FirePlace: Placing FireCracker Virtual Machines with Hindsight Imitation

Bharathan Balaji **Christopher Kakovitch** **Balakrishnan Narayanaswamy**
Amazon
Seattle, WA
`{bhabalaj, chriskak, muralibn}@amazon.com`

Abstract

Virtual machines (VM) form the foundation of modern cloud computing as they help logically abstract per-user compute from shared physical infrastructure. Users of these services require VMs of varying sizes and configurations, which the provider places on a set of physical machines (PMs). VMs on the same physical PM share memory and CPU resources so a bad packing directly impacts the quality of user experience. We consider the placement of FireCracker VMs (a form of Micro-VMs or μVMs) - lightweight VMs that are typically used for short lived tasks. Our objective is to place each VM as it arrives, so that the peak to average ratio of resource usage across PMs is minimized. Placement is challenging as we need to consider resource use in multiple dimensions, such as CPU and memory, and because resource use changes over time. Past approaches to similar problems have suggested that one could forecast VM resource use for placement. We see that in production traffic from Amazon Web Services (AWS), μVM resource use is spiky and short lived, and that forecasting algorithms are not useful. We evaluate Reinforcement Learning (RL) approaches for this task, but find that off-the-shelf RL algorithms are not always performant. We present a forecasting free algorithm, called FirePlace, that learns the placement decision using a variant of hindsight optimization, which we call hindsight imitation. We evaluate our approach using a production traffic trace of μVM usage from AWS Lambda. FirePlace improves upon baseline algorithms by 10% when placing 100K μVMs.

1 Introduction

Virtual Machines (VMs) [1] are an essential technology in modern computing and form the core of many cloud services. VMs are useful because they allow providers to allocate resources efficiently by fitting many VMs on a single physical machine (PM), while giving the functionality of a separate machine to the end user. VMs on the same physical PM share memory, CPU and network resources. A bad packing would impact the quality of computations and/or increase costs.

Algorithms for packing VMs efficiently into PMs have been studied for over a decade [2, 3, 4]. FireCracker VMs are a recent innovation - a form of lightweight VMs that have fast startup times and can be packed with high density, while still providing strong security [5, 6]. FireCracker VMs are an instantiation of MicroVMs, and we refer to them as μVMs for brevity. They are used to provide serverless services such as Function as a Service [7] with low overhead. We consider an online placement setting common in cloud computing, where μVMs are created and deleted based on exogenous demand, and where the objective is to place the VMs such that the total number of PMs used is minimized, while ensuring that resource use in any of the PMs does not exceed limits. A good packing directly translates to increased availability, reduced operational costs and energy savings [8].

34th Workshop on ML for Systems at NeurIPS 2020, Vancouver, Canada.

Our VM placement problem is similar to the problem of online bin packing [9, 10]. However, in our setting, not only do we need to consider multiple resource dimensions such as CPU and memory, but also *how resource use changes over time*. Since vector bin packing itself is an NP complete combinatorial problem, and APX hard for 2 or more dimensions [11], our problem in its general form is intractable. Related prior work has proposed forecasting of the resource used by each VM, followed by a well-known heuristic such as Best-Fit or genetic algorithms to decide placement [12]. In our production dataset from AWS Lambda, we find that μVM are short lived and their use is spiky, and hence, difficult to forecast. We can pack hundreds of μVMs in a physical machine (PM) as each μVM resource use is small in comparison to the PM capacity. Therefore, forecasting solutions become untenable as the error in the prediction of each VM accumulates and leads to poor placement. In particular, the best 90^{th} quantile forecast for μVM CPU use over its entire lifetime is 0. We present detailed comparisons with related literature in Apendix C.

Given the difficutly of forecasting, the best that any algorithm could realistically do is hedge placement decisions based on 'typical' CPU and memory use of a new μVM when compared with the historical CPU and memory use of the PM. This hedged decision rule could be state dependent. We formulate the problem as a Markov Decision Process (MDP), where the agent places a μVM at each time step. The MDP still suffers from the curse of dimensionality because of the large state and action spaces - in particular, the large number of PMs available at each time instant. Hence, we build on the *power of two choices* [13] to reduce the action space. We propose FirePlace, a hindsight imitation learning algorithm for μVM placement. We leverage historical data to identify placement decisions that could have been made using hindsight of future μVM CPU and memory use, similar to hindsight optimization [14]. We then cast placement as a supervised learning problem with the hindsight based decision as the label. We show that FirePlace does not require per μVM forecasting, outperforms off-the-shelf model-free deep RL algorithms, runs fast enough to deploy to a latency sensitive large scale production service, and generates a learned model that generalizes to unseen data.

2 Problem Setting

We formulate our problem by modeling the characteristics of a typical cloud system. μVMs are created by an external service based on user demand, and at each time step we receive a μVM to place. Our objective is to identify a PM in the fleet on which to place the μVM so that the Peak to Average Ratio (PAR) of resource use in the fleet is minimized while ensuring the resource use in any of the PMs does not exceed capacity *in the future* as the CPU and memory use of the μVMs change. If the total number of active VMs on a PM require more CPU or memory than the PM has available, then user experience is degraded. In practice, we observed that our PMs are not bottlenecked on resources such as network bandwidth. Therefore, we scoped the problem to only track the CPU and memory use of the PM. The placement decision needs to be made quickly ($\sim$20ms) and with reasonable throughput ($\sim$5 placements/s). Each fleet can consist of hundreds or thousands of PMs, and it is impractical to require the updated state of all PMs to make the placement decision. The algorithms we tried, including RL and baselines, did not scale well with large state and action spaces. Hence, we sample $\mathcal{K}$ PMs at a time and identify the PM to place, motivated by the power of two choices [13]. This limits our action space to size $k = |\mathcal{K}|$, rather than the total number of active PMs.

μVMs start executing user tasks sometime after they are created, consuming memory and CPU when they do so. An external service deletes the μVMs if it is idle for more than a specific period of time. A PM's resource use is simply the sum of resources consumed by the μVMs in the PM, we ignore overheads. We consider a fleet with a single type of PM with fixed CPU and memory capacity.

μVMs arrive online, and the order of arrival cannot be changed. We cannot migrate μVMs once they have been placed. μVM resource use is unknown before placement and *changes over time* as tasks are executed. The memory use of a PM increases monotonically over time until deletion, since μVMs do not release memory. The μVM uses CPU only when it is executed, thus the CPU use of the μVM is zero when it is idle (we ignore idle CPU overhead), and spikes up when it executes tasks.

Each μVM is represented by a timeseries of its resource use. Let c_t^v and m_t^v denote the CPU and memory use of μVM v at time t. Each PM consists of a collection of μVMs. Let $C_t^p = \sum_{v \in \boldsymbol{V}^p} c_t^v$ and $M_t^p = \sum_{v \in \boldsymbol{V}^p} m_t^v$ denote the CPU and memory use of PM $p \in \boldsymbol{P}$ at time t, where $\boldsymbol{V}^p$ is the set of μVMs in the PM and $\boldsymbol{P}$ is the set of PMs in the fleet. Let $\boldsymbol{c}^v = c_0^v, .., c_T^v$ and $\boldsymbol{m}^v = m_0^v, .., m_T^v$ denote the CPU and memory use timeseries of length T for μVM v. Let $\boldsymbol{C}^p = \sum_{v \in \boldsymbol{V}^i} \boldsymbol{c}^v$ and

$\boldsymbol{M}^p = \sum_{v \in \boldsymbol{V}^p} \boldsymbol{m}^v$ denote the PM CPU and memory timeseries respectively. We assume all PMs have the same same capacity for CPU and memory respectively. The placement algorithm places one μVM at a time in one of $\mathcal{K}$ PMs randomly sampled from the fleet.

We formulate the problem as an MDP, where the agent places the μVMs across an episode. We introduce $\tau \in [0, T]$ as the wall clock time at which μVM$_t$ is placed in the PM. The state includes both CPU and memory use: $s(C_t^p, M_t^p, c_t^v, m_t^v) | p \in \mathcal{K}_t, t \in [0, \tau]$. The agent takes action by picking one of $\mathcal{K}_t$ PMs. The reward function is given as:

$$R_t = \frac{\frac{1}{|\boldsymbol{P}|} \sum_{p \in \boldsymbol{P}} \max_{t \in \tau} C_t^p}{\max_{p \in \boldsymbol{P}, t \in \tau} \boldsymbol{C}^p} + \frac{\frac{1}{|\boldsymbol{P}|} \sum_{p \in \boldsymbol{P}} \max_t M_t^p}{\max_{p \in \boldsymbol{P}, t \in \tau} \boldsymbol{M}^p} - W_t \tag{1}$$

where $|\boldsymbol{P}|$ is the number of PMs in the fleet. $W_t = 1$ if the PM picked is too full to place this μVM, and $W_t = 0$ otherwise. PM resource use is represented by its maximum CPU use, as the maximum use should not exceed PM capacity. The reward function encourages the agent to increase the mean CPU use of the fleet compared to the max use (PAR), which encourages the agent to pack μVMs tightly yet maintain the same CPU use across PMs. This choice of metric represents our desire to be efficient, yet robust to e.g. single instance failures, at any time. The overall objective is to maximize the sum of rewards in an episode.

We consider a PM to be too full to fit a μVM in simulation by considering the future CPU and memory use. If the agent picks a PM that is too full, we repeatedly pick another PM from the fleet at random until we find a PM that can hold the μVM. The agent receives a penalty, since bad placements degrade performance. The episode terminates when all μVMs are placed, or if all the PMs in the fleet are full. Our formulation can be extended to heterogeneous fleets, where the PMs have different CPU and memory capacities, as we use normalized measures of CPU and memory in our metrics.

A simple baseline algorithm is to pick a PM uniformly at random, and try to place the μVM. An ideal algorithm will pick the PM in the fleet that minimizes the overall PAR *in the long term.* While we only have $|\mathcal{K}| \ll |\boldsymbol{I}|$ PMs from the fleet to choose from, prior work has shown that if we pick the best of $\mathcal{K}$ PMs, we can perform much better than random [15]. Appendix A gives an overview of our cloud service model, and Appendix B details the μVM CPU and memory use characteristics.

3 Hindsight Placement

We split the problem into two parts: forecasting and optimization [16, 17, 12]. We start with the optimization problem, assuming we have perfect forecasts available using real μVM use data from our historical data. Given this future knowledge, we can pack μVMs to greedily maximize the reward in Equation 1 at each step. We call this algorithm Hindsight based packing.

Hindsight based placement is still greedy, even though it uses perfect *hindsight* information, since we do not attempt to account for future μVM arrivals. Still, a well trained placement agent could learn to hedge against the 'typical' distribution of future arrivals, and predict future resource requirements of already placed VMs, if this is useful.

We assume we have access to the entire future timeseries of μVM use and PM use, together with the μVMs already placed in it. We pick the PM that minimizes the maximum of the L^2-norm of CPU and memory use if the μVM were placed in that PM if the μVM was placed in it:

$$B_k = \left\| \left(\max_{t \in T}(c_t^v + C_t^k), \max_{t \in T}(m_t^v + M_t^k) \right) \right\|^2 \tag{2}$$

We present a *Baseline* algorithm that simply uses the mean values to represent the μVM and PM CPU/memory use. This algorithm picks the the PM that minimizes the following:

$$B_k^{mem} = \frac{1}{T} \sum_{t=0}^{T} m_t^v + \frac{1}{T} \sum_{t=0}^{T} M_t^k \tag{3}$$

$$B_k = \|(B_k^{cpu}, B_k^{mem})\|^2 \tag{4}$$

Online bin packing is a much easier problem when all the items are of the same size [18]. Since our μVM resource use is small when compared to PM capacity, we can justify use of greedy hindsight techniques. We thus focus on the time varying nature of resource use, rather than the combinatorial optimization aspects.

4 Hindsight Imitation Learning

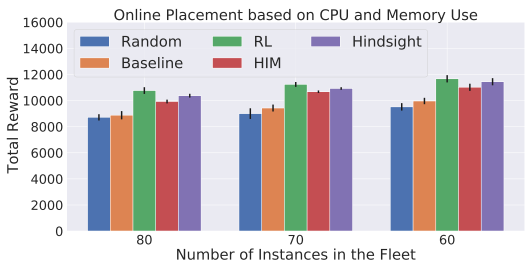
One approach we evaluate is the directly train an end-to-end placement approach using RL. As we will see in the results (Figure 1), RL is sometimes able to match or even slightly outperform the Hindsight algorithm we described in the previous section. But for many settings it is much worse. We also evaluate a form of imitation learning - a strategy that directly learns to mimic the actions of the Hindsight algorithm, using it as a teacher [19, 20]. As we have historical data from our production workloads, we can compute hindsight based decisions at each time step and create a dataset for training. Use of hindsight imitation learning circumvents the issues associated with forecasting, as we directly learn a relationship between input features and output decision that hedges placements. The idea is similar to hindsight optimization proposed by Chong et al. for tabular problems [14].

The input to the imitation model is features derived from the state and the output is the action. We found that the choice of features, model and dataset is important to avoid overfitting, even with large amounts of data, given the variance in state and action spaces. We use a random forest classifier [21] and support vector machines [22] as our model, whichever gives us the the best classification accuracy on our validation data. Simple 2 layer neural networks [23] did not perform well in our datasets. We use quantiles of PM resource use over different windows as our features. Specifically, we use the p10, p25, p50, p75, p100 and mean values for CPU use, and just the p100 and mean values for memory use as memory use is monotonic till deletion. We split the μVMs from historical production data to two partitions, use one partition to train our agent and the other to test. We simulate the online μVM placement in a fleet with our train partition, and use the features computed from state as well as the Hindsight algorithm actions to create our training dataset for imitation learning. We refer to the model learned as *Hindsight Imitation Model (HIM)*.

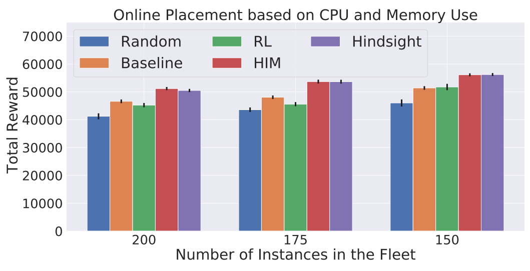
5 Results

We use multiple datasets from a single region: one with 20,000 μVMs and another with 100,000 μVMs respectively. We refer to them as the 20K and 100K datasets. We set $|\mathcal{K}| = 2$, i.e. the algorithm picks from 2 randomly sampled PMs in the fleet and places the μVM on one of these. The performance of the algorithm increases slightly with increase in $|\mathcal{K}|$ [13], but $|\mathcal{K}| = 2$ gives us a large improvement over random. We use PMs with 4 CPU cores and 16GB memory in the 20K dataset and use 8 CPU cores and 64GB memory in the 100K dataset. As the number of PMs in the fleet decreases, the placement becomes harder until all PMs very quickly become too full to accommodate μVMs. We vary the number of PMs in our experiments to show the impact of fleet size on the performance of placement algorithms. We run each experiment 10 times, and report mean rewards with error bars.

We assign alternate μVMs to two partitions - this makes the training data representative, but also well separated from test data. We use the train partition to create our dataset with the Hindsight algorithm. We use 75% of the dataset for training and 25% for validation.



(a)



(b)

Figure 1: μVM packing based on CPU and memory use in (a) and (b) for 20K and 100K datasets respectively.

Having already defined the problem as an MDP, reinforcement learning (RL) algorithms are a natural fit for the problem. In our case, the states are quantiles of CPU and memory over different windows, for each of the $|\mathcal{K}|$ PMs and the specific μVM we are placing in that time step. We have $|\mathcal{K}|$ actions corresponding to which of PMs we place the μVM. We evaluate off-the-shelf deep RL algorithms implemented in the Ray RLlib library [24]. We use the PPO [25] algorithm, and experimented with a number of hyper-paramaters and discuss results of the best settings we could find. We report the exact hyperparameters used in Appendix D and training curves are shown in Appendix E. To find a good neural network structure we also evaluated using the structure that performed the best at imitating the Hindsight algorithm. The results in Figure 1 show that the best RL algorithm sometimes performed as well as or even slightly better than the Hindsight algorithm, but usually performed worse. Figure 3 shows the training progress of the RL algorithms for the 20K and 100K datasets. While RL performs well in the 20K dataset, it fails to even beat the baseline algorithm in 100K dataset.

We also evaluated HIM. We ran the Hindsight algorithm, logged its state and actions. We then used various ML models to learn to predict the Hindsight decision from the state. We got the best performance using both Random Forests [21], and Support Vector Machines [22], we report whichever yielded the higher accuracy on the validation data here[1]. The models have a validation accuracy of 79% and 85% for 20K and 100K datasets respectively. We used the learned model to make placement decisions for the test μVM partition. We use the same learned model across different fleet sizes to test generalization. Figures 1 shows the results for all datasets across 10 runs on the test μVM partition.

Figure 1a shows the results of the μVM packing. Due to the sporadic nature of CPU use (Figure 2), the temporal characteristics of μVM and PM CPU use are useful in determining a good placement. The Hindsight algorithm has access to the entire future of PM and μVM use and can pick the PM whose low usage period (troughs in the timeseries plot) align with the peak μVM use. The Hindsight algorithm uses Equations 2 and 3 to determine the PM whose maximum CPU use is minimized after the μVM is placed. This temporal information is lost in the Baseline algorithm because of the coarse featurization, and hence the performance is worse. The Hindsight outperforming the Baseline and random algorithms by 12% and 21% on average respectively.

The HIM performance is somewhere between Baseline and Hindsight, demonstrating that the model has indeed learned to make good decisions from the Hindsight dataset, while relying only on features available at decision time. HIM improves upon Baseline by 11% on average, and performs almost as well as the Hindsight algorithm.

HIM generalizes to the 2D case and is able to find a good trade-off between CPU and Memory features. In Figure 1a, we see that RL outperforms the Hindsight expert. This indicates that better trade-offs between CPU and Memory than the L^2 norm we used in Equation (4) are possible in the short term to minimize the long term PAR. If we introduced μVM features or raw timeseries as features and used LSTMs to represent the policy, the model overfitted to the training data and gave poor validation results. This indicates that, while the data we have seems large, it is still insufficient for data hungry RL models. HIM is much more sample efficient than RL, and we plan to explore the use of HIM as a pre-trained model for RL training in future work. Finally, we observe that within an episode, there were occasionally large jumps in the reward, where a few placement decisions lead to large shift in rewards, while most decisions have no impact. In ongoing work, we are working to characterize these 'important' placement decisions, and adjust the loss function used in training HIM to further improve performance.

6 Conclusions

We have demonstrated that we can exploit historical data to learn to optimize μVM placement. The results presented here are an initial proof of concept, and the algorithms need to evaluated on a variety of datasets and tested for robustness over longer periods of time in production settings. We continue to evaluate how to seed RL models with the HIM model, since RL demonstrated the ability to perform better than the Hindsight algorithm. It may also be possible to improve on the results with better feature engineering and machine learning models. Much of our current fleet cost can be attributed to how many μVMs are present at a time, and we can use similar machine learning approaches to identify when to create and destroy μVMs to save on infrastructure costs. The approaches presented here can be extended to reduce the operational cost of other compute services such as allocating PMs for autoscaling and many other managed serverless services.

[1]The hyperparameters used are given in the Appendix

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[2] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing sla violations. In *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 119–128. IEEE, 2007.

[3] Zhen Xiao, Weijia Song, and Qi Chen. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE transactions on parallel and distributed systems*, 24(6):1107–1117, 2012.

[4] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, 2010.

[5] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, 2013.

[6] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 419–434, 2020.

[7] Nane Kratzke. A brief history of cloud application architectures. *Applied Sciences*, 8(8):1368, 2018.

[8] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. *ACM SIGARCH computer architecture news*, 35(2):13–23, 2007.

[9] Weijia Song, Zhen Xiao, Qi Chen, and Haipeng Luo. Adaptive resource provisioning for the cloud using online bin packing. *IEEE Transactions on Computers*, 63(11):2647–2660, 2013.

[10] Varun Gupta and Ana Radovanovic. Online stochastic bin packing. *arXiv preprint arXiv:1211.2687*, 2012.

[11] Henrik I Christensen, Arindam Khan, Sebastian Pokutta, and Prasad Tetali. Multidimensional bin packing and other related problems: A survey, 2016.

[12] Tao Chen, Yaoming Zhu, Xiaofeng Gao, Linghe Kong, Guihai Chen, and Yongjian Wang. Improving resource utilization via virtual machine placement in data center networks. *Mobile Networks and Applications*, 23(2):227–238, 2018.

[13] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[14] Edwin KP Chong, Robert L Givan, and Hyeong Soo Chang. A framework for simulation-based network control via hindsight optimization. In *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No. 00CH37187)*, volume 2, pages 1433–1438. IEEE, 2000.

[15] Andrea W Richa, M Mitzenmacher, and R Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.

[16] Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis. Efficient resource provisioning in compute clouds via vm multiplexing. In *Proceedings of the 7th international conference on Autonomic computing*, pages 11–20, 2010.

[17] Tao Chen, Yaoming Zhu, Xiaofeng Gao, Linghe Kong, Guihai Chen, and Yongjian Wang. Correlation-aware virtual machine placement in data center networks. In *Cloud Computing, Security, Privacy in New Computing Environments*, pages 22–32. Springer, 2016.

[18] Bochao Shen, Ravi Sundaram, Alexander Russell, Srinivas Aiyar, Karan Gupta, Abhinay Nagpal, Aditya Ramesh, and Himanshu Shukla. High availability for vm placement and a stochastic model for multiple knapsack. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2017.

[19] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

[20] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pages 305–313, 1989.

[21] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[22] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[23] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[24] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E Gonzalez, Michael I Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. *arXiv preprint arXiv:1712.09381*, 2017.

[25] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[26] Eugen Feller, Louis Rilling, and Christine Morin. Snooze: A scalable and autonomic virtual machine management framework for private clouds. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 482–489. IEEE, 2012.

[27] Adrien Lebre, Jonathan Pastor, and Mario Südholt. Vmplaces: A generic tool to investigate and compare vm placement algorithms. In *European Conference on Parallel Processing*, pages 317–329. Springer, 2015.

[28] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. {GRAPHENE}: Packing and dependency-aware scheduling for data-parallel clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 81–97, 2016.

[29] Sameer Agarwal, Srikanth Kandula, Nico Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Reoptimizing data parallel computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 281–294, 2012.

[30] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 99–112, 2012.

[31] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.

[32] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. 2019.

[33] Edward G Coffman, Jr, Michael R Garey, and David S Johnson. Dynamic bin packing. *SIAM Journal on Computing*, 12(2):227–258, 1983.

[34] Varun Gupta and Ana Radovanovic. Lagrangian-based online stochastic bin packing. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):467–468, 2015.

[35] Joseph Wun-Tat Chan, Prudence WH Wong, and Fencol CC Yung. On dynamic bin packing: An improved lower bound and resource augmentation analysis. *Algorithmica*, 53(2):172–206, 2009.

[36] Alexander L Stolyar and Yuan Zhong. A large-scale service system with packing constraints: Minimizing the number of occupied servers. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):41–52, 2013.

[37] Xin Li, Zhuzhong Qian, Sanglu Lu, and Jie Wu. Energy efficient virtual machine placement algorithm with balanced and improved resource utilization in a data center. *Mathematical and Computer Modelling*, 58(5-6):1222–1235, 2013.

[38] Jungsoo Kim, Martino Ruggiero, David Atienza, and Marcel Lederberger. Correlation-aware virtual machine allocation for energy-efficient datacenters. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1345–1350. IEEE, 2013.

[39] Rachael Shaw, Enda Howley, and Enda Barrett. A predictive anti-correlated virtual machine placement algorithm for green cloud computing. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, pages 267–276. IEEE, 2018.

[40] Aviv Tamar, Garrett Thomas, Tianhao Zhang, Sergey Levine, and Pieter Abbeel. Learning from the hindsight plan—episodic mpc improvement. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 336–343. IEEE, 2017.

[41] Eduardo F Camacho and Carlos Bordons Alba. *Model predictive control*. Springer Science & Business Media, 2013.

[42] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in neural information processing systems*, pages 5048–5058, 2017.

[43] Paulo Rauber, Avinash Ummadisingu, Filipe Mutz, and Juergen Schmidhuber. Hindsight policy gradients. *arXiv preprint arXiv:1711.06006*, 2017.

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

Appendix

A μVM Placement in our fleet

AWS Lambda provides function execution as a managed service. Users provide the function they want to execute, written in their preferred programming language, configure limits on memory use in 64 MB increments from 128 MB up to >3000 MB, and configure execution time maximums in 1 second increments up to 15 minutes. They can then execute the function as often as they like, using a variety of different event triggers. Users are alleviated from the burden of provisioning infrastructure and save on costs as they are billed only for the function execution time. The cloud provider provisions the compute resources, creates *μVMs*, installs the required dependencies and executes the function for the user. Users can use these functions for many different purposes, from real-time data processing to serving web requests. We aim to reduce the operational cost of the service.

A.1 Brief Overview of our Service

We describe the model we used for our problem formulation in the paper. The model system is inspired by AWS Lambda, but redacts sensitive proprietary information.

In our model system, the infrastructure consists of a fleet of data center servers, which we refer to as *physical machines (PMs)*. Once a user configures and calls the function, we create a μVM for that function in one of the PMs, and execute it. The μVM is not destroyed immediately, in case the user executes again immediately. The μVM is eventually deleted if it remains idle for a threshold amount of time. Each function call is called a *task*, and the user can execute the same function concurrently. Hence, a single function may have multiple μVMs. Each PM can host and execute multiple μVMs at a time.

Functions to μVMs have a one-to-many relationship, and μVMs are created and used for only one function. Two different functions cannot share execution on the same μVM. A μVM can only accommodate one task at a time. While the μVM is processing an task, it is said to be active. Once function execution ends for that task, the μVM enters an idle state and can be kept around to accommodate additional tasks if and when they arrive. Invoking on an idle μVM results in lower user latency compared to creating a new μVM.

Each decision in infrastructure management and function request routing impacts the operational cost and user experience. If an idle μVM for a particular function is available an task arrives, then the latency of execution is reduced. μVMs use memory even when they are not executing a function. Hence, idle μVMs waste resources. If μVMs are packed in the PMs tightly, the number of PMs in the fleet is reduced. However, we need to ensure the μVMs are not packed too tightly to ensure they can execute without hitting PM resource limits, or PM performance limits which would negatively impact the user observed latency or availability. We also need to ensure the PMs have enough resources to create new μVMs as they are needed. Proactive deletion of μVMs can reduce memory use. However, aggressive deletion may create churn with reactive creation of μVMs, and increase execution latency. Each of these decisions can be optimized to reduce cost. We focus on the placement of μVMs in PMs to ensure tight packing, and as a result, reduce the fleet cost.

B Data Characteristics

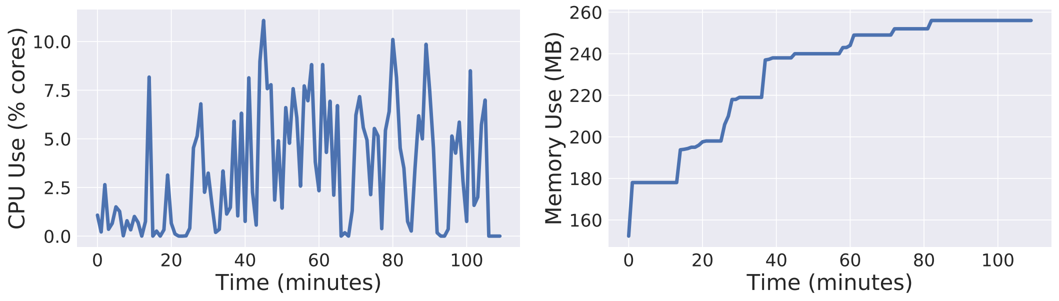


Figure 2: CPU and memory use of a long lived μVM in our dataset. CPU is used only when the μVM is executing tasks and memory use is the maximum of memory used in μVM history.

We collect a dataset with $\sim$200K μVMs across 24 hours of use from AWS Lambda, and analyze its characteristics. Figure 2 shows the CPU and memory use of a particularly extreme μVM in our dataset across 2 hours when it was active. Data is recorded every minute and consists of μVM CPU and memory use. The CPU use spikes up whenever tasks are executed. Most μVMs are relatively small (median values – memory: $\sim$100MB, CPU: $\sim$15-20% of a cpu core, execution time: 0.2-0.6s, inter arrival time: 50-100s) but a small percentage of μVMs have extreme values (99th percentile or p99 values – memory > 1GB, CPU > 150%, execution time > 90s, inter arrival time: >10 min). The median life of a μVM is $\sim$15 minutes and the p99 value is > 2 hours. The memory use grows gradually as tasks take up more memory and assigned memory is not reclaimed until μVM is deleted. At the time of the placement decision, these detailed μVM characteristics are unknown, since each μVM is unique from the perspective of the placement algorithm.

To make good placement decisions, we may need to forecast how μVMs use resources in aggregate. In particular, we need to place μVMs such that the aggregate use does not exceed the capacity of the PM. In case of memory, an individual μVM use has low variance. Hence, aggregate memory use can be approximated as sum of individual μVM memory use. However, aggregate CPU use varies depending on when tasks are executed and how the tasks are inter-leaved across μVMs. Hence, estimating aggregate CPU use is challenging. Aggregate resource use also depends on when the μVMs are deleted.

C Related Work

VM packing has been studied extensively in literature [2, 3, 4]. A popular class of problems is VM migration [2, 3, 26, 27]. Here VMs are migrated periodically to reduce hot spots in the data center and exploit freed resources. The VMs and the tasks associated have longer lifetime compared to the μVMs we studied, and forecasting algorithms such as exponentially weighted mean average, sufficed to predict the workload well. In contrast, we consider placement of μVMs when they are created, and do not consider migration in our formulation. Our tasks are short and bursty, making it resource use prediction hard. Some works consider the dependencies that exist between tasks [28, 29, 30, 4] using heuristics [28, 31] and even RL recently [32]. We work in an orthogonal setting, where the tasks and μVMs are independent from each other.

Our problem setting can be considered as a dynamic bin packing problem [33], where items are packed into bins one at a time, their arrival order is unknown and they depart at an unknown time. The theoretical properties of the problem has been studied extensively, and it has been shown that even myopic algorithms like First Fit perform well compared to optimal packing [34, 35, 36]. Xin et al. [37] show that multi-dimensional online packing is an NP-complete problem. They present a practical algorithm that avoids resource fragmentation and outperforms First Fit. However, in these works, the VM resource use is considered fixed and the algorithms have full access to the state of all PMs. In our use case, VM resource use changes over time. We need to pack VMs such that their peak usages are not aligned, so that the resource capacity of the PM is never exceeded.

Several works have considered packing of VMs such that their usage is *anti-correlated* with each other and leads to tight packing [16, 17, 38, 39, 12]. Kim et al. [38] consider the covariance of the VM use from historical data and use a Pearson correlation based heuristic for placement. Meng et al. [16] use ARMA and kernel density estimation based forecasting model to predict VM use. Chen et al. [17] use a neural network model for forecasting. In a follow up work [12], they improve their forecasting using PCA with ARIMA models. The forecasting models are used for placement using solutions like First Fit, Best Fit or genetic algorithms. All of these consider time scales of hours to days, making resource use pattern prediction feasible. They also consider the VM migration setting, where the horizon of forecasting is only one placement time step. In contrast, our work considers online placement with no migration, and hence we need to forecast the VM resource for its lifetime. In our μVM dataset, the time scales of task execution are of the order of seconds, and the bursty nature of tasks makes it a difficult forecasting problem. Therefore, we take a hindsight imitation approach, which circumvents the forecasting model and directly learns a hedged placement policy.

The hindsight optimization algorithm was proposed by Chong et al. [14]. They used hindsight information to learn a Q function in a tabular setting for a network traffic control problem. Tamar et al. [40] proposed a similar algorithm where they learn a task execution plan with model predictive control (MPC) [41] using hindsight data, available only after decisions have been made. The learned plan is used as to shape the cost function of the online MPC algorithm that plans on a shorter horizon. Our work is related to, and draws inspiration from these approaches, but we directly use classification based imitation learning instead of MPC or Q learning to learn from hindsight. Our work is orthogonal to Hindsight Experience Replay [42] and Hindsight Policy Gradients [43], which are designed for goal oriented problems.

D Hyperparameters

Below we provide the hyperparameters used for each experiment. Note that no formal hyperparameter search was conducted and the hyperaparameters were generally set to default values found in the Ray RLlib examples [24].

Table 1: Hyperparameters used for RL experiments. We used the Proximal Policy Optimization algorithm [25], as implemented in Ray RLlib repository [24].

Hyperparameter	**Value**
Gamma	0.995
KL coefficient	1.0
SGD iterations	5
Minibatch size	512
Train batch size	8192
Learning rate	0.00001
Hidden layers	[256, 256]
Use GAE	False

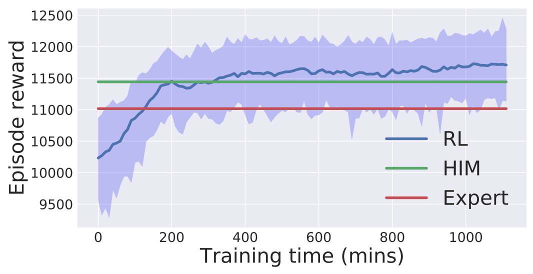
Table 2: Hyperparameters used for Hindsight Imitation Learning experiments with Support Vector Machines. We used the implementation in the Scikit-Learn library [44].

Hyperparameter	**Value**
C	10
Cache size	200
Class weight	None
Coef 0	0.0
Decision function shape	ovr
Degree	3
Gamma	Auto Deprecated
Kernel	RBF
Max Iter	-1
Probability	False
Random State	None
Shrinking	True
tol	0.001

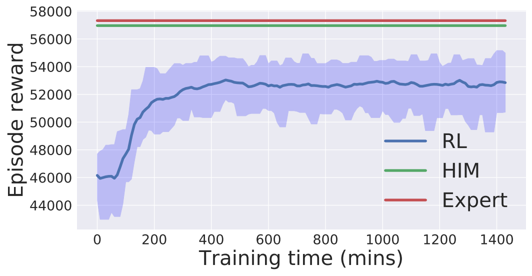
Table 3: Hyperparameters used for Hindsight Imitation Learning experiments with Random Forests. We used the implementation in the Scikit-Learn library [44].

Hyperparameter	**Value**
Bootstrap	True
Class Weight	None
Criterion	gini
Max Depth	None
Max Features	auto
Max Leaf Nodes	0.None
Min Impurity Decrease	0.0005
Min Impurity Split	None
Min Samples Leaf	1
Min Samples Split	2
Min Weight Fraction Leaf	0.0
Number of Estimators	50
Number of Jobs	None
OOB Score	False
Random State	1
Warm Start	False

E RL Training Curves



(a)



(b)

Figure 3: RL training curves for μVM packing based on memory and CPU use for the 20K (a) and 100K (b) datasets. We see that when RL succeeds, it does so quickly. However, often the performance plateaus much before performance of even the Baseline algorithm is reached.