VMR^{2}L: Virtual Machines Rescheduling Using Reinforcement Learning in Data Centers

Xianzhong Ding[*1] **Yunkai Zhang**[*2] **Binbin Chen**[3] **Donghao Ying**[2] **Tieying Zhang**[3]
Jianjun Chen[3] **Lei Zhang**[3] **Alberto Cerpa**[1] **Wan Du**[1]
[1]UC Merced [2]UC Berkeley [3]ByteDance

Abstract

Modern industry-scale data centers need to manage a large number of virtual machines (VMs). Due to the continual creation and release of VMs, many small resource fragments are scattered across physical machines (PMs). To handle these fragments, data centers periodically reschedule some VMs to alternative PMs, a practice commonly referred to as VM rescheduling.

Despite the increasing importance of VM rescheduling as data centers grow in size, the problem remains understudied. We first show that, unlike most combinatorial optimization tasks, the inference time of VM rescheduling algorithms significantly influences their performance, due to dynamic VM state changes during this period. This causes many existing methods to scale poorly. Therefore, we develop a reinforcement learning system for VM rescheduling, VMR^{2}L, which incorporates a set of customized techniques, such as a two-stage framework that accommodates diverse constraints and workload conditions, as well as a feature extraction module that captures relational information specific to rescheduling. We conduct extensive experiments with datasets generated from an industry-scale data center. Our results show that VMR^{2}L can achieve a performance comparable to the optimal solution but with a running time of seconds.[2]

1 Introduction

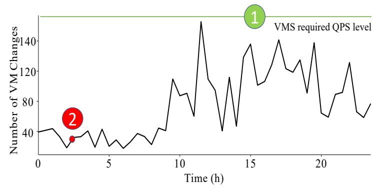

Figure 1: Avg. VM changes.

Cloud service providers allow end-users to access computing resources, such as CPU and memory. They adopt resource virtualization to maximize hardware utilization, allocating Virtual Machines (VMs)[1, 2] with the requested resources to end-users [3, 4, 5]. An industry-scale data center typically has several clusters, where each cluster has hundreds to thousands of Physical Machines (PMs), and each PM can host multiple VMs that run independently [6]. A central server manages all VM requests on PMs by performing two tasks, scheduling and rescheduling, in order to achieve different resource utilization goals, such as minimizing the overall fragment rate (FR) or maximizing number of available PMs.[3]

VM Scheduling (VMS). When new VM requests arrive, there can be multiple available PMs to host them. Figure 1 shows the maximum number of VMs changes (VMs arriving and exiting) per minute averaged over a 30-day period of a cluster from our in-house data center. To ensure the system is robust, the VMS algorithm needs to meet the maximum number of VMs changes as indicated by the

[*]The authors contribute equally.

[2]We release our datasets and code here: `https://anonymous.4open.science/r/VMR2L-BEA6`.

[3]Due to space limits, we mainly explain our methods in terms of minimizing FR, whose formal definition is presented in Appendix A.2.

green line. The high queries per second (QPS) requirement deems only heuristic methods feasible for VMS. In fact, ByteDance uses best-fit [7, 8], which sorts all PMs that meet the requirements of the current VM according to the amount of FR reduction before and after this VM is added, and chooses the PM with the largest reduction.

VM Rescheduling (VMR). However, simple heuristics are often far from optimal, and the continual exiting of completed VMs results in many fragments scattered across PMs. As a result, rescheduling is critical to optimize resource usage, which migrates VMs from their current PMs to new destination PMs. Due to the overhead of VM migrations, a migration number limit (MNL) is typically chosen to be $2 \sim 3\%$ of all VMs. In most cases, rescheduling is only allowed to happen within the same cluster. Note that while the VMR algorithm computes a solution, VMS is still handing new VM requests and completed VMs are also being deleted. The dynamic nature of VM states causing the computed VMR solution to no longer be optimal or even feasible[4]. Therefore, even though VMR mostly happens during off-peak hours where there are fewer VM changes[5], VMR still needs to be very efficient. We conduct an experiment to quantify how the VMR inference time affects the achieved performance.

Motivation Experiment. We can formulate the VM rescheduling problem as a Mixed Integer Programming (MIP) problem, where the constraints come from the service expectations and the available hardware resources. An off-the-shelf MIP solver, such as Gurobi [9] and CPLEX [10], can achieve a near-optimal objective, but it suffers from an inference time that grows exponentially with the number of VMs and PMs, and thus fails to scale to large data centers.

To see how a near-optimal solution can result in a suboptimal achieved performance due to its poor inference time, we conduct an experiment on real traces from our in-house data center by selecting 20 random initial VM-PM mappings. For each mapping, we use Gurobi to compute a near-optimal solution to the MIP formulation of VMR, which takes 50.55 minutes. However, since VMs were dynamically arriving and exiting, most actions were no longer feasible and will fail to be deployed after 50 minutes. We then compute the final performance that could be achieved as if the near-optimal solution was instead returned in a shorter period of time, averaged over the 20 mappings. Figure 2 shows that the solution remains near-optimal if it could be computed within 5 seconds. However, FR reduction quickly diminishes when the inference time exceeds 7 seconds.

Limitations of the Current Methods. The experiment results reveal that different from bin-packing and other MIP applications, VM rescheduling requires an algorithm that has an inference time strictly under five seconds. To accelerate the running speed of the MIP approach, hand-tuned heuristics can be integrated into the process, e.g., adding constraints to limit the solver's search space. These heuristics must trade off between the optimality of the solution and the tractability of the problem. Unfortunately, even highly-skilled experts need many iterations to manually find a proper trade-off, and no universal heuristics can achieve a good trade-off for all VM rescheduling scenarios.

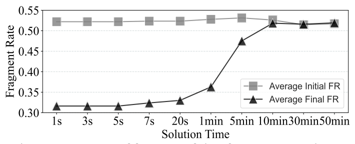

Figure 2: Effect of inference time on achieved performance.

In this work, we develop VMR^{2}L, a deep Reinforcement Learning (RL) system for VM rescheduling. VMR^{2}L trains a Deep Neural Network (DNN) as the rescheduling agent. RL is a great fit for VMR since VMR has no aleatoric uncertainties, i.e., the next state can be exactly simulated given the current state and action. This allows us to build a simulator that only requires the initial VM-PM mappings for training, without having to interact with a real data center. With our RL formulation and customized embedding techniques, inference can be done within a few seconds and can scales easily to a large number of VMs and PMs. Extensive evaluation on two collected datasets demonstrates that VMR^{2}L can generate a solution that is only 7.14% worse than the optimal solution. We summarize the contributions of this paper as follows:

- **RL for VM rescheduling.** We identify the unique characteristics of the VM rescheduling problem in terms of latency requirement and aleatoric uncertainties, which motivate its formulation as a RL problem.

[4]A VM will not be rescheduled if it has exited or the destination PM no longer has enough resources.

[5]In less common cases, VMR is also performed if a high FR is observed that could potentially lead to insufficient resources for upcoming VM requests, which requires an even higher latency.

- **Customized RL techniques for VM rescheduling.** We tackle two challenges in designing a RL framework for VM rescheduling with two customized techniques — a two-stage framework and an effective feature extraction module.
- **A VMR^{2}L prototype and extensive evaluation.** We develop a prototype of VMR^{2}L and conduct extensive experiments over two real datasets. We release the datasets and a custom Gym environment for RL training.

2 Design of VMR^{2}L

2.1 VM Rescheduling as an RL Problem

In this paper, we adopt a deep reinforcement learning (RL) approach to address the VM rescheduling problem. A VMR request starts an episode, which involves migration number limit (MNL) steps. At each migration step, the agent reschedules one VM from its source PM to a new destination PM based on the current PM and VM states. Notably, the environment is deterministic – given the initial state and an action, we can directly simulate the change in objective and the next state. This allows us to build a simulator to train the agent, as detailed in Appendix C.1.

State Representation. The state input to the DRL agent contains two sets of features. The first set contains four PM features for each NUMA[6], specifically the remaining CPU and memory resources, current FR, and fragment sizes. The second set contains 14 VM features including requested CPU and memory for each NUMA, fragment sizes, and the source PM details. If a single NUMA is requested, zeros are used as placeholders for the other NUMA. Features are min-max normalized.

Action Representation. The action at each step can be represented as a 2-tuple (k, i). Specifically, the action is to reschedule a VM k from its source PM to a destination PM i. Note that the source PM can be retrieved once we select k.

Reward Representation. The goal of VM rescheduling is to minimize the FR across all PMs. While we could return the FR of all PMs as a single final reward to the agent after finishing an entire episode, it corresponds to a form of sparse reward and it is known to be difficult for training [11]. Instead, we propose to generate dense rewards and use the change in FR on the source PM and the destination PM as an intermediate reward at each step. As such, the reward range is naturally scaled down to $[-2, 2]$, which we further normalize by dividing with a constant c [12]. We calculate the rescaled fragment size on each NUMA by $S_i = \sum_{j=0}^{1} \left(\tilde{U}_{i,j} \% X \right) \div c$, and define reward as $R = (S_{\text{before, src}} - S_{\text{after, src}}) + (S_{\text{before, dest}} - S_{\text{after, dest}})$, where $S_{\text{before},\cdot}$ and $S_{\text{after},\cdot}$ are the fragment changes before and after the selected VM leaves (enters) the source (destination) PM.

2.2 A Two-Stage Framework

We uses PPO [13] as the backbone of our DRL framework. To better accommodate a variety of constraints, we leverage the characteristics of the VMR problem and design a two-stage framework that allows the action tuple to be generated sequentially. As shown in Figure 3, in Stage 1, the VM actor selects the VM candidate to be rescheduled. Once a candidate VM is selected, we can efficiently mask out all the PMs that cannot host the candidate VM and then proceed to Stage 2, where the PM actor selects an appropriate destination PM from the remaining PMs. This design completely avoids illegal actions for various types of constraints and thus circumvents the necessity of heavy penalties. Also, it dedicates two separate networks to select the VM candidate and the destination PM, which simplifies the VMR task by decomposing the action tuple.

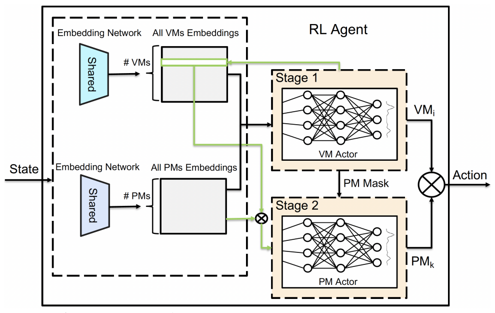

Figure 3: The two-stage RL agent.

Scalability. To make effective rescheduling decisions, VMR^{2}L must extract meaningful representations of the state observation, which include features of each individual PM and VM as well as

[6]Each PM contains two non-uniform memory accesses (NUMAs).

their affiliations. However, the number of VMs can vary drastically even in the same data center. To encode these features, we propose to share two small embedding networks across all VMs and PMs (Figure 4) — one to process each PM's features and another one to process each VM's features. As such, the number of weight parameters is *independent* of the number of machines in the system. This is achieved via an attention-based transformer model [14] but tailored for rescheduling. Transformers have demonstrated strong performances in combinatorial optimization, such as in vector bin-packing [15, 16]. However, there is a notable difference in VM rescheduling: we must choose from a set of VMs that have already been assigned to PMs.

2.3 Feature Extraction with Sparse Attention

Tree-level Features. Consider the example shown in Appendix B.1. Not knowing which exact types of VMs are hosted on the same PM prevents the vanilla transformer from optimizing for such long-term rewards. We argue that each VM must be aware of which VMs are co-hosted on the same PM, resembling a tree structure with PM as the root and VMs as leaves. To enable VMs to recognize co-hosted VMs, we introduce an additional stage of *sparse local-attention* within each PM tree. This restricts PMs and VMs to attend to each other if and only if they belong to the same tree. Detailed network architecture can be found in Appendix B.5.

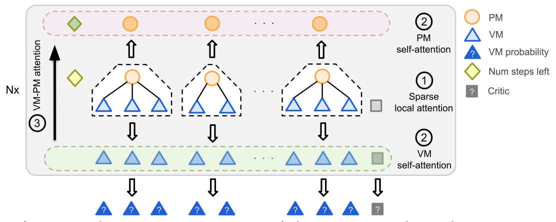

Figure 4: VM actor with sparse local-attention capturing tree-level features.

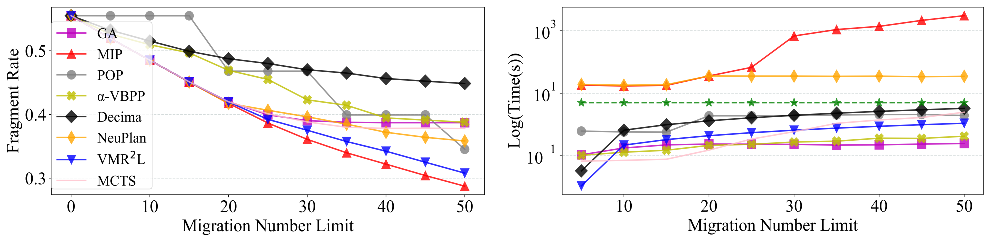

Figure 5: FR (left) and inference time (right) on the Medium dataset at different MNLs.

3 Evaluation

We collect two datasets from an industry-scaled cloud data center – one Medium dataset with 2089 VMs and 280 PMs, and one Large dataset with 4546 VMs and 1176 PMs. We implement VMR^{2}L based on the CleanRL framework [17] with PyTorch using PPO as the backbone [13]. We compare with seven baseline methods from six categories: heuristics (e.g., greedy, α-VBPP), optimization algorithms (e.g., MIP), approximate algorithms (e.g., POP), search-based algorithms (e.g., MCTS), deep learning-based (e.g., Decima), and hybrid methods (e.g., NeuPlan).[7]

Experiment. We compare VMR^{2}L with the baselines on the dataset in terms of the optimality and inference latency under different migration number limits (MNLs). Figure 5 summarizes the results, which shows that VMR^{2}L is able to consistently reduce FR given different MNLs at a rate closest to MIP compared to other baselines. For a more comprehensive analysis, refer to Appendix C.3. The VMR^{2}L training details can be found in Appendix A.5.

4 Conclusion

Compared to conventional bin-packing applications, VM rescheduling presents unique challenges due to the expanding size of data centers. It needs to handle many VMs while meeting a strict inference speed requirement. To this end, we introduce VMR^{2}L, a tailored deep RL solution featuring a two-stage framework for diverse service constraints and a sparse attention module for better feature extractions. We hope that our released datasets and RL environment will support future research in this area.

[7]See Appendix C.2 and C.1 for details on the baselines and datasets, respectively.

References

[1] Jörg Thalheim, Peter Okelmann, Harshavardhan Unnibhavi, Redha Gouicem, and Pramod Bhatotia. Vmsh: hypervisor-agnostic guest overlays for vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 678–696, 2022.

[2] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286, 2005.

[3] Xianzhong Ding, Le Chen, Murali Emani, Chunhua Liao, Pei-Hung Lin, Tristan Vanderbruggen, Zhen Xie, Alberto Cerpa, and Wan Du. Hpc-gpt: Integrating large language model for high-performance computing. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 951–960, 2023.

[4] Mao Lin, Keren Zhou, and Pengfei Su. Drgpum: Guiding memory optimization for gpu-accelerated applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 164–178, 2023.

[5] Kang Yang, Yuning Chen, Xuanren Chen, and Wan Du. Link quality modeling for lora networks in orchards. In *Proceedings of the 22nd International Conference on Information Processing in Sensor Networks*, pages 27–39, 2023.

[6] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean: Vm allocation service at scale. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 845–861, 2020.

[7] Chi Trung Ha, Trung Thanh Nguyen, Lam Thu Bui, and Ran Wang. An online packing heuristic for the three-dimensional container loading problem in dynamic environments and the physical internet. In *Applications of Evolutionary Computation: 20th European Conference, EvoApplications 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, Part II 20*, pages 140–155. Springer, 2017.

[8] Francisco Parreño, Ramón Alvarez-Valdés, Jose Manuel Tamarit, and Jose Fernando Oliveira. A maximal-space algorithm for the container loading problem. *INFORMS Journal on Computing*, 20(3):412–422, 2008.

[9] Gurobi solver. `https://www.gurobi.com/`.

[10] Cplex optimizer. `https://www.ibm.com/analytics/cplex-optimizer`.

[11] Desik Rengarajan, Gargi Vaidya, Akshay Sarvesh, Dileep Kalathil, and Srinivas Shakkottai. Reinforcement learning with sparse rewards using guidance from offline demonstration. *arXiv preprint arXiv:2202.04628*, 2022.

[12] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

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

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[15] Jingwei Zhang, Bin Zi, and Xiaoyu Ge. Attend2pack: Bin packing through deep reinforcement learning with attention. *ArXiv*, abs/2107.04333, 2021.

[16] Dongda Li, Changwei Ren, Zhaoquan Gu, Yuexuan Wang, and Francis Lau. Solving packing problems by conditional query learning, 2020.

[17] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

[18] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3):729, 2012.

[19] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[20] Raja Wasim Ahmad, Abdullah Gani, Siti Hafizah Ab Hamid, Muhammad Shiraz, Abdullah Yousafzai, and Feng Xia. A survey on virtual machine migration and server consolidation frameworks for cloud data centers. *Journal of network and computer applications*, 52:11–25, 2015.

[21] Qingpeng Cai, Will Hang, Azalia Mirhoseini, George Tucker, Jingtao Wang, and Wei Wei. Reinforcement learning driven heuristic optimization. *Workshop on Deep Reinforcement Learning for Knowledge Discovery (DRL4KDD)*, abs/1906.06639, 2019.

[22] Haoyuan Hu, Xiaodong Zhang, Xiaowei Yan, Longfei Wang, and Yinghui Xu. Solving a new 3d bin packing problem with deep reinforcement learning method, 2017.

[23] Lu Duan, Haoyuan Hu, Yu Qian, Yu Gong, Xiaodong Zhang, Jiangwen Wei, and Yinghui Xu. A multi-task selected learning approach for solving 3d flexible bin packing problem. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '19, page 1386–1394, Richland, SC, 2019. International Foundation for Autonomous Agents and Multiagent Systems.

[24] Ye Xia, Mauricio Tsugawa, Jose AB Fortes, and Shigang Chen. Large-scale vm placement with disk anti-colocation constraints using hierarchical decomposition and mixed integer programming. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1361–1374, 2016.

[25] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. *research. microsoft. com*, 2011.

[26] A. Paul Davies and Eberhard E. Bischoff. Weight distribution considerations in container loading. *European Journal of Operational Research*, 114(3):509–527, May 1999.

[27] Xijun Li, Mingxuan Yuan, Di Chen, Jianguo Yao, and Jia Zeng. A data-driven three-layer algorithm for split delivery vehicle routing problem with 3d container loading constraint. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*, KDD '18, page 528–536, New York, NY, USA, 2018. Association for Computing Machinery.

[28] Hang Zhao, Qijin She, Chenyang Zhu, Yin Yang, and Kai Xu. Online 3d bin packing with constrained deep reinforcement learning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 741–749. AAAI Press, 2021.

[29] Qianwen Zhu, Xihan Li, Zihan Zhang, Zhixing Luo, Xialiang Tong, Mingxuan Yuan, and Jia Zeng. Learning to pack: A data-driven tree search algorithm for large-scale 3d bin packing problem. In *Proceedings of the 30th ACM International Conference on Information Knowledge Management*, CIKM '21, page 4393–4402, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Ameer Haj-Ali, Qijing Jenny Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems*, 2:70–81, 2020.

[31] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[32] Marc Etheve, Zacharie Alès, Côme Bissuel, Olivier Juan, and Safia Kedad-Sidhoum. Reinforcement learning for variable selection in a branch and bound algorithm. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 176–185. Springer, 2020.

[33] Prateek Gupta, Maxime Gasse, Elias Khalil, Pawan Mudigonda, Andrea Lodi, and Yoshua Bengio. Hybrid models for learning to branch. *Advances in neural information processing systems*, 33:18087–18097, 2020.

[34] Haoran Sun, Wenbo Chen, Hui Li, and Le Song. Improving learning to branch via reinforcement learning. 2020.

[35] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut. In *International conference on machine learning*, pages 9367–9376. PMLR, 2020.

[36] Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre A Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3677–3687, 2021.

[37] Lara Scavuzzo, Feng Yang Chen, Didier Chételat, Maxime Gasse, Andrea Lodi, Neil Yorke-Smith, and Karen Aardal. Learning to branch with tree mdps. *arXiv preprint arXiv:2205.11107*, 2022.

[38] Jialin Song, Yisong Yue, Bistra Dilkina, et al. A general large neighborhood search framework for solving integer linear programs. *Advances in Neural Information Processing Systems*, 33:20012–20023, 2020.

[39] Thomas Barrett, William Clements, Jakob Foerster, and Alex Lvovsky. Exploratory combinatorial optimization with reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3243–3250, 2020.

[40] Meng Qi, Mengxin Wang, and Zuo-Jun Shen. Smart feasibility pump: Reinforcement learning for (mixed) integer programming. *arXiv preprint arXiv:2102.09663*, 2021.

[41] Hang Zhu, Varun Gupta, Satyajeet Singh Ahuja, Yuandong Tian, Ying Zhang, and Xin Jin. Network planning with deep reinforcement learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 258–271, 2021.

[42] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 521–537, 2021.

[43] 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.

[44] Sham M Kakade. A natural policy gradient. *Advances in neural information processing systems*, 14, 2001.

[45] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *ICML*. PMLR, 2015.

[46] Kyriakos G Vamvoudakis and Frank L Lewis. Online actor–critic algorithm to solve the continuous-time infinite horizon optimal control problem. *Automatica*, 46(5):878–888, 2010.

[47] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[48] Dan Hendrycks and Kevin Gimpel. Gaussian Error Linear Units (GELUs). 2016.

[49] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[50] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, 2007.

[51] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. Time limits in reinforcement learning, 2018.

[52] Tianyu He, Xu Tan, Yingce Xia, Di He, Tao Qin, Zhibo Chen, and Tie-Yan Liu. Layer-wise coordination between encoder and decoder for neural machine translation. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[53] Vincent Mai, Kaustubh Mani, and Liam Paull. Sample efficient deep reinforcement learning via uncertainty estimation. In *International Conference on Learning Representations*, 2022.

[54] Xianzhong Ding, Wan Du, and Alberto E Cerpa. Mb2c: Model-based deep reinforcement learning for multi-zone building control. In *Proceedings of the 7th ACM international conference on systems for energy-efficient buildings, cities, and transportation*, pages 50–59, 2020.

[55] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[56] Facebook AI Research. Pytorch: An open source machine learning framework. `https://pytorch.org/`, 2019. Accessed: April 23, 2023.

[57] Kubernetes scheduler. `https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/`.

[58] Marius-Constantin Popescu, Valentina E Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8(7):579–588, 2009.

Appendix Contents

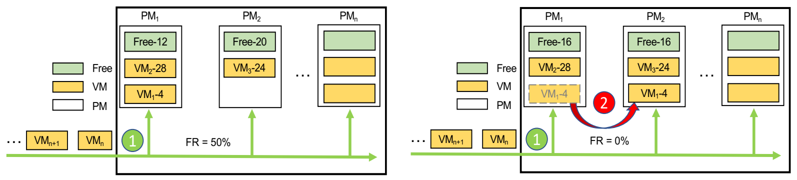

Figure 6: VM scheduling (left) and VM rescheduling (right).

A Appendix

A.1 Why we need VM rescheduling?

Fragment rate (FR) is the ratio between the remaining CPU resources that cannot be further utilized by 16-Core VMs[8] and the total free CPU resources across all PMs. Consider the FR in the left subfigure of Fig. 6. PM$_1$ has 12 CPUs left and PM$_2$ has 20 CPUs left, but only PM$_2$ can host another 16-Core VM, and the remaining $12 + (20 - 16) = 16$ CPUs become fragments. The FR is therefore $16/(12 + 20) = 50\%$. In the right subfigure, VMR reassigns VM$_1$ from PM$_1$ to PM$_2$, leaving 16 free CPUs on each PM, which is just enough to handle an additional 16-Core VM. VMR is able to lower FR to 0%.

A.2 VM Rescheduling Problem Formulation

In this section, we formulate the VM rescheduling problem as a Mixed-Integer Programming (MIP) problem.

In a data center, let $\mathcal{V}, \mathcal{P}$ be the set of VMs and PMs, respectively. On the supply side, a PM $i \in \mathcal{P}$ has two NUMAs[9], where NUMA j can provide $U_{i,j}$ CPU resources and $V_{i,j}$ memory resources. On the demand side, a VM $k \in \mathcal{V}$ requires u_k CPU resources and v_k memory resources and should be deployed on a single PM using $w_k \in \{1, 2\}$ NUMAs. w_k is the number of NUMAs required by VM k (1 for single-NUMA deployment, 2 for double-NUMA). After deploying several VMs on PM $i \in \mathcal{P}$, there remains $\tilde{U}_{i,j}$ spare CPU resources on NUMA j. We define **X-core fragment** of PM i as $\sum_j (\tilde{U}_{i,j} \% X)$, i.e., the remaining CPU resources cannot be further utilized by any additional X-core VMs.

Given an initial assignment of M VMs each onto one of the N PMs, the VM rescheduling task is to reassign a subset of deployed VMs and migrate them each onto a new PM. The maximum number of VMs that we can migrate for a given task is called *Migration Number Limit (MNL)*. We formulate the VM rescheduling as an optimization problem that searches for a reassignment of MNL selected VMs, in order to minimize the total X-core fragments across all PMs:

[8] We define FR in terms of 16-Core VMs, the default VM type in-house, but it can also be defined based other X-Core VMs under different specifications.

[9] Non-uniform memory access.

Table 1: The VM types we consider in our experiments.

VM Types	large	xlarge	2xlarge	4xlarge	8xlarge	16xlarge	22xlarge
Requested CPU	2	4	8	16	32	64	88
Requested Memory (GB)	4	8	16	32	64	128	256
Deploy NUMA	Single	Single	Single	Single	Double	Double	Double

$$\text{Minimize:} \quad \sum_{i,j}\left(U_{i,j} - \sum_k \frac{x_{k,i,j}\cdot u_k}{w_k} - Xy_{i,j}\right) \tag{1}$$

$$\text{Subject to:} \quad \sum_k \frac{x_{k,i,j}\cdot u_k}{w_k} + Xy_{i,j} \le U_{i,j}, \tag{2}$$

$$\sum_k \frac{x_{k,i,j}\cdot v_k}{w_k} \le V_{i,j}, \tag{3}$$

$$\sum_{i,j} x_{k,i,j} = w_k, \tag{4}$$

$$\sum_k (1 - x_{k,i_k,j_k}) \le MNL, \tag{5}$$

$$x_{k,i,0} = x_{k,i,1}, \quad \forall k \in \{k|w_k = 2\}, \tag{6}$$

$$x_{k,i,j} \in \{0,1\} \text{ and } y_{i,j} \in \mathbb{Z}. \tag{7}$$

Here, $\{x, y\}$ are the decision variables, where $x_{k,i,j}$ represents whether VM k is deployed to the NUMA j of PM i in the new assignment (0 for No, 1 for Yes), and $y_{i,j}$ represents the maximum number of X-core VMs can be deployed on NUMA j of PM i using the remaining CPU resources. The objective in Equation 1 is to minimize the total X-core fragments.

Equation 2 and 3 enforce that the resource usage by VMs cannot exceed the total capacity of a PM. Equation 4 indicates that each VM must be deployed on exactly one PM. Equation 5, in which i_k and j_k are the initial PM id and NUMA id (0 for double-NUMA VMs) of VM k, means the total migration number should not exceed the limit. Lastly, Equation 6 restricts VMs with double NUMAs from deploying both of its NUMAs on the same PM.

Note (1) each PM has two NUMAs; (2) w_k is a constant for each VM as determined by their types (Table 1). Thus, $\sum_{i,j} x_{k,i,j} = w_k$ (Equation 4) enforces that the actual NUMA allocation number of VM k matches the desired configuration. When $w_k = 1$, Equation 4 constraints VM k to be deployed on one NUMA of a PM; when $w_k = 2$, Equation 4 constraints VM k to be deployed on both NUMAs of a PM. Note that deploying VM k on two NUMAs of two different PMs (each PM hosting a NUMA) violates $x_{k,i,0} = x_{k,i,1}, \forall k \in \{k|w_k = 2\}$ (Equation 6). Because $w_k \neq 0$, it guarantees each VM is deployed.

A.3 Background on (Deep) Reinforcement Learning

In this section, we give a brief overview on (deep) reinforcement learning while referring the readers to [18, 19] for a more detailed introduction.

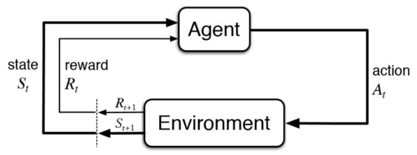

Figure 7: Illustration diagram of reinforcement learning.

Reinforcement Learning A standard Reinforcement Learning (RL) problem is typically characterized by a Markov Decision Process (MDP), where an agent continuously interacts with its environment, as depicted in Fig. 7. At each step (or period), the agent observes a state s_k from the environment and responds with an action a_k in accordance with the current policy π, i.e., $a_k \sim \pi(\cdot|s_k)$. The agent subsequently receives a reward $r_k = r(s_k, a_k)$ as feedback, based on the state and action. The transition to the next state depends solely on the current state and action, i.e., $s_{k+1} \sim \mathbb{P}(\cdot|s_k, a_k)$, where $\mathbb{P}(\cdot|\cdot,\cdot)$ represents the transition probability. In a tabular RL setting, both the state space S and the action space A are assumed to be finite and discrete.

Starting with the initial state s_0, the primary goal of the agent is to learn a policy that optimizes the so-called value function. This function is defined as the expected cumulative rewards received over MNL steps, i.e.,

$$J^\pi(s_0) := \mathbb{E}\left[\sum_{k=0}^{MNL} \gamma^k r(s_k, a_k)\middle|\pi, s_0\right]. \tag{8}$$

Here, the expectation is taken over all possible trajectories under policy π. γ is a discount factor that affects how much weight is given to future rewards. Additionally, for any initial state-action pair (s_0, a_0), the corresponding state-action value function (Q-function) is defined as

$$Q^\pi(s_0, a_0) := \mathbb{E}\left[\sum_{k=0}^{K} \gamma^k r(s_k, a_k)\middle|\pi, (s_0, a_0)\right]. \tag{9}$$

Deep Reinforcement Learning Deep RL, a subset of RL, integrates RL and deep learning. In tabular RL, the policy is directly optimized as a table of dimensions $|S| \times |A|$. However, the exponential scaling of $|S|$ and $|A|$ in high-dimensional state and action spaces makes traditional RL algorithms impractical. Deep RL addresses this issue by employing (deep) neural networks to represent the policy function (or other learned functions) and developing specialized algorithms for scalability.

A.4 Related Work

Connections to Bin Packing. The use of optimized placement mechanisms proved to be successful in a broad set of use cases, including production quality scenarios [20] as well as transportation logistics [21, 22, 23, 24]. A typical solution exploits heuristics based on bin packing [25]. In fact, VM placement can be modeled as a bin-packing problem, where VMs and PMs are objects and bins, respectively. Bin packing typically involves packing a set of items into fixed-sized bins such that the number of bins required [21] or the total surface area is minimized [22, 23]. However, there are two notable differences. First, the problem of VM rescheduling concerns adjusting an initial assignment of VMs to PMs. On the other hand, rebalancing items already packed in bins has received little attention in the context of other bin-packing applications. A critical aspect of the initial assignment is the current VM affiliations, which existing binpacking solutions often do not consider but we show is critical to VMR. Second, the total number of VMs and PMs in a data center can easily go into the range of several thousands or more [24] and is far more than the typical scale of bin packing problems, which typically involve no more than a few hundred items [26, 27, 28, 29].

RL for Optimization Problems. RL has been recently introduced to solve optimization problems, e.g., building ML compilers and optimizing neural network architectures [30]. In particular, RL is used to select branching variables or find cutting planes in the Branch-and-cut method [31, 32, 33, 34, 35, 36, 37]. Besides, RL can also be applied to existing heuristics for MIPs to further increase the quality of solutions [38, 39, 40]. In fact, most state-of-the-art solutions for optimization problems often involve MIP or searching [41], but these methods are not directly appropriate for the VM rescheduling task due to their poor computation complexity. Although they are designed to accelerate MIPs, as shown in Section 3 even a state-of-the-art technique such as POP [42] fails to deliver a satisfying solution within the second-level time limits of the VM rescheduling task. While learning-based methods [15, 43] can meet the latency requirement by leveraging their generalization ability at deployment to avoid retraining, they do not involve techniques that are tailored for VMR, which we propose in VMR^{2}L.

A.5 PPO

The algorithm VMR^{2}L is developed based on the Proximal Policy Optimization (PPO) algorithm [13]. Recognized for its stability and robustness to various hyperparameters and network architectures

Algorithm 1: VMR^{2}L Overview.

```
Input: initial mapping dataset $\mathcal{D}$; initial VM actor, PM actor weights $\theta_{vm}, \theta_{pm}$.
for update_step $= 0, 1, \dots$ do
    // Collect B trajectories
    RB = [];      // Initialize replay buffer
    for batch_idx $= 0, 1, \dots, B$ do
        Sample initial state $s_0$ from $\mathcal{D}$;
        for migration_step $t = 0, 1, \dots, MNL - 1$ do
            $\log p_{vm,t} \leftarrow \pi(a|s_t, \theta_{vm})$;
            $a_{vm,t} \leftarrow p_{vm,t}$.sample();
            $\log p_{pm,t} \leftarrow \pi(a|s_t, a_{vm,t}, \theta_{pm})$;
            $p_{pm,t}$[Incompatible($a_{vm,t}, s_t$)] $= 0$;            // Mask out all incompatible PMs
            $a_{pm,t} \leftarrow p_{pm,t}$.sample();
            $s_{t+1}, R_t \leftarrow$ Simulator($a_{vm,t}$, $a_{pm,t}$);
            RB.add($s_t, \log p_{\cdot,t}, a_{\cdot,t}, R_t$);
        Compute advantage estimate $\tilde{A}_t$ for each entry in RB [47];
    Update $\theta_{vm}, \theta_{pm}$ via policy gradient with RB [13];
```

[13], PPO has exhibited superior performance compared to Natural Policy Gradients (NPG) [44] and Trust Region Policy Optimization (TRPO) [45], and exhibited less bias compared to Q-learning [46]. Despite PPO's high sample complexity, it becomes less of a concern due to our cost-effective simulator as detailed in Section C.1. In VMR^{2}L, we employ two actors — a VM actor $\pi\left(a_{vm}|s;\theta_{vm}\right)$ and a PM actor $\pi\left(a_{pm}|s, VM;\theta_{pm}\right)$. Given the current state encoding s, the VM actor samples the next-step vm action, denoted as a_{vm}. Based on both s and a_{vm}, the PM actor generates the pm distribution. We let the probabilities of all PMs that are incompatible with a_{vm} be zeros, and sample a_{pm} from the resulting distribution, which is the destination PM.

Below, we provide a brief overview for the training algorithm, with the complete pseudocode shown in Algorithm 1. The algorithm first initializes the parameters of the VM actor, the PM actor, and the critic all with random weights. To simplify the implementation, we incorporate the critic network into the VM actor by appending a special VM with all '-1' features and using its output as the critic score.

Each episode is made up of *MNL* steps, with each step representing a rescheduling action. For each training episode, the initial state s_0 is randomly sampled from the set of all vm-pm mappings available for training. At each step t in the episode, the VM and PM actors collaboratively reschedule one VM from its source PM to a new destination PM, which updates the current vm-pm mapping to get the next state s_{t+1} (Line 8-13). The episode is terminated after MNL migration steps.

Upon the completion of each episode, we compute the Generalized Advantage Estimate for step k (GAE$_k$) [47] by

$$GAE_k = r_k + \gamma \cdot v_{k+1} - v_k + \gamma \cdot \lambda \cdot GAE_{k+1}, \tag{10}$$

where r_i, v_i represent the reward and the critic's output at step i respectively. γ is the discount factor and λ is a smoothing parameter for variance reduction. The critic gradient loss is defined as the mean-square error between the reward to go and the critic's output, v, across the batch. The reward-to-go is calculated by applying the discount factor to the intermediate rewards.

Note that the parameters of the critic is updated together with the parameters of the actor and critic networks, with the loss term being a weighted sum of the mean-square error for the reward-to-go estimate and the policy gradient loss for the two actors.

B Sparse Attention Details

In this section, we delve into the detailed formulation of *sparse attention*.

B.1 Why We Need Tree-level Features?

Consider a PM with 2 CPUs left. It contains a VM with 4 CPUs and another VM with 2 CPUs. Suppose a second PM has a fragment size of 8 while hosting a VM with 8 CPUs. In order to minimize the total 16-core fragments, an ideal approach would be to first remove the two VMs with 2 and 4 CPUs from the first PM, and then reassign the VM with 8 CPUs from the second PM to the first PM. However, if we merely include the source PM's features in each of the VM's features and feed them into the vanilla transformer model, there will not be sufficient information for the two actors to take the above actions. Instead, each VM must also be aware of the other VMs that are hosted on the same PM, which is not possible in the vanilla transformer model. In fact, each PM can be viewed as a tree of depth one, where the PM acts as the root node and the VMs it hosts act as the leaf nodes. In order to allow every VM to recognize which other VMs are hosted on the same PM, we propose to include an additional stage of *sparse local-attention* within each PM tree, i.e., we only allow PMs and VMs to attend to each other if and only if they belong to the same tree.

B.2 Conceptual Overview.

Fundamentally, attention can be understood as a trainable dictionary involving queries, keys, and values. Given an embedding vector (a VM or a PM) that requires an update, and a set of reference vectors (selected VMs and/or PMs), we project the vector-to-be-updated into a query vector. Concurrently, we project all reference vectors into key and value vectors. We then compute the similarity score between the query vector and each key vector. Based on these scores, we update the target embedding vector as the weighted sum of the corresponding values.

The term "VM self-attention" refers to the process of updating each VM's embedding vector using all VM's embedding vectors (including its own) as a reference. "PM self-attention" operates similarly for PMs. "VM-PM attention" involves updating each VM's embedding vector using all PM's embedding vectors as references. Lastly, the proposed tree-level *sparse attention* involves updating each VM or PM using only other VMs or PMs within the same tree—that is, those affiliated with the same PM.

B.3 Attention Formulation.

Formally, let $\mathcal{V} = \{v_i\}$ and $\mathcal{P} = \{p_i\}$ denote the set of feature vectors for each VM and each PM, respectively. Consider an embedding vector $x_i \in \mathbb{R}^{d_{\text{model}}}$ that we aim to update, which is projected linearly[10] from $v \in \mathcal{V}$ or $p \in \mathcal{P}$. Let $(y_1, \cdots, y_n)$ be a set of reference vectors that could be a combination of embeddings of $v \in \mathcal{V}$ and $p \in \mathcal{P}$, including x_i itself. Instead of directly operating in the feature space $\mathbb{R}^{d_{\text{model}}}$, we project these vectors into an embedding space $\mathbb{R}^{d_k}$.

An attention function updates the target vector x_i by first projecting it into a query $Q_i = x_i W^Q$ using a linear transformation, where $W^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ is a learnable weight matrix. Similarly, each reference vector y_j is projected into a key $K_j = y_j W^K$ and a value $V_j = y_j W^V$, where $W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$[11]. Note that we can perform the attention function on a set of queries in parallel by stacking Q_i's into a larger matrix Q[12].

The compatibility function, measuring the similarity between Q_i and K_j, is typically implemented as the dot product of these two vectors. However, when d_k is large, the magnitude of dot products tends to increase, leading to a high similarity for only a few keys and a marginal similarity for the rest of the references. This has been known to cause extremely small gradients [14]. To address this, we scale the dot product by $\frac{1}{\sqrt{d_k}}$,

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V. \tag{11}$$

[10]A linear projection is necessary since v and p can have different number of features.

[11]W^V can have different dimensions than W^K. We keep them the same here for simplicity.

[12]As we use the same W^Q, W^K, Q^V for all vectors, the total number of weight parameters remains independent of the number of VMs or PMs in the problem, making this approach suitable for scaling to large data centers.

For the proposed *sparse attention*, we introduce an additional mask M. Here, $M_{i,j} = -\infty$ if x_i and y_j are not part of the same PM tree and zero otherwise.

$$\text{Sparse-Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V. \tag{12}$$

B.4 Attention Block.

In actual applications, we often utilize multiple sets of W^Q, W^K, and W^V. Each set, referred to as a single attention head, enables the model to extract pertinent information from a distinct representation space. We combine the resulting matrices from each attention head in multi-head attention by concatenation.

$$\text{Multi-Head}(Q, K, V) = \text{Concat}\left(\text{head}_1, \cdots, \text{head}_h\right)W^O, \tag{13}$$

where head_i represents the output of each attention head from Equation 12 using W_i^Q, W_i^K, and W_i^V. $W^O \in \mathbb{R}^{h \cdot d_k \times d\text{model}}$ is an additional learnable weight parameter.

Within each attention block, we consecutively execute the process for tree-level *sparse attention*, VM/PM self-attention, and VM-PM attention. Following these three attention submodules, we further process each updated embedding vector with a fully connected feed-forward network, which operates independently on each embedding vector.

$$\text{FFN}(x_i) = \text{GELU}\left(xW_1 + b_1\right)W_2 + b_2, \tag{14}$$

where GELU refers to the Gaussian Error Linear Units activation function [48]. The output of this process is then supplied to the next attention block, and the process is iteratively repeated. For more in-depth details regarding the attention module, we encourage readers to consult the paper by Vaswani et al. [14].

B.5 Architecture Overview

To better incorporate the proposed sparse local-attention module, we modify the vanilla transformer architecture as follows. The model is composed of several attention blocks, where each block includes three stages as shown in Fig. 4:

1. All PMs and VMs exchange information if they belong to the same tree via sparse local-attention.
2. Each PM attends to other PMs' updated embeddings and each VM attends to other VMs' updated embeddings with self-attention.
3. The new VM embeddings attend to the new PM embeddings through VM-PM attention.

After the three stages, each machine is further processed by two dense layers and layer norm [49]. The updated embeddings are then fed into the next block and the process repeats. Finally, the VM embeddings from the last block are linearly projected into a set of logits followed by Softmax [50] to generate the probability of selecting each VM.

The architecture incorporates the remaining MNL steps as supplementary input for both the VM and PM actors, processed concurrently with the PMs. This inclusion is pivotal for reinforcement learning in scenarios with fixed-length episodes as demonstrated in prior studies[51]. As for the critic, we add a node with all -1's to be processed together with the VMs. The output embedding from the final block is linearly projected into the critic score. Note that the PM embeddings are updated block-by-block together with the VM embeddings, as it encourages the PMs and VMs to better coordinate and update gradually from low-level to high-level [52].

As for the PM actor, we adapt the vanilla encoder-decoder transformer, since we can directly inject the relational information by including the updated VM and PM embeddings from the VM actor as input. We only feed in the selected VM candidate to the encoder, while the decoder still takes in all PMs. Additionally, we also add the VM-PM attention score from stage 3 for the selected VM, since the score indicates which PMs the VM actor attends to and encourages the two actors to better coordinate. The output embeddings of each PM is linearly projected into a logit. Based on the selected VM, we mask out all the illegal PMs by setting their logits to be $-\infty$. The remaining logits are translated into the probability of selecting each PM as the destination PM.

C Experiment Details

We conduct extensive experiments to answer:

- How far is VMR^{2}L from the optimal solution? (§ C.3)
- Can VMR^{2}L optimize a different objective: given a target FR, minimize the number of VM migrations? (§ C.4)
- How well does VMR^{2}L generalize to different MNLs? (§ C.5)
- How well does VMR^{2}L generalize to larger clusters with more VMs and PMs? (§ C.6)
- How much performance gain does *sparse attention* provide? (§ C.7)
- Where the improvements come from intuitively? (§ C.8)

C.1 Simulator and Datasets

Simulator. While DRL can be very powerful, its main drawback is the amount of training data required [53, 54]. In light of this, we design a simulator for the VM rescheduling task. The simulator follows the OpenAI Gym environments [55] including specific file hierarchy and function abstractions. Given an existing VM-PM mapping and a rescheduling action, we can directly calculate the change in FR caused by the action. Thus, during training, VMR^{2}L only needs to interact with the simulator instead of with the real environment, which drastically lowers the amount of real-world data required to train the agent.

Datasets. We have two seed initial mappings from an industry-scaled real cloud data center – one medium dataset with 2089 VMs and 280 PMs, and one large dataset with 4546 VMs and 1176 PMs. Note that the RL agent must be able to generalize to VM-PM mappings unseen during training and a dynamic number of VMs at deployment time. To better evaluate the agent's performance under various initial mappings, we generate 4400 initial mappings with different numbers of VMs for both the Medium and the Large datasets. Each mapping is generated by removing the existing VMs on each PM and randomly scheduling some of them to any PM that can fit them. We leverage 4000 datasets for training, 200 datasets for both validation and test. Both the simulator and datasets are available to the research community.

Algorithm Specifics. We implement VMR^{2}L based on the CleanRL framework [17] using PPO as the backbone [13]. VMR^{2}L contains about 8.5K lines of Python code. The overall framework is implemented using PyTorch [56]. The number of model parameters is independent of the number of VMs or PMs, allowing it to scale to large data centers. The hyperparameters are listed in Table 2. VMR^{2}L is lightweight and only has two attention blocks.

Experiment Setup. All models are trained on a Linux server using eight CPUs (Intel Xeon E5) and one GPU (NVIDIA RTX 3090). VMR^{2}L with sparse attention and without sparse attention takes 92h and 48h to train, respectively. We report the average over 3-5 runs with different random seeds and show the confidence intervals in the convergence plots.

Table 2: Hyperparameters of VMR^{2}L.

RL Parameters	Value	General Parameters	Value
Clip_coefficient	0.1	Total_iterations	4e6
Discount_factor	0.99	Learning_rate	2.5e-4
PPO update_epochs	8	Attention_blocks	2
PPO minibatches	128	Attention_heads	2
GAE_Lambda	0.95	Transformer d_{ff}	64
Value_coefficient	1e-2	Transformer d_{hidden}	32

C.2 Existing Baseline Algorithms

Existing baselines can be summarized into six categories: heuristics (e.g., greedy, α-VBPP), optimization algorithms (e.g., MIP), approximate algorithms (e.g., POP), search-based algorithms (e.g., MCTS), deep learning-based (e.g., Decima), and hybrid methods (e.g., NeuPlan). We compare with at least one representative algorithm from each category.

MIP Algorithm: We formulate the VM rescheduling problem as an optimization problem (Appendix A.2) and solve this problem with an off-the-shelf MIP solver such as Gurobi [9] and CPLEX [10], which can find an optimal solution through algorithms of branch & bound, cutting planes, etc. In our experiments, we use the former.

Greedy Algorithm: To obtain a feasible solution within a short time frame, greedy algorithms [57] are often used. They normally include two stages: filtering and scoring. In the filtering stage, we calculate the change in FR for each VM if it is removed from its source PM, and only select the VM candidate that corresponds to the most significant drop in FR. In the scoring stage, we calculate the change in FR if the selected VM is migrated to each of the eligible PMs. We then greedily assign the selected VM to the PM that leads to the largest drop in FR. The above two stages are repeated until the migration number limit is reached.

Vector Bin Packing Problem (α-VBPP): We generalize the VBPP [25] algorithm for initial scheduling to rescheduling. We first divide the entire episode into MNL/α stages. During each stage, we greedily remove α number of VMs that lead to the most fragments, and then apply VBPP to treat them as incoming VMs. We carefully tune α (10 in our case) to achieve the best FR reduction.

Partitioned Optimization Problems (POP): The POP method [42] solves the optimization problem formulated in Section A.2 by randomly splitting the problem into subproblems (each containing a subset of VMs and PMs), applying an MIP solver to each subproblem, and finally combining the results into a global solution. We perform a grid search for the best POP parameter that balances the FR and time on both the Medium and the Large datasets.

Monte-Carlo Tree Search (MCTS) [29]: As traditional search-based methods need to perform multiple rollouts during inference time to achieve a good performance, we use DDTS [29] to prune the search space.

Decima [43]: Decima uses RL and neural networks to learn the VM rescheduling algorithm. A graph neural network is leveraged to encode the VM and PM information in a set of graph embedding vectors to process a large amount of state information. Decima balances the size of the action space and the number of actions required by decomposing VM rescheduling decisions into a two-dimensional action, which outputs (i) the VM that needs to migrate, and (ii) an upper number of PM subsets to choose as the destination.

NeuPlan [41]: NeuPlan uses a two-stage hybrid approach to address MIP's scalability issue. In the first stage, an RL-based method takes in the problem in the form of a graph and generates the first few steps of VM rescheduling to prune the MNL search space. In the second stage, it uses a MIP solver to find the optimal VM migration given the remaining MNL. A relax factor β (30 in our case) is used to control the size of the MNL space to explore by MIP.

C.3 Performance on the Medium Dataset

Fig. 5 shows the FR and inference latency of all methods on the Medium dataset. VMR^{2}L achieves a lower FR compared to all baselines. Notably, VMR^{2}L is merely **7.14%** behind the optimal MIP solution (0.3079 vs. 0.2859) when $MNL = 50$. Meanwhile, VMR^{2}L can generate one trajectory within 1.1s, while MIP requires 50.55 minutes to provide the near-optimal solution. It is worth noting that with higher MNLs, the performance gap between VMR^{2}L and MIP does not increase as significantly as compared to other baselines.

α-VBPP only removes α number of the worst VMs for each stage based on a single timestep, failing to consider future opportunities to replace them back, which leads to its inferior performance. POP fails to achieve good performance since it still relies on MIPs to solve each subproblem. To meet the second-level latency requirement, we must divide the problem into many subproblems, causing its solutions to be only locally optimal. On the other hand, Decima reduces the large action space by limiting the PM actor to only select from a subset of PMs, but the subsampling of PMs is completely random, as opposed to our solution. While MCTS with DDTS uses neural networks to prune the search space, it still requires a significant number of rollouts to achieve stable performance. Lastly, NeuPlan also fails to deliver a satisfying solution for high MNLs, since it can only use MIP to solve a small number of steps in order to meet the latency requirement.

To summarize, algorithms that involve MIP or searching often fail to deliver a satisfying solution under the strict latency requirement. Heuristic methods are fast but are also suboptimal. Deep learning-based methods can meet the latency requirement since the models do not need to be retrained at inference time, but are difficult to train without the set of customized techniques we proposed for VMR.

C.4 A Different Objective: Minimize Number of VM Migrations Given FR Goals

VMR^{2}L's flexibility enables it to learn different policies depending on the high-level objective. We now consider a new objective: we would like to minimize the number of VM migrations to reach a given FR goal, to reduce migration costs. To support this objective, we simply modify the original reward function as follows:

$$R_{fr} = (S_{\text{before, src}} - S_{\text{after, src}}) + (S_{\text{before, dest}} - S_{\text{after, dest}}), \tag{15}$$

$$R = \begin{cases} -1 + R_{fr}, & FR > FR_{\text{Goal}}, \\ 10 + R_{fr}, & FR \leq FR_{\text{Goal}}. \end{cases} \tag{16}$$

On top of the original reward, we add a penalty of -1 if the FR falls below the goal as it indicates additional VM migrations are required, and a bonus of +10 if VMR^{2}L reaches the goal. In Fig. 8, the top subfigure shows the number of migration steps, while the bottom subfigure shows the achieved FR, both sharing the x-axis as different FR goals. In general, the number of VM migrations required by GA, VMR^{2}L, and MIP increase with lower FR goals (lower FR is more challenging). On average, MIP and VMR^{2}L achieve 14.77% and 11.11% fewer MNLs than GA, respectively. VMR^{2}L requires only 3.66% more VM migrations, but with millisecond-level solution time. Note that at the FR goal of 0.55, the average initial FR of the test set is 0.53, so no VM migrations are required. Also, no method can achieve the FR goal of 0.25, since the optimal FR is 0.2859 at MNL 50.

C.5 Generalizing to Different MNLs

In practical scenarios, MNL often fluctuates due to varying business needs, such as when VMR is performed. We show that training a single VMR^{2}L agent with $MNL = 50$ can yield effective results across a range of MNLs $\in \{10, 20, 30, 40, 50\}$. To compare, we train a separate VMR^{2}L agent for each MNL, denoted as VMR^{2}L $_{\text{SEP}}$. As shown in Fig. 9, VMR^{2}L performs only marginally worse than VMR^{2}L $_{\text{SEP}}$ with an average FR performance gap of 1.16%. This suggests that the VMR^{2}L agent trained with a large MNL can be readily applied to tasks with smaller MNLs. It avoids the overhead of maintaining a separate VMR^{2}L agent for each MNL.

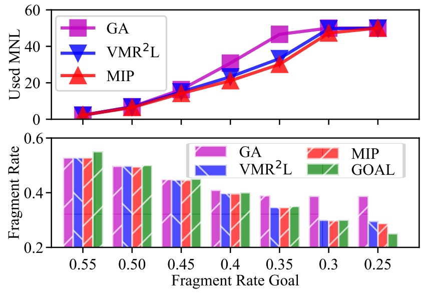

Figure 8: MNL performance under different FR goals.

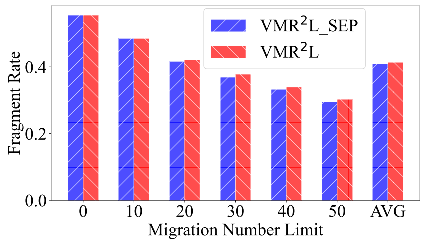

Figure 9: Gap between VMR^{2}L and VMR^{2}L $_{\text{SEP}}$.

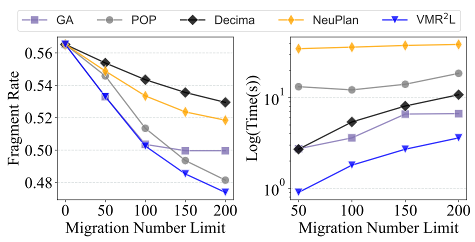

Figure 10: Fragment rate and inference time on the Large dataset.

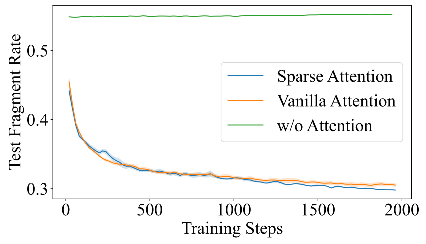

Figure 11: An ablation study on the proposed sparse-attention module.

C.6 Scalability to the Large Dataset

To see how VMR^{2}L scales to a larger cluster, we conduct experiments on the Large dataset with 4546 VMs and 1176 PMs. Fig. 10 shows the performance against different baselines when MNL varies from 50 to 200. MIP is not included here since it takes more than an hour to solve a single migration path. Similar to the Medium dataset, VMR^{2}L achieves lower FRs than the baselines, with an inference time of 3.8s to solve one mapping.

C.7 Ablation on Sparse Attention

We compare against two baselines. *w/o Attention* uses a multilayer perceptron (MLP) [58] as the feature extraction module. MLP concatenates features of all PMs and VMs and thus requires much more trainable parameters that scale linearly with the number of machines in the system. *Vanilla Attention* has fewer parameters as it shares a single small embedding network for all PMs and a second small embedding network for all VMs, but it uses the original encoder-decoder transformer [14] without attending to tree-level features. Fig. 11 shows that MLP fails to converge due to its large number of trainable parameters. As training progresses, *Sparse Attention* learns to capture relational features unique to VMR and gradually outperforms *Vanilla Attention*. We show a case study of how such relational information can benefit VMR in Section C.8.

C.8 A Case Study

Intuitively, where do the improvements of VMR^{2}L come from? To answer this question, we build a tool to visualize which VM is being migrated at each step. We randomly select one mapping from 200 test mappings on the Medium dataset and analyze how VMR^{2}L reduces FR when $MNL = 50$[13].

VMR^{2}L optimizes FR from a global point of view. We analyze the three PMs involved during steps 38-40, where each PM has two NUMAs as represented by the two side-by-side bars. At step 38, VMR^{2}L removes a VM with 4 CPUs to achieve zero fragments on the destination NUMA. However, this also creates four fragments on the source NUMA, leading to a net reward of zero at this step. Then at step 39, it reschedules another VM with 4 CPUs to achieve zero fragments on both the source and the destination NUMAs. Although the agent receives a reward of zero at step 38, *sparse attention* allows the agent to realize that there were two VMs with 4 CPUs on the source NUMA, so the agent can reassign the second VM with 4 CPUs elsewhere in order to achieve zero fragments on the source NUMA as well.

This example shows that VMR^{2}L is able to sacrifice immediate rewards for long-term FR performance due to the cumulative reward design in reinforcement learning.

[13]The original GiF animation can be found at `https://anonymous.4open.science/r/VMR2L-BEA6/readme-figures/visual-traj.gif`. Users can use the script ./eval_plot_steps.py to better interpret and analyze the actions taken by the agent.

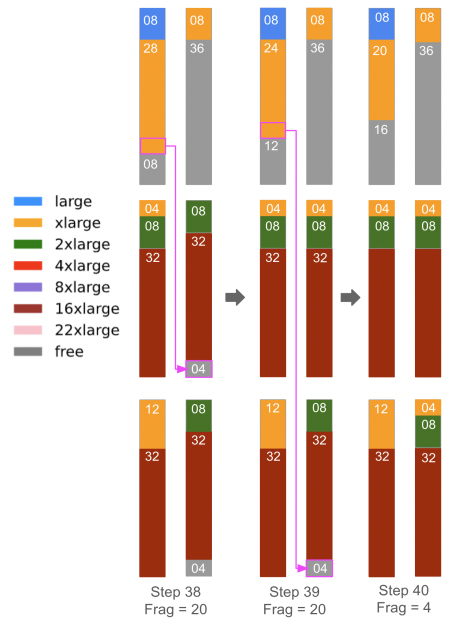

Figure 12: VM-PM Migration Details.