Toward Cost-effective Memory Scaling in Clouds: Symbiosis of Virtual and Physical Memory

Xinying Wang*, Cong Xu†, Ke Wang‡, Feng Yan*, Dongfang Zhao*

*University of Nevada, Reno †HP Labs, Palo Alto, CA ‡Microsoft Azure, Redmond, WA

***Abstract*—When deploying memory-intensive applications to public clouds, one important yet challenging problem is selecting 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 cloud: 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 with errors ranging from 1% to 4% and the proposed just-in-time migration approach reduces the monetary cost by up to 66%.**

I. 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 [7], Spark [2], TensorFlow [24], Myria [17], and SciDB [21], 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 [20], MPI [16]), our prior work [15] showed that new challenges are emerging from those big data systems such as application migration, memory management, among many others.

As a concrete example, in [15] we evaluated an astronomical application, namely Large Synoptic Survey Telescope (LSST [12]), by migrating it to a distributed database Myria [17] on the Amazon Web Services (AWS) public cloud [1]. In LSST, the telescope periodically takes photos of the sky (i.e., sky surveys at multiple visits) and detects the sources and tracks of celestial objects. As Figure 1 shows, the converted code worked fine for up to 8 visits of sky surveys but then started to experience out-of-memory (OOM) errors for 12 or more visits because the allocated data to each node exceeds the physical memory capacity. We had to spend considerable

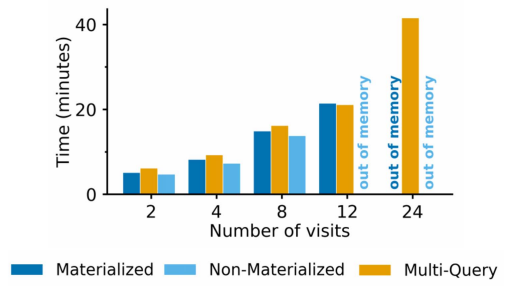


Fig. 1: The LSST astronomical application encounters numerous out-of-memory errors when deployed to AWS. [15]

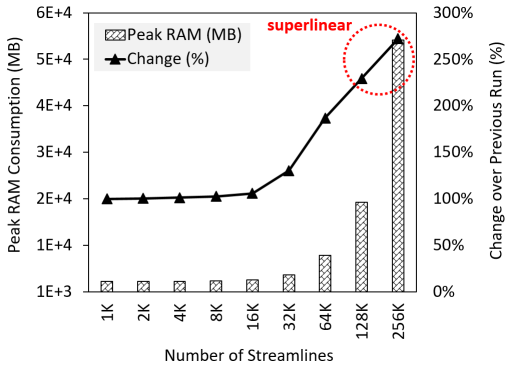


Fig. 2: The memory footprint of an MRI application increases superlinearly to the input size. [9]

time to completely rewrite the original application using multi-query to avoid memory errors.

The OOM error 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 application with finer granularity, reconfiguring the underlying system, redeploying the application, and recomputing many results. Despite of 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 as shown in Figure 2 reported in one of our more recent works [9]. The number of streamlines represents the input

2159-6190/18/$31.00 ©2018 IEEE
DOI 10.1109/CLOUD.2018.00012

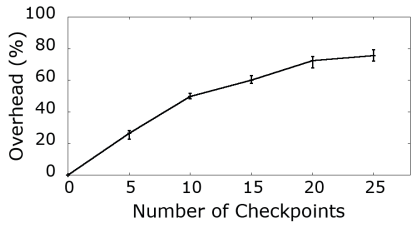


Fig. 3: Checkpointing overhead.

size for an Magnetic Resonance Imaging (MRI) application detailed in [15].

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 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, which often 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 3 shows the overhead when different numbers of checkpointing are applied to a real-world application used in [15] using a popular checkpointing tool CRIU [4]. We observe a fast-growing trend of performance overhead incurred by checkpointing, which motivates us to develop a checkpoint-free mechanism for memory-intensive applications—the second objective of this work: scaling the physical memory using checkpointing only one time.

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 that 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 first technique assumes the I/O patterns (e.g., uniform) are well-studied, which is true for many applications, for example, in areas like high-performance computing [8] and scientific applications [32]. 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. The second technique drops the assumption of the I/O patterns and introduces 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%.

[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 I/O bandwidth.

II. Related Work

Memory management in cloud computing recently draws plenty of research interests in the community. In particular, Spinner et al. [23] 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). In [18], authors conducted a quantitative study of the impact of overcommitment of physical memory to dockers. In contrast to aforementioned related work, this paper for the first time proposes to leverage virtual memory to achieve lower monetary cost by trading off a fraction of running time.

A rich literature focuses on the optimization of resource allocation and cost-effectiveness for a cluster of cloud instances. In [11], authors proposed to reconfigure the constituent instances serving the same workload with lower cost. More recently, contracts-based resource sharing model [27] was proposed to save the cost. To the best of our knowledge, this paper is the first work focusing on cost reduction for memory-intensive applications using virtual memory technology.

Much work has been focused on the modeling and scheduling part in resource management. In [28], an automatic resource allocation model for scaling virtual machine was developed. More recently, in [29] authors proposed an autonomic and elastic resource scheduling framework was proposed; an scheduling algorithm based on Lyapunov optimization was proposed in [22]. 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 [26]; for hybrid clouds (i.e., applications deployed to both an on-premises cluster and a public cloud), various techniques [13]

were proposed to minimize the overall cost; for cloud federation (i.e., inter-cloud), various techniques [14] 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 [3], databases [25], networking [30], big data systems [9, 31]) are switching from conventional cluster computing to cloud computing and minimizing the cost is also actively researched. 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.

III. Preliminaries

A. 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.

Because swap requires readdressing of the memory space, one limitation of swap is that it cannot be dynamically updated at runtime. In our prior work [15], we showed that accurately predicting applications' memory usage could be highly time consuming. So in the remainder of this paper, we will use swap and virtual memory interchangeably.

B. Checkpointing

In cloud computing, checkpointing has been extensively studied [5, 10]. 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 I/O 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.

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

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

It follows that $n = \sqrt{\frac{T \cdot m}{2t}}$. We will use n 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.

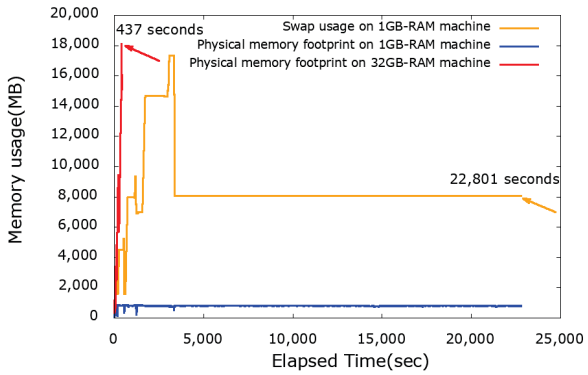


Fig. 4: Huge overhead introduced by improper use of virtual memory

IV. Cost-aware Exploitation of Virtual Memory

A. Overview

For applications whose memory footprint exceeds the available physical memory[2], 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. Unfortunately, the change of virtual memory capacity would not take effect during the runtime of the applications, meaning that a dynamic allocation of virtual memory is out of the question. 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 4. The application incurs a peak memory usage of about 18 GB and completes in less than

[2]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.

500 seconds (red line) when the machine is equipped with enough memory (i.e., swap portion 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.

The key question is: *if the users are willing to trade some time off to complete their jobs on the current instances (with data swapped between memory and disk), 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 4.

B. 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. In many areas such high-performance computing and MapReduce-like workloads, I/O-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 I/O patterns.

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.

C. 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 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, 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

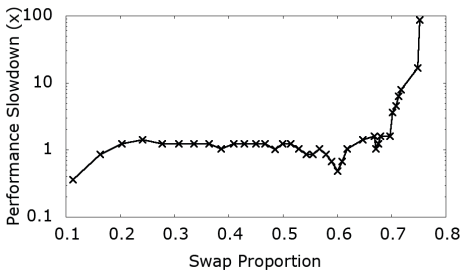


Fig. 5: Slowdown vs. swapness with 4GB-RAM physical memory

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 5, 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 memory capacities (we will discuss more when evaluating the system in Section VI). In theory, a higher proportion of swap space should imply a super-linear increase in I/O cost if the replacement policy is least-recently-used (LRU) [32]. As a result, the skeleton of the model we chose to fit is exponential in the following form:

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

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:

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

Because of the fluctuations exhibited by Figure 5, we also provide confidence intervals (maximum and minimum) as follows:

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

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

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 VI.

V. Just-in-Time Application Migration

A. 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 I/O-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 I/O patterns are not well studied, then the techniques proposed in Section IV 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 3. Ideally, the migration should occur when only absolutely necessary, 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.

B. 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 IV and what we will discuss in this section, meaning that swapness can be between 1% and 100% in the real world. 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.

C. 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. 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 a input and take 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. Taking a quadratic model for example, the model M in its original form

$$M(x) = \alpha \cdot x^2 + \beta \cdot x + \gamma$$

becomes

$$MM(x) = f_1 \cdot x^2 + f_2 \cdot x + f_3$$

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 I/O 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. If the probability falls below a threshold, the application will be paused and gets ready to be migrated to another instance (with larger memory capacity). 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. 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.

VI. Evaluation

A. Experimental Setup

All experiments are carried out on AWS [1]. The instances for various memory capacities and prices are listed in Table I. We implement our models using Python and Shell scripts. The source code is available at the project website: https://www.cse.unr.edu/hpdic/proj/cme.

TABLE I: AWS instances used for evaluation

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

We evaluated the proposed techniques with three real-world applications. The first application is an MRI image processing scientific application from [15]. The second application is a data mining application on taxi's location data from Didi Inc described at [6]. The third application is a common numerical application on adding dense matrices extracted from the popular Python library Numpy [19].

B. 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. All the experiments were carried out with 4GB-RAM instances on AWS (i.e., `t2.medium`).

Figure 6 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:

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

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 7 shows the slowdown of the data mining application and the swap portion is between 0.55 and 0.8. The portion range is different from the first application because this one is more memory-hungry (or, more memory-intensive); that is why the swap space starts to kick in directly from the 0.55—more than half of the memory is "virtual". Because of the memory-hungry nature of this application, we observe even more fluctuations than the MRI application. But again, most 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 8 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 10 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.

We also report the execution time for all three applications in Figure 9. We observe an order of magnitude higher time cost for both Didi and AddMatrix, which is partially attributed to their I/O 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.

C. 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.

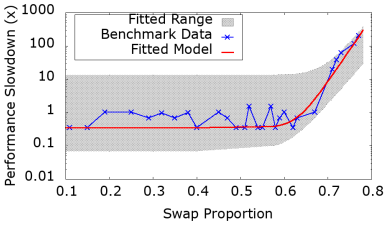


Fig. 6: Swap model for MRI [15]

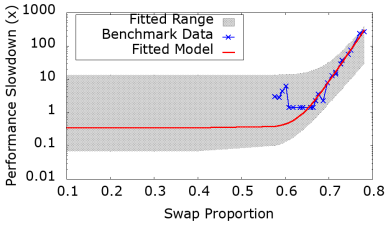


Fig. 7: Swap model for Didi [6]

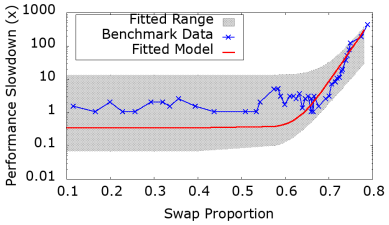


Fig. 8: Swap model for AddMatrix [19]

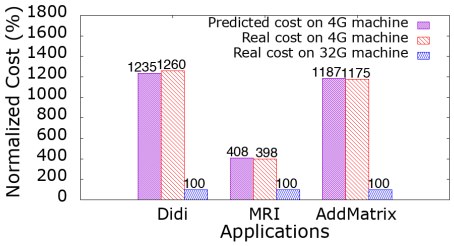


Fig. 9: Execution time normalized to the 32G-RAM machine

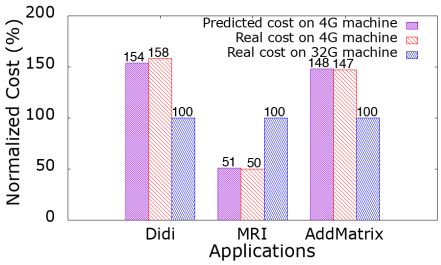
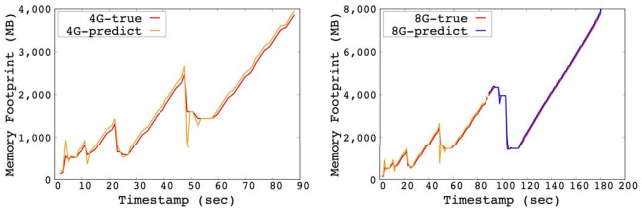
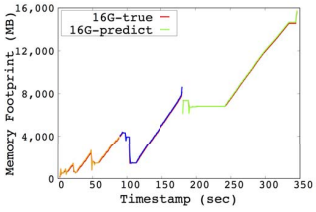


Fig. 10: Monetary cost normalized to the 32G-RAM machine

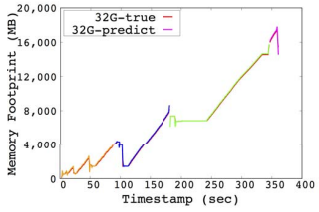


(a) Stage 1 (4GB-RAM)

(b) Stage 2 (8GB-RAM)



(c) Stage 3 (16GB-RAM)



(d) Stage 4 (32GB-RAM)

Fig. 11: Memory elevation after migrating applications

Figure 11 shows the MRI application's memory footprint over its entire course using our prediction-elevation approach. The lifespan of the application comprises four stages on

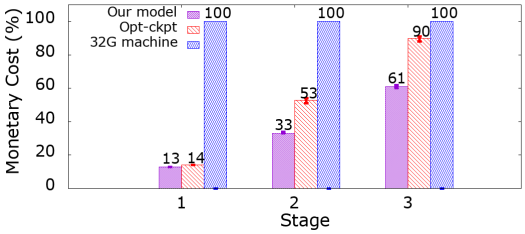


Fig. 12: Cost comparison (normalized to the 32GB-RAM machine)

four instance types: 4GB-, 8GB-, 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 V. As we can see, our predicted values are highly accurate for most of time, 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 12 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). We can see that for all types of instances, our proposed approach incurs the least cost—significantly cheaper than both the checkpoint approach and the baseline with all physical memory allocated. Specifically, our approach costs only 12% of the baseline approach at 4GB-RAM instances, and 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 approach: 14% (vs. 13%), 53% (vs. 33%), and 90% (vs. 61%) on the above instances. Overall, our approach 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 13 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 approach 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

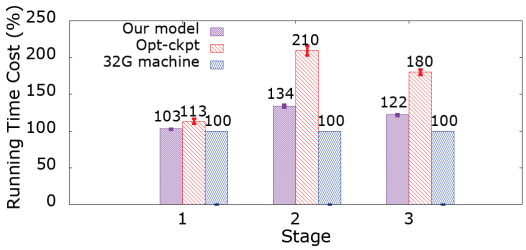


Fig. 13: Time comparison (normalized to the 32GB-RAM machine)

the baseline while the optimal-checkpointing approach takes much more: 13%, 110%, and 80%. Taking all together, the time overhead of the proposed prediction-elevation approach is only 22% but saves 65% cost comparing to the baseline.

VII. Conclusion

This paper presents two techniques to help deploy memory-intensive applications in public clouds with low monetary cost. The first approach assumes the users are well aware of the application's I/O 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* I/O 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 based on dynamic meta-models adjusted by the application's own traits. Taking both techniques together, this work covers a wide spectrum of data-intensive applications regarding the trade-off between performance and cost using both virtual and physical memory scaling approaches.

Acknowledgments

This work is in part supported by an Amazon Web Services (AWS) Research Grant, a Microsoft Azure Research Award, and a National Science Foundation (NSF) award #1756013.

References

[1] Amazon EC2. http://aws.amazon.com/ec2, Accessed March 6, 2015.
[2] Apache Spark. http://spark.apache.org/, Accessed December 23, 2015.
[3] 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.
[4] CRIU. https://criu.org, Accessed February 10, 2018.
[5] 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.
[6] Didi Data Mining Competition. http://research.xiaojukeji.com/competition/detail.action?competitionId=DiTech2016, Accessed February 12, 2018.
[7] Hadoop. http://hadoop.apache.org/, Accessed September 5, 2014.
[8] 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.
[9] 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.
[10] 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.
[11] 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.
[12] Large Synoptic Survey Telescope. https://www.lsst.org/, Accessed June 29, 2017.
[13] 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.
[14] 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.
[15] 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.
[16] MPICH. http://www.mpich.org, Accessed December 10, 2014.
[17] Myria. http://myria.cs.washington.edu, Accessed July 18, 2016.
[18] R. Nakazawa, K. Ogata, S. Seelam, and T. Onodera. Taming performance degradation of containers in the case of extreme memory overcommitment. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 196–204, June 2017.
[19] Numpy: adding matrix. https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.add.html, Accessed March 2, 2018.
[20] OpenMP. http://openmp.org/wp/, Accessed March 24, 2015.
[21] SciDB. https://paradigm4.atlassian.net/wiki/display/ESD/SciDB+Documentation, Accessed July 25, 2016.
[22] 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.
[23] 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.
[24] TensorFlow. https://www.tensorflow.org/, Accessed July 26, 2016.
[25] 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.
[26] 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.
[27] 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.
[28] 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.
[29] 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.
[30] 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.
[31] 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.
[32] 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.