating system (OS) and the other inspired by statistical prediction models aiming to eliminate the overhead of the conventional checkpoint-based mechanism.

The proposed memory scaling methodology is based on two scenarios: 1) when some information of memory footprints is known a priori; 2) when the memory footprints is unknown in advance. Specifically, in the first scenario, we assume the swap access patterns are well-studied. Here the swap access pattern is defined as trend of the performance slowdown versus swap portion, e.g., exponential trend or uniform trend. This is true for many batch-processing applications[3], for example, in areas like high-performance computing [19] and scientific applications [49]. Specifically, we build models that predict applications' performance slowdown and monetary benefit (or, loss) according to the proportion of virtual memory being used. Experiments show that our models are highly accurate as they exhibit only 1% – 4% error rates when testing with multiple real-world applications in AWS.

In the second scenario, we do not impose any assumptions on the memory access patterns and propose a runtime estimation method. We introduce meta-models that can predict memory footprint with dynamic adjustment according to the application's own traits at runtime. The meta-models are applied in a heuristic manner, meaning that the application is deployed to instances in the increasing order of their memory capacity and gets migrated to more powerful (and more costly) instances only when the meta-models determine an OOM is forthcoming. Experiments show that this approach incurs significantly lower monetary cost than both the checkpoint-based approach and the naive pure-memory solution by up to 66%.

To summarize, this paper focuses on developing a practical system tool for preventing OOM error while balancing the performance and cost trade-off for big data applications in the public cloud rather than making theoretical contributions. Specifically, we make the following three main contributions.

(i) We develop analytic models to predict applications' performance and cost when various portions of virtual memory are used in public clouds. As a result, users will know how, and by how much, virtual memory will affect the overall performance and cost of their applications before actually executing them.

(ii) We propose multiple cost-effective approaches for preventing OOM error in public clouds. Instead of periodically checkpointing memory states, the proposed elevation-migration approaches incur only one batch of memory accesss when encountering memory depletion.

(iii) We extensively evaluate the proposed techniques using benchmarks and real-world applications. Experimental results demonstrate high effectiveness of both techniques on AWS.

2 Related Work

Based on the memory footprint patterns, big data applications can be categorized into two types. Type A: relatively deterministic memory footprint and Type B: highly dynamic memory footprint. Applications implemented in Spark [3] and other similar systems belong to Type A as they are usually READ intensive and only generate relatively small or deterministic amount of intermediate data. For this type of applications, their memory footprint can be proactively measured through simple profiling. For instance, in Spark,

2. Assume the running time of two different number of checkpoints are T_1 and T_2, the normalized overhead is defined as $(\frac{T2-T1}{T1}) \cdot 100\%$.

3. Streaming-data applications are beyond the scope of this paper

one can create an RDD to put into cache, and then look at the Storage page in the web UI to figure out how much memory the RDD is occupying as the size of RDD is deterministic [4]. MRI [28] and Didi [11] belong to the Type B big data applications as their memory footprint is highly dynamic due to the significant change of intermediate data that they produce in run time. We focus on Type B big data applications in this paper as their memory footprint is highly dynamic during runtime and thus challenging to be proactively determined and often result in OOM errors.

Memory management in cloud computing recently draws plenty of research interests in the community. One of the hot topics is predicting memory usage to prevent application crashes or detect abnormality, and a common method is using time series. In particular, Spinner et al. [36] proposed to predict the memory usage of applications on virtual machine in a proactive manner using time series analysis (with a training period usually in terms of days). Santosh Aditham et al. [1] applies LSTM recurrent neural networks for prediction the memory usage as a part to prevent data attacks. Also, Lingxue Zhu et al. [50] were motivated by recent Long Short Term Memory networks and successfully apply it to large-scale time series anomaly detection at Uber. In contrast to aforementioned related work, this paper for the first time combines checkpointing with time series. To reduce cost, a rich literature focuses on the optimization of resource allocation and cost-effectiveness for a cluster of cloud instances. In [22], authors proposed to reconfigure the constituent instances serving the same workload with lower cost. More recently, contracts-based resource sharing model [44] was proposed to save the cost. Some works [18, 43] focused on minimizing the cost of spot instances; Furthermore, some works [26] showed that it is even possible to achieve both the high reliability of on-demand instances and the low cost of spot instances on AWS using the proposed scheduling techniques. To the best of our knowledge, this paper is the first work focusing on cost reduction for memory-intensive applications using virtual memory technology.

Besides, much work has been focused on the modeling and scheduling part in resource management. In [45], an automatic resource allocation model for scaling virtual machine was developed; a similar work [16] on automatic leasing virtual machines was published in the same year. In particular, in [46] authors proposed an autonomic and elastic resource scheduling framework was proposed; a scheduling algorithm based on Lyapunov optimization was proposed in [35]. It should be noted that those techniques for better resource allocation are also applicable to other metrics in addition to cost, such as energy consumption that is one of the most important research topics, such as [6]. While the primary goal of this paper is to investigate new approaches to reduce the monetary cost for memory-intensive applications, the reduced cost also implies reduced energy consumption as a co-product.

In more sophisticated scenarios such as both Infrastructure-as-a-Service (IaaS) and Software-as-a-Service (SaaS) being offered, a two-stage optimization model was developed in [41]; for hybrid clouds (i.e., applications deployed to both an on-premises cluster and a public cloud), various techniques [13, 24] were proposed to minimize the overall cost; for cloud federation (i.e., inter-cloud), various techniques [5, 12, 25] were developed to automate the resource selection and configuration. Although in this paper we assume the underlying cloud infrastructure comes from a single cloud vendor (i.e., AWS), there is nothing technical to prevent users from applying the proposed approach to multiple, heterogeneous clouds.

Many other domains (e.g., high-performance computing (HPC) [7, 17, 27], databases [39, 40], networking [42, 47], big data systems [20, 48]) are switching from conventional cluster computing to cloud computing and minimizing the cost is also actively researched. Of note, *working set* has been actively studied in HPC (e.g., [8, 30]), which refers to the overall size of the program along with the initial and intermediate data to be held in (virtual) memory. The techniques proposed by this paper, although mainly targeted on and evaluated on public clouds, have the potential to be extended to apply to other domains as well.

3 Preliminaries

3.1 Swap space in Unix-like systems

The swap space in Unix-like systems is a portion of disk space that is reserved to be used as an extended memory that is addressable by the memory management module in the operating system (OS) kernel. Swap is sometimes called virtual memory, in the sense that it is not really manipulating data on the physical memory. The virtual memory in the context of swap should be differentiated from the OS-level virtual memory, which refers to the logically continuous memory pages that are mapped to possibly disjoint pages on the physical memory. In the remainder of this paper, we will use swap and virtual memory interchangeably.

Because swap requires readdressing of the memory space, one limitation of swap is tent to be longer running time. It can be said the execution speed of the same process is same, but never exceed. In fact, in our prior work [28], we have showed that accurately predicting applications' memory usage could be highly time consuming.

3.2 Checkpointing

Checkpointing is widely used in many computing paradigms, namely, high-performance computing, cloud computing, cluster computing, and so forth. In cloud computing, checkpointing has been extensively studied [10, 21]. The key idea is to periodically dump the memory status to persistent media, usually a local hard disk. One classical problem in checkpointing is how frequently it should be applied (assuming the checkpointing is applied at equal time intervals): if it is applied too frequently, the checkpointing itself (causing many memory access operations) could be unacceptable in terms of performance; if it is rarely applied, say only once, then up to 50% of work would get lost if the application or system crashed at the very last moment before the checkpoint. Therefore, the optimal checkpointing frequency usually lands in somewhere between the aforementioned two extreme cases.

A rich literature (e.g., [10]) investigated on how to estimate the optimal checkpoint intervals. In this paper, we will take the following approach to estimate the optimal checkpointing intervals (literature took a similar approach with only some varieties on assumptions and corner cases). Let T indicate the total runtime of the application, n indicate the total number of checkpoints (with equal time intervals), and t indicate the time that a single checkpoint takes, and m indicate the total number of expected failures. Without loss of generality, we assume the failures occur in the middle of two adjacent checkpoints. It then follows that the total time of lost work is $\frac{T\cdot m}{2n}$. The goal is to find n so to minimize the end-to-end wall time comprised of the application's own execution time T, the summation of all checkpointing time $n \cdot t$, and the summation of all lost work $\frac{T\cdot m}{2n}$ as they have to be redone. That is, we need to find n such that

$$\arg\min_{n} F(n) = T + n \cdot t + \frac{T \cdot m}{2n} \qquad (1)$$

It follows that if we take the first-order derivative of $F(n)$ and solve the equation

$$F'(n) = \frac{d(F(n))}{dn} = t - \frac{T \cdot m}{2n^2} = 0 \qquad (2)$$

then we have $n = \sqrt{\frac{T\cdot m}{2t}}$. Because the second-order derivative of $F(n)$ is always positive, i.e.,

$$F''(n) = \frac{d(F'(n))}{dn} = \frac{T \cdot m}{n^3} > 0 \qquad (3)$$

we are guaranteed that $n = \sqrt{\frac{T\cdot m}{2t}}$ is the solution to the minimal value of $F(n)$, and the optimal checkpointing interval Δ_t is

$$\Delta_t = \frac{T}{n} = \sqrt{\frac{2 \cdot t \cdot T}{m}} \qquad (4)$$

We will use the above equation to calculate the optimal checkpoint interval as part of the baseline performance and compare it against the performance of our proposed mechanism in later sections.

4 Cost-Aware Exploitation of Virtual Memory

4.1 Overview

For applications whose memory footprint exceeds the available physical memory[4], it is possible to extend the memory usage to the swap space (i.e., virtual memory) with expected performance slowdown due to the data swap between the physical memory and the disk. In theory, the entire hard disk drive can be used as virtual memory; in Unix-like systems, this can be easily configured before the applications starts. Consequently, in theory, an application would unlikely run into out-of-memory (OOM) errors as long as we simply extended the virtual memory to the entire disk assuming the application's data can be accommodated by the disk size; but this might not be always practical because the performance overhead could be prohibitive in the real world.

As a concrete example, we show that an MRI application's performance on two different (and extreme-case) setups of virtual memory in Figure 2. The application incurs a peak memory usage of about 18 GB and completes in less than 500 seconds (red line) when the machine is equipped with enough memory (i.e., swap portion P, defined by the occupied swap size divided by occupied swap size plus occupied RAM size, is 0%). However, when we specify a combination of 1 GB physical memory and a 18 GB swap space on the machine (i.e., swap portion $\frac{18}{19} > 94\%$), the same application's total runtime exceeds 20,000 seconds. The red line is the real-time memory footprint when the application runs on a 32GB-RAM machine; the orange and blue lines record the swap and physical memory usages, respectively (on the 1GB-RAM 18GB-swap machine). However, the overall cost on the 1GB-RAM 18GB-swap machine might be lower than running the application on a 32GB-RAM machine.

4. By "physical memory", we do not exclude the memory allocation in a virtual machine; it is used in this context only to differentiate the "virtual memory" or "swap space" used in Unix-like systems.

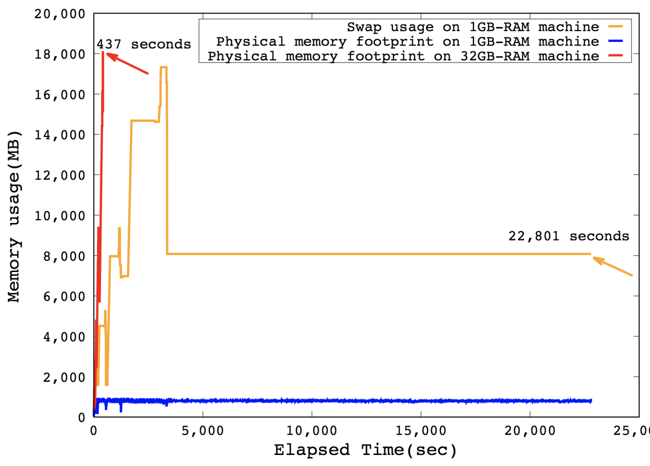


Fig. 2: Huge Overhead Introduced by Improper Use of Virtual Memory

The key question is: *if the users are willing to trade some time off to complete their jobs on the current instances (with enough swap space without OOM errors), what would the monetary benefit and time overhead look like, quantitatively?* That is, users would know better about the distribution of cost-time correlations in the parameter space between the two extreme-cases presented in Figure 2.

4.2 Assumptions

We assume the swap space would be able to accommodate the application's memory usage at all times. This assumption is easy to satisfy in the real world, as the capacity of hard disk drives is usually orders of magnitudes larger than physical memory. In addition, adding more hard disk drives is usually one of the most economic upgrades if more swap space is needed.

We also assume the swap portion is known in advance (we address the swap portion unknown scenario in Section 5). In many areas such high-performance computing and Mapreduce-like workloads, swap access-patterns including memory footprint are well studied (e.g., approaches including profiling an sample run on a subset of input data). Therefore, as long as the hardware specification of the machines is determined, it is easy to calculate the swap portion based on the physical memory capacity and the application's swap access patterns.

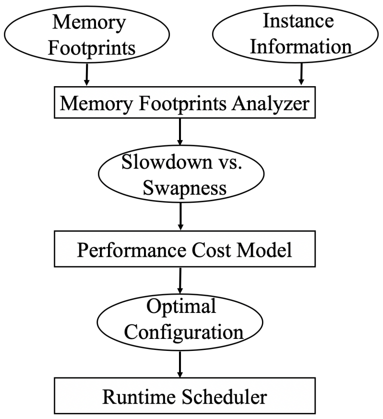


Fig. 3: Overview of Cost-Aware Exploitation of Virtual Memory Approach

The last assumption is that the pricing of different instance types is fixed. Indeed, there are cases where instance prices are versatile (e.g., AWS's spot instances), but we do not consider those scenarios in our models because they are highly dependent on the non-technical contexts such business models that are beyond the scope of this paper.

4.3 Methodology

We aim to develop models that characterize the interconnection between applications' performance and instances' swap portion (i.e., "swapness") under various instance types with different memory capacities. The overview of cost-aware exploitation of virtual memory approach is shown in Figure 3. The approach needs both application's memory footprints and the information of the instance as inputs, and outputs Slowdown vs. Swapness. The models are crafted for specific memory sizes for two reasons: First, they will achieve higher accuracy than a single global model applied to all memory capacities; Second, most cloud vendors offer a limited number of instance types regarding memory capacities.

To study the correlation between performance slowdown and swap portion in normal running conditions, we start with measuring the swap-introduced slowdown in one of the most widely used matrix operations: matrix initialization. We chose this application as the baseline benchmark because of its representative, i.e., uniform, access to the (virtual) memory. Specifically, a 17,000 $\times$ 17,000 two-dimensional matrix is initialized with random integer values, which incurs about 18 GB peak memory usage. The test bed is an AWS `t2.medium` instance with 4 GB memory. The benchmark application is decomposed into various phases according to their memory usages, while the end-to-end execution times of each phases are recorded with the corresponding swap portion. For data-intensive applications, recording all the information on swap portions at a fine granularity is both space- and time-consuming. To address that, we apply cumulative density functions (CDF) to compress the numbers and sizes of those records rather than storing all data points in their raw format.

As shown in Figure 4, the benchmark exhibits a strong exponential curve between the slowdown and the swapness. Although this is the trend exhibited with 4GB physical memory, similar trends are indeed observed with other

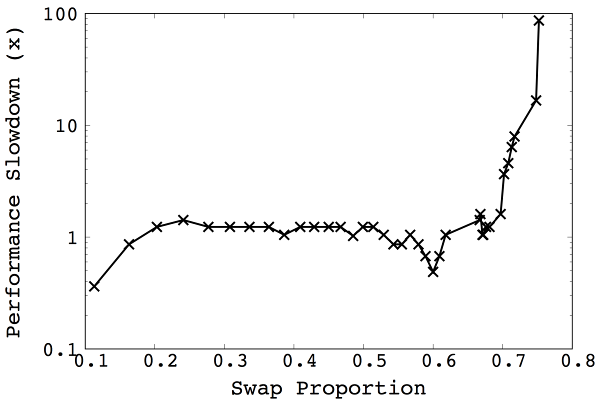


Fig. 4: Slowdown vs. Swapness with 4GB-RAM Physical Memory

memory capacities (we will discuss more when evaluating the system in Section 6). In theory, a higher proportion of swap space should imply a super-linear increase in swap access cost if the replacement policy is least-recently-used (LRU) [49]. As a result, the skeleton of the model we chose to fit is exponential in the following form:

$$S = \alpha \cdot e^{\beta \cdot x} + \theta, \tag{5}$$

where S is the performance slowdown, e is Euler's number, x is the swapness, and (α, β, θ) are coefficients to be determined during the model fitting.

The following illustrates how we fit the model for performance slowdown and swap portion. We segment the benchmark's execution into pieces with different swap portions at runtime. For each swap proportion (i.e., swapness), we maintain a bucket to store the number of pieces falling into the bucket to save space (i.e., a cumulative density function, CDF). We applied binary searches on each of the coefficients in the model and determine them when the overall error is minimal. The resultant model is:

$$f(x) = (3.09E - 13) \cdot e^{44.21x} + 0.35 \tag{6}$$

Because of the fluctuations exhibited by Figure 4, we also provide confidence intervals (maximum and minimum) as follows:

$$f_{max}(x) = (1.16E - 10) \cdot e^{36.34x} + 13.33 \tag{7}$$

$$f_{min}(x) = (4.48E - 11) \cdot e^{34.89x} + 0.075 \tag{8}$$

The f_{max} models a subset of the benchmark data points landing on the top-left edge, while the f_{min} models the subset of the benchmark data points landing on the bottom-right edge. We will be using the above models to predict more real-world applications and report its accuracy and more importantly, the correlation between the overall monetary cost and the performance, in Section 6.

5 Just-in-Time Application Migration

5.1 Overview

There are various reasons why users decide not to use swap space for out-of-memory errors. For instance, the applications might be highly memory access-intensive and using swap, even by a very small portion, would slow the application down by orders of magnitude. As another example, the application's memory footprint and memory access patterns are not well studied, then the techniques proposed in Section 4 are not applicable.

The question now becomes: *Without introducing swap, how could we reduce the overall monetary cost in face of memory depletion?* Obviously, the risk of encountering memory errors would become the lowest if users started with the most powerful (and, expensive) instance to run the applications; doing so implies, however, the highest chance of underutilization of the memory resources and consequently incurs unnecessary monetary cost. One heuristic approach would be starting with the cheapest instances and then migrate the application to a larger instance when memory is depleted, which means additional performance overhead from periodical checkpointing and relaunching virtual machines.

Recall the significant overhead of checkpointing as shown in Figure 1. Ideally, the migration should occur when only absolutely necessary as it may introduce performance degradation such as starting and warm up the memory of the new instance, transferring the data from old instance to new instance, i.e., at the point just before the memory errors out, what we called *just-in-time application migration* in the following discussion. The terminology is inspired by the well-known compiler technique "just-in-time compilation," meaning that the source code is only compiled at runtime rather than prior to the execution—one of the unique features in functional programming languages like Lisp. It is worth to point out that here we focus on predicting "when" to do migration instead of "how" to do migration, so any live migration is compliment arty to our approach proposed in this section.

5.2 Assumptions

We assume the application is migrated between different instance types without physical data movements. This is not true in conventional clusters but very common in cloud vendors, for example in AWS the current instance can be shut down with saved status (i.e., snapshot) and then get restarted with larger memory capacity allocated along with other possible upgrades. Microsoft Azure and IBM BlueMix provide similar functionalities. Note that, in conventional cluster computing, a failed node's data are first checkpointed from memory to disk and then transferred to a healthy node, usually incurring a higher overhead.

Another assumption is that swap is completely excluded. That is, the swap portion in the remainder of this section is 0%. Indeed, there is nothing preventing us to combine the models in Section 4 and what we will discuss in this section, meaning that swapness can be between 1% and 100% in the real world. In practice, both approaches work with SWAP. It is also worth to note that our experimental results show that even with SWAP on, OMM can happen if the memory demand is much larger than the RAM capacity. However, this section will focus on only the techniques of just-in-time application migration, which is isolated from others so we know how, and by how much, just-in-time application migration can facilitate the cost reduction.

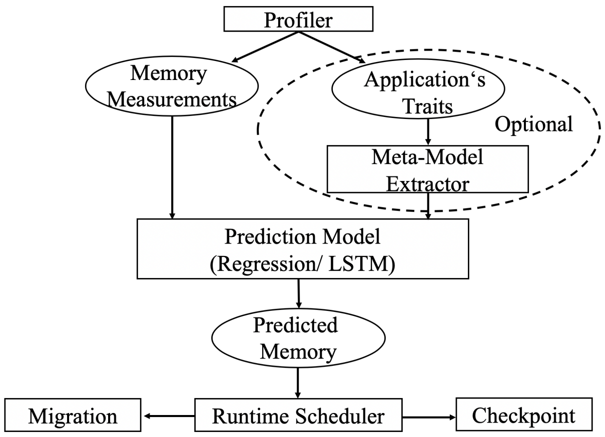


Fig. 5: Overview of Just-In-Time Application Migration Approach

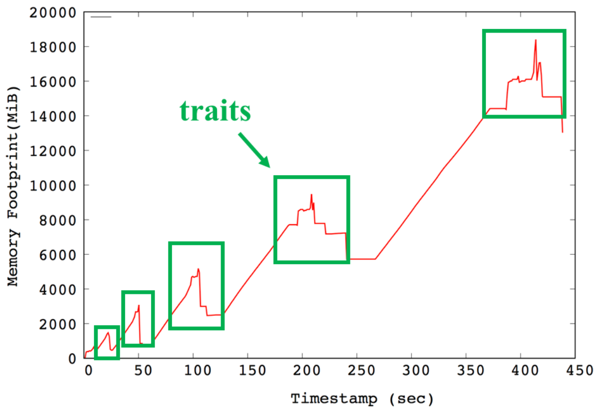


Fig. 6: Memory Footprint of MRI

5.3 Methodology

Since we plan not to apply periodical checkpointing, the key challenge is how to accurately predict when the memory will error out. Our approach considers both statistical models and the hint extracted from the application. The overview of the approach is shown in Figure 5. Specifically, our approach takes multiple fitting models (e.g., polynomial, exponential) and looks back different numbers of data points (i.e., history depending on certain decay functions). Conventional models simply look back a certain number of data points, apply regression to achieve the least aggregate errors, and calculate a future data point; In contrast, our approach considers those fitting models as inputs and takes the application's own traits extracted from profiling a small portion of execution as another input, both of which constitute a higher-level of model that we call *Meta-Model* (MM). That is, the meta-model we propose is not only fit by the existing data but also correlated to the application's sample runs. Figure 6 shows the memory footprint of MRI over time when running on a 32G memory instance. There are 5 similar shaped repeated patterns highlighted with boxes and we call the repeated pattern as trait. As the the application running, the trait becomes bigger in terms of memory footprint and time span. Therefore, by modeling the size of the traits, we can predict future traits' memory footprint and time span. Assume we use f_1 and f_2 representing the scale of applications traits, f_3 representing the position of the traits, we can derive a quadratic model as following:

$$MM(x) = f_1 \cdot x^2 + f_2 \cdot x + f_3 \tag{9}$$

where the vector $\vec{F} = [f_1, f_2, f_3]$ represents the relation between the coefficient and the adjusted impact to the application. In the simplest form, $\vec{F}$ can be a linear transformation according to the swap access patterns we can observe from profiling a subset of sampled data points. In other words, our model extends the conventional fitting approaches by generalizing those constant coefficients into additional function that characterizes the application's own information.

After calculating the prediction values of multiple meta-models, we compute the probability of memory crash. The checkpoint and migration is triggered when 90% of RAM capacity is reached to offset the prediction error. If the probability falls below a threshold, the application will be paused and ready to be migrated to another instance (with larger memory capacity). Also, we adjust the aggressiveness of prediction algorithm based on the immediate history RAM usage and thus can avoid the accumulation of the errors. Specifically, if the predicted memory value is smaller than the measured value, we will make our prediction algorithm more aggressive. An alternative approach is to run a vote from all the participating meta-models and the majority wins (either continue with the current instance or migrate to a larger one).

If the system decides (or, predicts) that a memory crash is to occur, the application will be migrated to the least expensive instance type that has larger memory capacity than the current running instance. The reasoning is that in most public cloud vendors, the memory capacity roughly follows an exponential pattern (i.e., 2 GB, 4 GB, 8 GB, and so forth). Our protocol is conservative and hopes that doubling the memory capacity could satisfy the application's memory requirement. A more aggressive protocol is possible, for example instead of increasing 2× memory size we can multiple 3, 4, or even larger factors. In practice, doing so would imply missing some intermediate instance types. This paper will only discuss the scenarios where all the instances are strictly ordered by the memory capacity. Finding out the optimal factor of multiplying memory capacity is an interesting question and might be addressed in our future work.

5.3.1 Regression Models

A time series is a sequence S of historic measurements y_t of an observable variable y at timestamp t. For time series data, the current value is highly dependent on its predecessor and even its grand predecessors in a recursive fashion with possibly decay functions. A regression model can be constructed to depict the pattern and hopefully forecast the new values in the future. The easiest approach to model this is, arguably, based on minimizing ordinary least squares. Suppose that we have time series data available on two variables, say y and x. Assuming the timestamp t is indicated by $t = 1, \cdots, T$ where y_t and x_t are updated simultaneously. We can set a regression model between y and t as following:

$$y_t = \beta_0 + \beta_1 \cdot x_t + \varepsilon \tag{10}$$

where ε indicates a random noise term with zero mean and normal distribution.

The conventional static model assumes that when a change is made to x at time t, then y is immediately affected as follows:

$$\Delta y_t = \beta \cdot \Delta x_t \tag{11}$$

where $\Delta\varepsilon_t = 0$. Thanks to this regression models, we can estimate the correlation between y and x and then predict the value difference at two adjacent timestamps. In this paper we extend the conventional approach from a single-step prediction to a *multi-step prediction*. In essence, the proposed multi-step regression repeatedly applies the single-step model by filling out the pseudo-real values with predicted values in a sliding window of various sizes. In the following discussion and experimental evaluation, we apply both three and five steps to the multi-step regression model. More specifically, in the case of predicting the future memory footprint for the next 3 or 5 seconds, we first predict the value at the first second, then this prediction is used as an (pseudo-)observation input to predict the value at the second second, and so on.

5.3.2 LSTM Models

In this chapter, we also apply Long-Short Term Memory (LSTM) models for the prediction of memory footprints. The key challenge is how to accurately forecast the footprint in multiple steps. In essence, the challenge of one-step prediction based on LSTM is further complicated by the uncertainty incurred by these steps due to the accumulation of errors.

We modify the conventional LSTM model [15] to predict the future values by reapplying single-step LSTMs. Specifically, we first predict x_{t+1} using the previous m values such as x_t, x_{t-1}, $\cdots$, x_{t-m+1}, and then predict x_{t+2} based on its previous m values, which include the predicted value at x_{t+1}. The procedure is repeated until the last value ~~x_{t+n},~~ is completed. Because it is sufficient to construct a single-step model to make the prediction, we predict the value at each step using a possibly different model. Given an initial data set like $\{x_1, x_2, ..., x_m\}$ as input, we first create h training sets, each of which has the same input but different output. The size of the prediction window h is case-dependent and highly influenced by the nature of the application.

5.3.3 Full Elasticity

Inspired by the conventional swap space offered by many Unix-like systems, we build a fully elastic strategy for running application on expanded or shrunk memory. The elastic scaling strategy not only migrate the memory-intensive applications to a larger instance when needed but also downgrade the instance when some of the taken memory is released, on the premise that an application can run continuously without OOM errors. In doing so, we are able to dump memory status to the disk (to avoid out of memory problem) and to reduce the monetary cost when the memory is no more needed. When building such models, we set a threshold to decide whether to downgrade the instance which is not a trivial task but an optimal problem because checkpoint's overhead has to be taken into account. The rule of thumb is, though, if the cost on checkpointing memory states is more than the saved cost by switching to a smaller instance, then the total monetary cost will be higher than simply running on the existing large instance (we will demonstrate this in the evaluation section).

5.4 Hybrid Approach

The just-in-time application migration approach proposed in this section can be combined with the cost-ware exploitation of virtual memory approach proposed in Section 4 to form a hybrid approach. Specifically, when the memory footprint of an application is unknown a priori, the application can start with the just-in-time application migration as it does not impose any assumption on memory footprint. After running a while, if a repeating pattern is detected in history memory footprints, it can switch to the cost-ware exploitation of virtual memory approach for better cost-effectiveness.

6 Evaluation

6.1 Experimental Setup

All experiments are carried out on AWS [2]. The instances for various memory capacities and prices are listed in Table 1. We chose reserved instances for evaluation such that little performance interference is expected. Nonetheless, we do not impose assumptions for specific instance. If users prefer spot instances, the proposed work can be integrated with other spot instance management work, e.g., the SPOTon [37]. That is, our work is complementary to existing spot-instance framework.

We implement our models using Python and Shell scripts. The system prototype can be directly deployed to any Linux-like operating system and serves as a middleware for upper-level frameworks, including but not limited to big data systems such as Spark and Myria. The reason why we isolate the memory-scaling subsystem from these big data systems is that OOM errors are common across different big data systems [28]; therefore, a loosely-couple middleware would contribute to the largest possible spectrum of big data systems. The source code of this work is available at the project website:https://www.cse.unr.edu/hpdic/proj/cme. We ran each experiment for 10 times and took the average values for the reported monetary cost and execution time.

TABLE 1: AWS instances used for evaluation

Instance Name	Memory Capacity (GB)	Price (US$ per Hour)
t2.small	2	0.023
t2.medium	4	0.0464
t2.large	8	0.0928
t2.xlarge	16	0.1856
t2.2xlarge	32	0.3712

6.2 Cost-aware Exploitation of Virtual Memory

The goal of this section is two-fold: a) demonstrating the accuracy of the proposed performance model with various real-world applications; b) reporting quantitative monetary benefit (and the compromise made on performance) when different portions of swap space are taken into account. We evaluated the model with three real-world applications: 1)

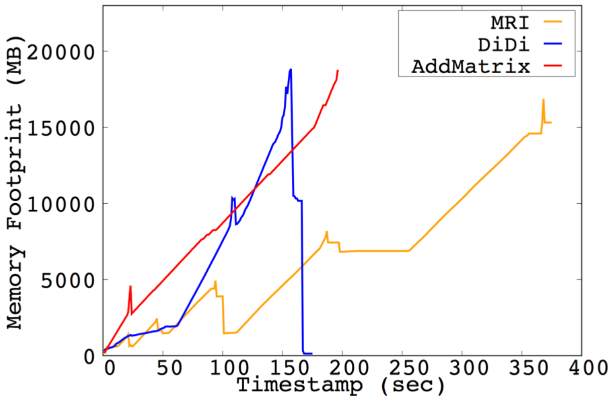


Fig. 7: Memory Needs of Applications

MRI application [28], a scientific image processing application using image data to make inferences about the brain. And the measurements of the app are used to estimate large-scale brain connectivity. Since measurements are repeated 288 times on each person, OOM errors are so common in MRI that we can easily use it as an particular example to solve OOM errors; 2) Didi application [11], an open-source data mining application on taxi's location data collected by Didi Inc (the collaborator of Uber in China). The research focuses on the ability to identify stationary and moving objects from a moving car using a range of data points and attempts to advance the development of self-driving cars; 3) an open-source implementation of adding multidimensional matrices AddMatrix [32] from the popular Numpy library. The memory needs of these three applications is shown as Figure 7. All the experiments were carried out with 4GB-RAM instances on AWS (i.e., `t2.medium`) unless otherwise noted.

For this method, a repeating cycle of memory footprint is needed to build the Slowdown vs. Swapness model (as shown in Figure 4). Figure 9 shows the slowdown of the MRI application when swap portion is between 0.1 and 0.8. We do see some fluctuations in the real slowdown, which is expected as arbitrary (e.g., not LRU) memory-access patterns occur in this application. However, the trend still follows an exponential curve in the big picture. More importantly, all data points fall into the ranges (i.e., the gray shadow) of the model, indicating a high accuracy of the proposed model. To quantify that, we apply the Pearson correlation coefficient (PCC) defined as:

$$\texttt{PCC} = \frac{\Sigma_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\Sigma_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\Sigma_{i=1}^{n}(y_i - \overline{y})^2}} \tag{12}$$

where n indicates the total number of data points, x_i and y_i indicate the real data and modeled data, and $\overline{x}$ and $\overline{y}$ indicate the mean of each data series. The correlation of the real data points and our model is extremely strong, as the PCC turns out to be 0.994.

Similarly, Figure 8 shows the slowdown of the data mining application and the swap portion is between 0.175 and 0.8. The portion range is a little different from the first application because this one is more memory-hungry (or, more memory-intensive). And because of the memory-hungry nature of this application, we observe even more serious slowdown than the MRI application. But again, most

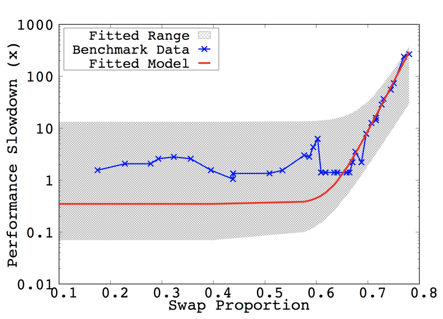


Fig. 8: Swap model for Didi [11]

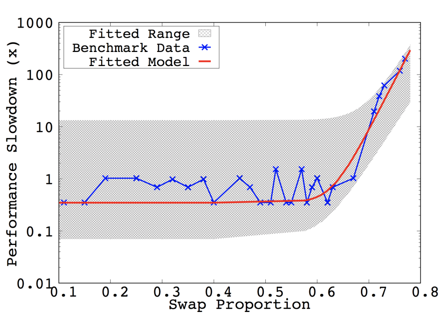


Fig. 9: Swap model for MRI [28]

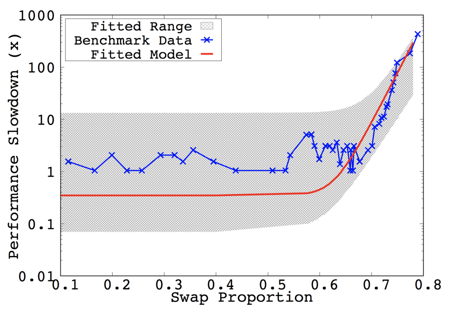


Fig. 10: Swap Model for AddMatrix [32]

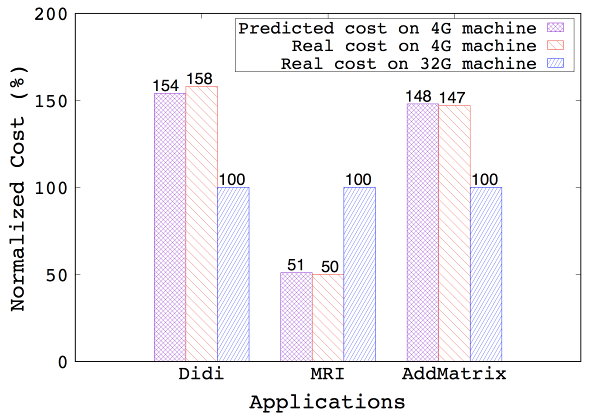


Fig. 11: Monetary Cost Normalized to the 32G-RAM Machine

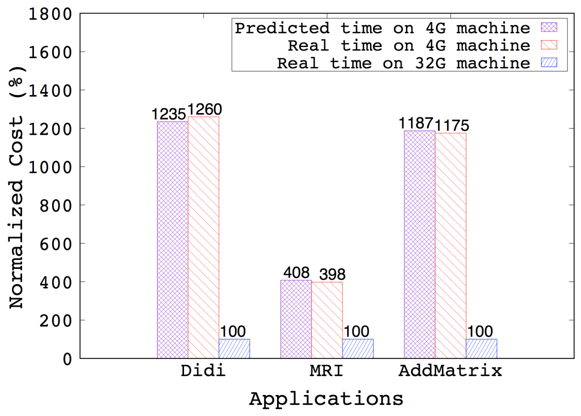


Fig. 12: Execution Time Normalized to the 32G-RAM Machine

of the real data points fall into the prediction intervals (i.e., the gray shadow) and the PCC of this application is also extremely high: 0.988.

Lastly, Figure 10 shows the slowdown of the matrix adding application and the swap portion is between 0.1 and 0.8. This application exhibits a similar pattern as MRI; the PCC coefficient is also extremely high: 0.991.

Figure 11 compares the cost predicted by the proposed model and the real cost on the 4GB-RAM instance, both of which normalized to the cost on the 32GB-RAM instance that provides enough physical memory for both applications (the peak of real memory footprint is about 18 GB). Since in all cases the application is not migrated between instances, the cost is the same in terms of both execution time and monetary cost. We can see that the cost predicted by our model is highly accurate: The error rates are: Didi 4%, MRI 1%, and AddMatrix 3%. The figure also shows that using swap does not guarantee reduced cost: the cost of the Didi application using swap is increased by more than 158%, for MRI the cost is reduced by 50%, and the cost for AddMatrix is increased by 47%. The root cause of these results is that the Didi application is highly memory-intensive, meaning that introducing swap on a memory-restrained instance will likely incur much more running time outweighing the benefit of the cheaper per-hour price. The findings, in fact, prove the effectiveness of our model: our model can tell, in a very high accuracy, that whether applying swap on a memory-constrained and inexpensive instance would result in higher or lower total cost.

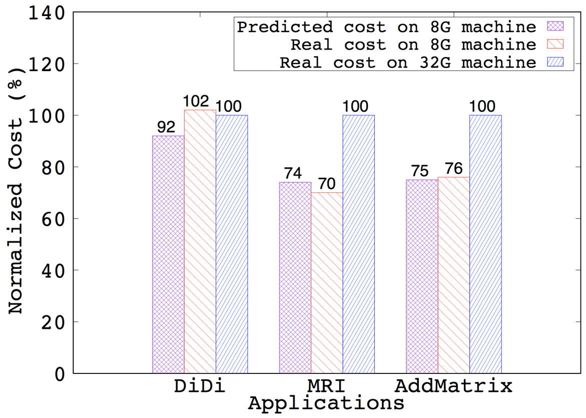


Fig. 13: Monetary Cost on an 8G-RAM Instance (normalized to the 32G-RAM machine)

We also report the execution time for all three applications in Figure 12. We observe an order of magnitude higher time cost for both Didi and AddMatrix, which is partially attributed to their swap access intensiveness between swap and physical memory and therefore cause the overall monetary cost exceeding the baseline case. What is more interesting, however, is the MRI application. The MRI results make a strong case for trading off performance for cost: by compromising 4X running time the overall cost can be reduced to half. This is exactly the point of the first contribution of this paper—allowing users to make compromise between time and cost.

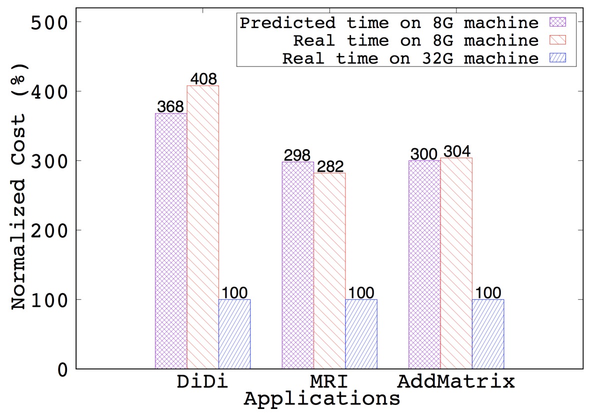


Fig. 14: Execution Time on a 8G-RAM Instance (normalized to the 32G-RAM machine)

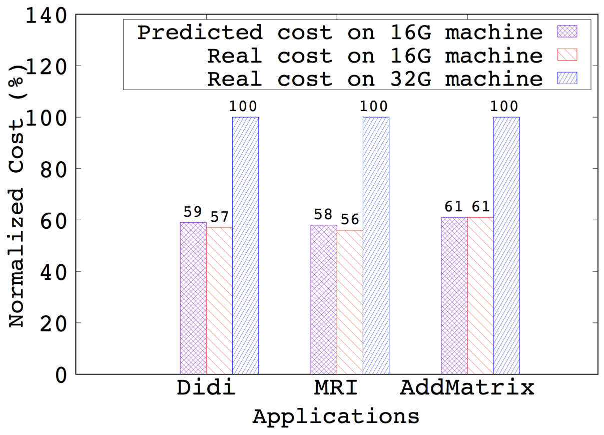


Fig. 15: Monetary Cost on an 16G-RAM Instance (normalized to the 32G-RAM machine)

Figure 13 compares the cost predicted by the proposed model and the real cost on the 8GB-RAM instance (i.e., `t2.large`), both of which normalized to the cost on the 32GB-RAM instance that provides enough physical memory for both applications (the peak of real memory footprint is about 18 GB). We observe similarly high accuracy in this experiment as in Figure 11. Specifically, the largest error occurs for the Didi application but still under 10% ($\frac{102-92}{102} = 9.8\%$).

Figure 14 compares the execution time predicted by the proposed model and the ground-truth on the 8GB-RAM instance, both of which normalized to the cost on the 32GB-RAM instance that provides enough physical memory for both applications (the peak of real memory footprint is about 18 GB).

Figure 15 compares the cost predicted by the proposed model and the real cost on the 16GB-RAM instance, both of which normalized to the cost on the 32GB-RAM instance that provides enough physical memory for both applications (the peak of real memory footprint is about 18 GB). Again, we can see that the cost predicted by our model is highly accurate: The error rates are: Didi 3.5%, MRI 3.5%, and AddMatrix 0%. The figure shows that a lot of cost can be saved when the instance provides almost the same amount of memory as the application: the cost of the Didi application using swap is reduced by more than 43%, for MRI the cost is reduced by 44%, and the cost for AddMatrix is reduced by 39%. Once again, the accuracy is high for all applications.

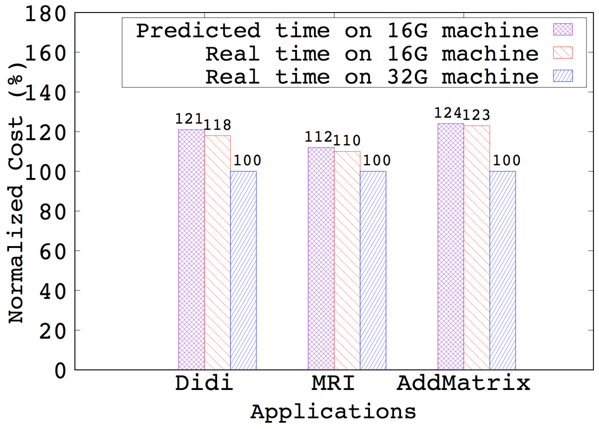


Fig. 16: Execution Time on a 16G-RAM Instance (normalized to the 32G-RAM machine)

Figure 16 compares the execution time predicted by the proposed model and the ground-truth on the 16GB-RAM instance, both of which normalized to the cost on the 32GB-RAM instance that provides enough physical memory for both applications (the peak of real memory footprint is about 18 GB. We can find that when virtual memory is used very little, the running time does not increase dramatically).

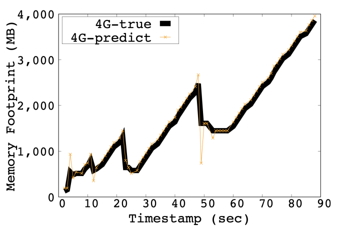
Note that the total monetary cost equals to the execution time times the cost rate of the instance. Therefore, even though higher memory capacity instance can speed up the execution time by reducing the SWAP proportion, the increased cost rate may not always overcome the increased cost rate of the more expensive instance, thus the overall cost can be higher, vice versa. E.g., in Figure 9, execute MRI using 32G memory instance has higher cost compared to instance with 4G memory.

To summarize, this section demonstrates that our models can predict both time and monetary costs with extremely high accuracy on multiple instance types. Such a low sensitivity on instance types is high desired, as this property would facilitate a wide spectrum of usability of the proposed models.

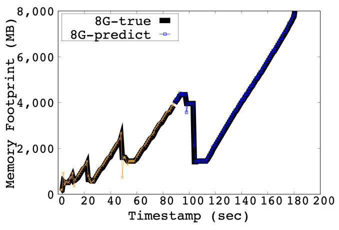
6.3 Just-in-Time Application Migration

This section answers the following questions using real-world applications: a) illustrating the real-time performance for applications continuously deployed to a series of cloud instances in an increasing order of memory capacities (application migrates to another instance when the current one's memory is to be depleted); b) reporting the quantitative monetary benefit when applying the proposed techniques of predicting memory footprint and elevating memory capacity.

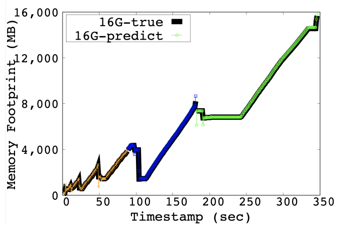
Figure 17 shows the MRI application's memory footprint over its entire course using our prediction-elevation approach. To facilitate the description, we define stage as periods that are separated by checkpoints. Having 4 stages means there are 3 checkpoints that separates the entire running duration into 4 periods. The lifespan of the application comprises four stages on four instance types: 4GB-, 8GB-,



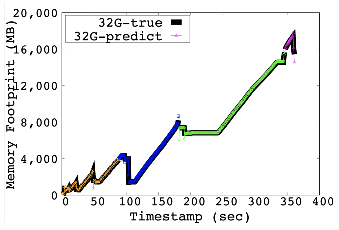
(a) Stage 1 (4GB-RAM)



(b) Stage 2 (8GB-RAM)

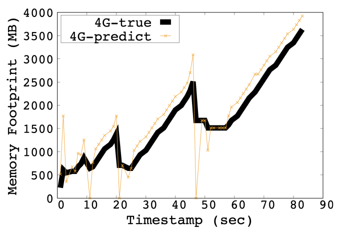


(c) Stage 3 (16GB-RAM)

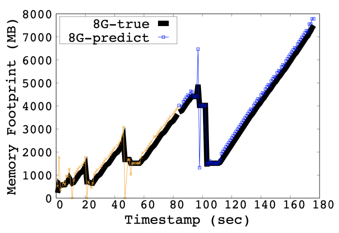


(d) Stage 4 (32GB-RAM)

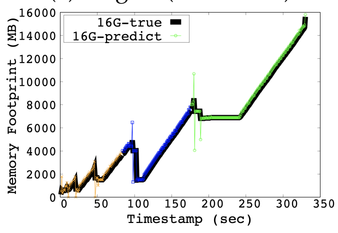
Fig. 17: Memory Elevation after Migrating Applications, MAPE = 2.97%



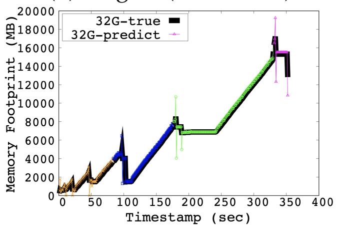
(a) Stage 1 (4GB-RAM)



(b) Stage 2 (8GB-RAM)

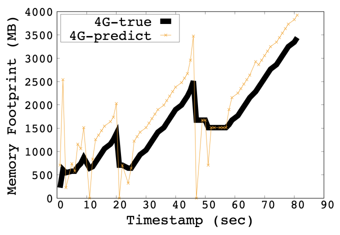


(c) Stage 3 (16GB-RAM)

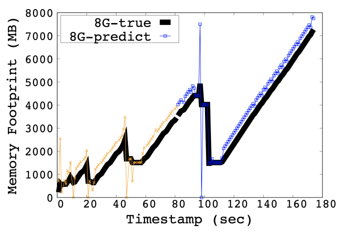


(d) Stage 4 (32GB-RAM)

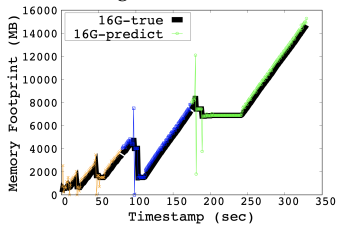
Fig. 18: Memory Footprint Predicted by Regression with Step = 3, MAPE = 8.13%



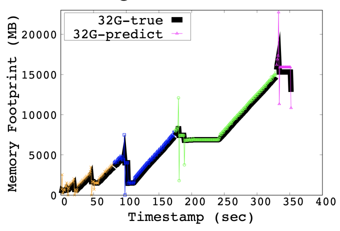
(a) Stage 1 (4GB-RAM)



(b) Stage 2 (8GB-RAM)

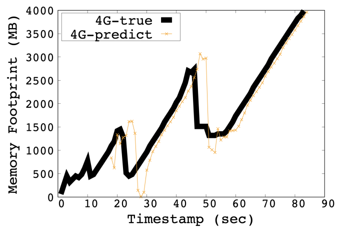


(c) Stage 3 (16GB-RAM)

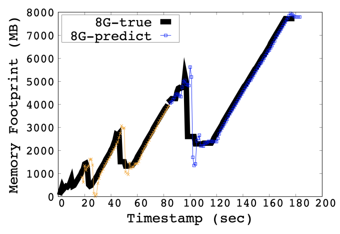


(d) Stage 4 (32GB-RAM)

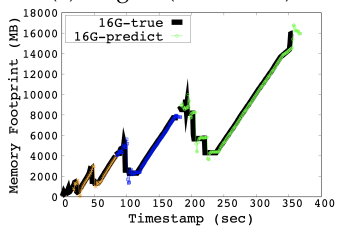
Fig. 19: Memory Footprint Predicted by Regression with Step = 5, MAPE = 13.02%



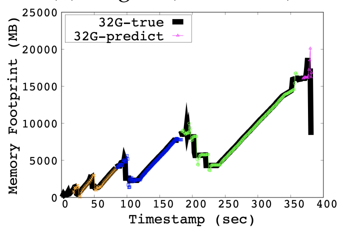
(a) Stage 1 (4GB-RAM)



(b) Stage 2 (8GB-RAM)



(c) Stage 3 (16GB-RAM)



(d) Stage 4 (32GB-RAM)

Fig. 20: Memory Footprint Using LSTM with Step = 3, MAPE = 11.52%

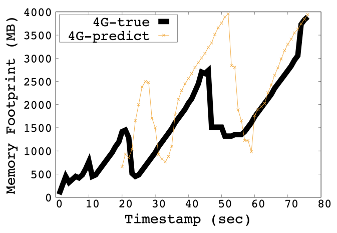
16GB-, and 32GB-RAM instances, as shown in sub-figures. For each stage, we plot the true values and the predicted values using the approach we discussed in Section 5. As we can see, our predicted values are highly accurate for most of time [5], except that at around 50 seconds the predicted value is noticeable lower than the true value. The reason for that is because the application just completed a memory-intensive iteration and the system is busy with memory recycling (i.e., garbage collection). We will further fine-tune our model by considering such corner scenarios in our future work.

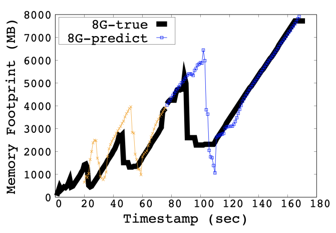
Figure 18 illustrates how the memory footprint is predicted using the regression model with step size = 3. We observe slightly larger errors compared to the baseline predictor (Figure 17, particularly on smaller instances such as Figure 18a and Figure 18b. Therefore, time series models are more sensitive on smaller instances.

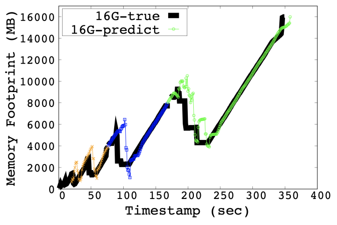
Figure 19 illustrates how the memory footprint is predicted using the regression model with step size = 5. This experiment reconfirms the conclusion drawn from the last, Figure 18: time-series models are sensitive on (at least) two factors: instance capacity and step size. As for instance capacity, more noticeable prediction errors are found in Figure 19a (4GB-RAM instance) and Figure 19b (8GB-RAM instance) compared to Figure 19c (16GB-RAM instance) and Figure 19d (32GB-RAM instance). As for step size, the gaps in Figure 19a are more significant than those in Figure 18a, and gaps in Figure 19b are more significant than those in Figure 18b.

Figure 20 illustrates how the memory footprint is predicted using the LSTM model with step size = 3. It suffers the same high sensitivity as regression models on smaller instances, e.g., 4GB-RAM (Figure 20a), 8GB-RAM (Figure 20b). One interesting observation is that the sensitivity of LSTM models on prediction error now comes as an offset rather than a vertical error in regression models. To see this, take

5. It is worth to mention that the prediction results of Stage 1 looks like have larger error than Stage 2-4 is mainly because the y-axis scale is only from 0 to 4,000 MB, much smaller than Stage 2-4 (e.g., in Stage 4, y-axis is from 0 to 20,000 MB), which makes the error more pronounced.


(a) Stage 1 (4GB-RAM)


(b) Stage 2 (8GB-RAM)


(c) Stage 3 (16GB-RAM)

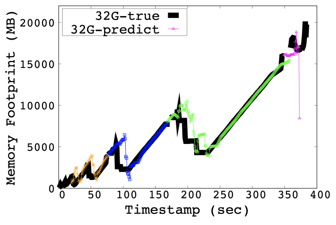

(d) Stage 4 (32GB-RAM)

Fig. 21: Memory Footprint Using LSTM with Step = 5, MAPE = 18.17%

Figure 20a for example, both the prediction and true-value plots are in the same pattern but with a small offset, whereas the counterpart in Figure 18a exhibits the gaps in the vertical direction. The root cause of this phenomena is that our LSTM model "optimistically" takes the estimated values as true values in future prediction, resulting in a lag (or, a time-wise offset) in prediction.

Figure 21 illustrates how the memory footprint is predicted using the LSTM model with step size = 5. This model is by far the most inaccurate model due to the large step size. In particular, the predicted values at small instance of 4GB-RAM (Figure 21a are significantly off the true values. In addition, even at larger instances (e.g., Figure 21c) there are even noticeable errors, which are almost negligible in other models.

For this method, the models need a warm up phase for online adaptive prediction. The duration of warm up phase is usually very short, for example, for using LSTM model on MRI application, it needs about 17 seconds to warm up (shown as Figure 20a and Figure 21a). It is worth to note that based the above experimental evaluation, the regression based model is slightly more accurate in prediction, but it involves manual efforts of choosing the regression function. LSTM based model on the other hand is slightly less accurate but is fully data-driven and automatic, thus more convenient. Also, regression is a simple model while LSTM is a sophisticated model. Usually a regression model can be trained well with little data, but the limitation is that it could not capture very complex behaviors. LSTM has potential to capture more complex behaviors but also needs more training data and fine-tuning. Here LSTM performs worse as the pattern is relatively simple and the training data is relatively limited. The MAPE (Mean Absolute Percentage Error) for both models is shown in Table 2. Our approach supports both prediction models and users can choose the more suitable one based on their needs.

Figure 22 illustrates the overall costs at various stages normalized to the baseline cost (we did not show the 32GB-RAM comparison since they all incur the same cost). Here, we evaluate both the monetary cost and running time cost

TABLE 2: MAPE(%) of Two Proposed Algorithms

Step / Algo.	1	3	5	7	9	15
Regression	2.97	8.13	13.02	18.33	24.44	34.15
LSTM	4.13	11.52	18.17	25.46	34.22	42.31

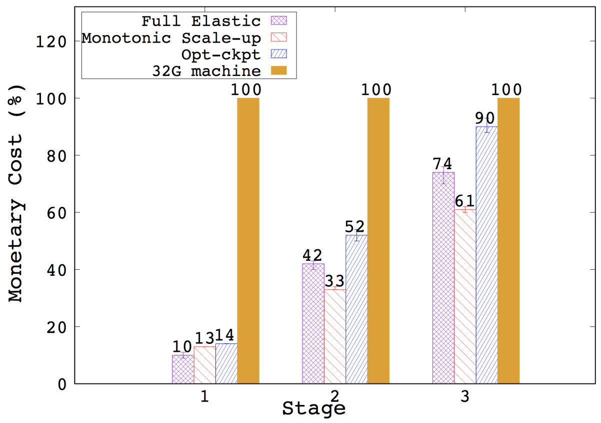


Fig. 22: Cost Comparison (normalized to the 32GB-RAM machine)

under four cases: 1) Full Elastic represents the case that the application does not only migrate to a larger instance when needed but also downgrade the instance when memory usage decreases. It is worth noting that Full Elastic Method is not always preferred such as the memory usages are growing most of the time and durations of descent are very short. But if the memory usage often keeps low for a relatively long duration, the full elastic method will be more cost-effective; 2) Monotonic Scale-up represents the case that the application only migrates from smaller instance to larger instance; 3) Opt-ckpt represents the case of using optimal checkpointing method; and 4) 32G machine represents the case that instance has large enough memory at the beigning. We can see that for all types of instances, the proposed techniques (either the Full Elastic or the Monotonic Scale-up approach) incur the least cost—significantly cheaper than both the checkpoint approach and the baseline with all physical memory allocated. In Stage 1, Full Elastic costs only 10% of the baseline approach at 4GB-RAM instances; In Stage 2 and State 3, Monotonic Scale-up saves 33% and 60% on 8GB-RAM and 16GB-RAM instances, respectively. The optimal-checkpointing approach is also cheaper than the baseline but more expensive than our proposed approaches: 14% (vs. 10% and 13%), 53% (vs. 42% and 33%), and 90% (vs. 74% and 61%) on the above instances. Overall, Full Elastic scaling takes 42% of the baseline cost and 81% of the optimal checkpointing, while Monotonic Scale-up takes only 35% of the baseline cost and 66% of the optimal-checkpointing approach.

Indeed, the saved cost does require more running time, and Figure 23 reports the time overhead on each type of instances normalized to the baseline running time. Again, we did not report the 32GB-RAM case since the numbers would look exactly the same. We can see that for each instance type, the proposed Monotonic Scale-up does not incur as significant overhead as the optimal-checkpoint does. For the first three stages, the proposed approach takes 3%, 34%, and 22% more time than the baseline while the optimal-

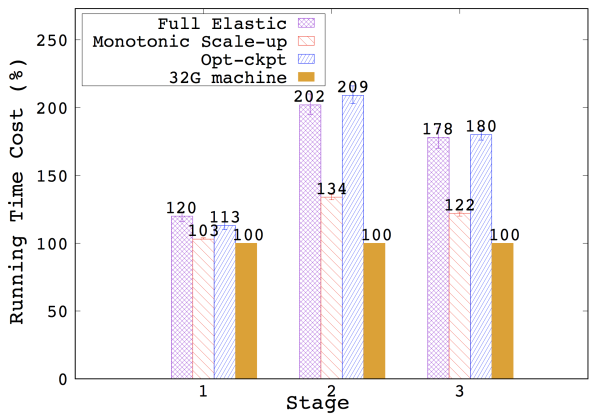


Fig. 23: Time Comparison (normalized to the 32GB-RAM machine)

checkpointing approach takes much more: 13%, 110%, and 80%. Taking all together, the time overhead of Monotonic is only 22% but saves 65% cost comparing to the baseline. The overhead introduced by Full Elastic scaling is, indeed, higher than Monotonic Scale-up, on par with the optimal checkpointing. This implies that Full Elastic is not a top pick when the application is time-sensitive; and yet, Full Elastic does provide the highest flexibility that might be a must-have property for some specific workloads.

7 Conclusion and Future Work

This paper presents two techniques to help deploy big data applications with dynamic and intensive memory footprint on cloud-based big data systems with low monetary cost. The first approach assumes the users are well aware of the application's swap access patterns such as uniform access, and the proposed performance-cost model can accurately predict how, and by how much, virtual memory size would slow down the application and consequently, impact the overall monetary cost. The second approach removes the assumption of *a priori* memory access patterns by proposing a lightweight memory usage prediction methodology. The key idea is to eliminate the periodical checkpointing and migrate the application only when the predicted memory usage exceeds the physical allocation of the big data systems based on dynamic meta-models adjusted by the application's own traits. Taking both techniques together, this work covers a wide spectrum of big data applications regarding the trade-off between performance and cost using both virtual and physical memory scaling approaches.

The future direction of this work is how to further improve the data-locality at the swap space. In this paper we assume the replacement policy in swap is least-recently-used (LRU), which is true in almost all operating systems but may not well aligned with applications' swap access patterns. This is exactly why in experiments we did observe some fluctuations when modeling the time-swapness correlation (Fig. 4). We believe if we can hack into the swap implementation, the performance of memory-intensive applications using the proposed techniques would be further improved. One possible solution is to implement a user-level swap and integrate it to the approaches proposed in this work.

References

[1] S. Aditham, N. Ranganathan, and S. Katkoori. Taming performance degradation of containers in the case of extreme memory overcommitment. In *2017 31st IEEE International Parallel and Distributed Processing Symposium*, pages 1259–1267, May 2017.

[2] Amazon EC2. http://aws.amazon.com/ec2, Accessed March 6, 2015.

[3] Apache Spark. http://spark.apache.org/, Accessed December 23, 2015.

[4] Apache Spark. https://spark.apache.org/docs/latest/tuning.html#determining-memory-consumption, Accessed December 23, 2015.

[5] U. Bellur, A. Malani, and N. C. Narendra. Cost optimization in multi-site multi-cloud environments with multiple pricing schemes. In *2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*, pages 689–696, June 2014.

[6] A. Beloglazov and R. Buyya. Energy efficient allocation of virtual machines in cloud data centers. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 577–578, May 2010.

[7] R. Chard, K. Chard, K. Bubendorfer, L. Lacinski, R. Madduri, and I. Foster. Cost-aware elastic cloud provisioning for scientific workloads. In *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, pages 971–974, 2015.

[8] J. Chen, J. Liu, P. Roth, and Y. Chen. Using working set reorganization to manage storage systems with hard and solid state disks. In *Proceedings of the 2014 43rd International Conference on Parallel Processing Workshops (ICPP)*, 2014.

[9] CRIU. https://criu.org, Accessed February 10, 2018.

[10] S. Di, Y. Robert, F. Vivien, D. Kondo, C. L. Wang, and F. Cappello. Optimization of cloud task processing with checkpoint-restart mechanism. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2013.

[11] Didi Data Mining Competition. http://research.xiaojukeji.com/competition/detail.action?competitionId=DiTech2016, Accessed February 12, 2018.

[12] D. J. Dubois, G. Valetto, D. Lucia, and E. D. Nitto. Mycocloud: Elasticity through self-organized service placement in decentralized clouds. In *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, pages 629–636, June 2015.

[13] M. R. H. Farahabady, Y. C. Lee, and A. Y. Zomaya. Pareto-optimal cloud bursting. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 25(10):2670–2682, Oct 2014.

[14] Hadoop. http://hadoop.apache.org/, Accessed September 5, 2014.

[15] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[16] P. Hoenisch, S. Schulte, S. Dustdar, and S. Venugopal. Self-adaptive resource allocation for elastic process execution. In *2013 IEEE Sixth International Conference on Cloud Computing (CLOUD)*, pages 220–227, June 2013.

[17] H. Huang, L. Wang, B. C. Tak, L. Wang, and C. Tang. Cap3: A cloud auto-provisioning framework for parallel processing using on-demand and spot instances. In *2013 IEEE Sixth International Conference on Cloud Computing (CLOUD)*, pages 228–235, June 2013.

[18] B. Javadi, R. K. Thulasiram, and R. Buyya. Characterizing spot price dynamics in public cloud environments. *Future Gener. Comput. Syst.*, 29(4):988–999, June 2013.

[19] X. Ji, C. Wang, N. El-Sayed, X. Ma, Y. Kim, S. S. Vazhkudai, W. Xue, and D. Sanchez. Understanding object-level memory access patterns across the spectrum. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 25:1–25:12, 2017.

[20] L. Jiang, K. Wang, and D. Zhao. Davram: Distributed virtual memory in user space. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Accepted, 2018.

[21] S. Khatua and N. Mukherjee. Application-centric resource provisioning for amazon ec2 spot instances. In *European Conference on Parallel Processing (Euro-Par)*, pages 267–278, 2013.

[22] P. Kokkinos, T. A. Varvarigou, A. Kretsis, P. Soumplis, and E. A. Varvarigos. Cost and utilization optimization of amazon ec2 instances. In *2013 IEEE Sixth International Conference on Cloud Computing (CLOUD)*, pages 518–525, June 2013.

[23] Large Synoptic Survey Telescope. https://www.lsst.org/, Accessed June 29, 2017.

[24] Y. C. Lee and B. Lian. Cloud bursting scheduler for cost efficiency.

In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 774–777, June 2017.
[25] A. F. Leite, V. Alves, G. N. Rodrigues, C. Tadonki, C. Eisenbeis, and A. C. M. A. d. Melo. Automating resource selection and configuration in inter-clouds through a software product line method. In *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, pages 726–733, June 2015.
[26] L. M. Leslie, Y. C. Lee, P. Lu, and A. Y. Zomaya. Exploiting performance and cost diversity in the cloud. In *2013 IEEE Sixth International Conference on Cloud Computing (CLOUD)*, pages 107–114, June 2013.
[27] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2011.
[28] P. Mehta, S. Dorkenwald, D. Zhao, T. Kaftan, A. Cheung, M. Balazinska, A. Rokem, A. Connolly, J. Vanderplas, and Y. AlSayyad. Comparative evaluation of big-data systems on scientific image analytics workloads. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*, volume 10, pages 1226–1237, Aug. 2017.
[29] MPICH. http://www.mpich.org, Accessed December 10, 2014.
[30] R. Murphy, A. Rodrigues, P. Kogge, and K. Underwood. The implications of working set analysis on supercomputing memory hierarchy design. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS)*, 2005.
[31] Myria. http://myria.cs.washington.edu, Accessed July 18, 2016.
[32] Numpy: adding matrix. https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.add.html, Accessed March 2, 2018.
[33] OpenMP. http://openmp.org/wp/, Accessed March 24, 2015.
[34] SciDB. https://paradigm4.atlassian.net/wiki/display/ESD/SciDB+Documentation, Accessed July 25, 2016.
[35] S. Shi, C. Wu, and Z. Li. Cost-minimizing online vm purchasing for application service providers with arbitrary demands. In *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, pages 146–154, 2015.
[36] S. Spinner, N. Herbst, S. Kounev, X. Zhu, L. Lu, M. Uysal, and R. Griffith. Proactive memory scaling of virtualized applications. In *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, pages 277–284, June 2015.
[37] Supreeth, T. Subramanya, P. Guo, D. Sharma, P. Irwin, and Shenoy. Spoton: a batch computing service for the spot market. In *Proceedings of the sixth ACM symposium on cloud computing*, pages 329–341, 2015.
[38] TensorFlow. https://www.tensorflow.org/, Accessed July 26, 2016.
[39] P. Upadhyaya, M. Balazinska, and D. Suciu. How to price shared optimizations in the cloud. *International Conference on Very Large Data Bases (VLDB)*, 5(6):562–573, Feb. 2012.
[40] P. Upadhyaya, M. Balazinska, and D. Suciu. Price-optimal querying with data apis. *International Conference on Very Large Data Bases (VLDB)*, 9(14):1695–1706, Oct. 2016.
[41] V. D. Valerio, V. Cardellini, and F. L. Presti. Optimal pricing and service provisioning strategies in cloud systems: A stackelberg game approach. In *2013 IEEE Sixth International Conference on Cloud Computing (CLOUD)*, pages 115–122, June 2013.
[42] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 1–9, March 2010.
[43] P. Wang, Y. Qi, D. Hui, L. Rao, and X. Liu. Present or future: Optimal pricing for spot instances. In *2013 IEEE 33rd International Conference on Distributed Computing Systems (ICDCS)*, pages 410–419, July 2013.
[44] J. Xu and B. Palanisamy. Cost-aware resource management for federated clouds using resource sharing contracts. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 238–245, June 2017.
[45] L. Yazdanov and C. Fetzer. Vscaler: Autonomic virtual machine scaling. In *2013 IEEE Sixth International Conference on Cloud Computing (CLOUD)*, pages 212–219, June 2013.
[46] Z. Yin, H. Chen, J. Sun, and F. Hu. Eaers: An enhanced version of autonomic and elastic resource scheduling framework for cloud applications. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 512–519, June 2017.
[47] L. Zhang, Z. Li, and C. Wu. Dynamic resource provisioning in cloud computing: A randomized auction approach. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 433–441, April 2014.
[48] D. Zhao, N. Liu, D. Kimpe, R. Ross, X.-H. Sun, and I. Raicu. Towards exploring data-intensive scientific applications at extreme scales through systems and simulations. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 27(6):1824–1837, June 2016.
[49] D. Zhao, K. Qiao, and I. Raicu. Hycache+: Towards scalable high-performance caching middleware for parallel file systems. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 267–276, 2014.
[50] L. Zhu and N. Laptev. Deep and confident prediction for time series at uber. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 103–110, November 2017.