Maestro: VM memory overcommit balancing platform

Anonymous Author[s]

1 Introduction

Modern online services are memory-intensive and often hosted in the cloud. Due to the high cost of memory resources, cloud providers are looking for ways to reduce memory requirements. Users provision virtual machines (VMs) with memory to handle peaks, thus resorting to large instance types. Consequently, VMs frequently under-utilize memory. For example, in Google data centers, approximately 30% of server memory remains unused for minutes and is considered cold [13]. This indicates that cold memory can be reclaimed and repurposed.

Operating systems use swapping to move cold memory to a more cost-effective but slower medium. Current systems typically use 4KiB granularity for swapping, as it offers the lowest latency on page-fault. In contrast, virtualization stacks perform better on in-memory Hugepages (2MiB), which shorten the nested page walk time. 2MiB swapping has been proposed [5] but is not supported by mainstream kernels such as Linux.

However, we observe that loading a 2MiB block on a modern server grade SSD does not take significantly longer than 4KiB. Specifically, a 2MiB load takes about 10× longer than a 4KiB load, while loading 512× as much data. Furthermore, certain workloads have a regular enough access pattern that bringing in 2MiB pages does not harm performance compared to 4KiB. Other workloads with a more random access pattern suffer significantly from 2MiB swapping. This leads us to revisit memory assignment policies [9, 15, 17].

To develop custom balancing policies, we introduce Maestro: A platform designed to transparently balance the memory of multiple VMs. Maestro, running in userspace, collects information from the system such as page-fault count and working set estimate, exposes this information to a flexible policy, which in turn adjusts the VMs memory limit. Currently, Maestro works with a user-space VM memory manager that uses SPDK [1] for swapping [8].

To make no assumptions about our guests, we focus on non-cooperative swapping. This is necessary as fallback even in cooperative scenarios. Also, such methods can achieve comparable performance [2]. Nevertheless, we plan to include cooperative swapping by exposing (for example) a guest memory balloon to policies.

2 A first policy

We present a first policy for Maestro that is adapted to 2MiB swapping, it uses the WSS estimate as well as fixed window of past page fault counts, which serve as a proxy of the workloads sensitivity to the memory limit.

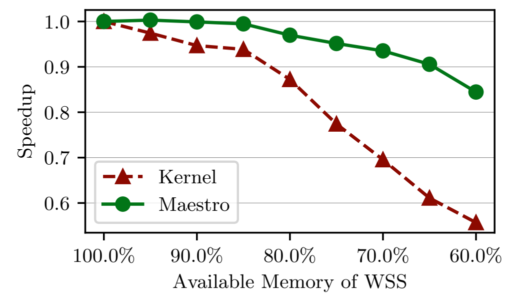

Figure 1. Average performance impact with multiple VMs as available memory is reduced

The policy determines the VM memory **limit** by minimizing $\frac{\text{wss+pf\_weight}}{\textbf{limit}}$ for each VM, favoring a higher memory allocation. The WSS estimate is calculated by periodically scanning idle EPT pages [9]. Considering all VMs, we minimize the following expression:

$$\min\left(\sum_{i=1}^{n}\left(\frac{\text{wss}_i(t)+\text{pf\_weight}_i(t)}{\textbf{limit}_i\textbf{(t)}}\right)^2\right) \quad (1)$$

While having the following constraints:

$$\sum_{i=1}^{n}\textbf{limit}_i\textbf{(t)} \le \text{host\_mem\_total} \quad (2)$$

$$\text{min\_mem} \le \textbf{limit}_i\textbf{(t)} \le \text{vm\_mem\_max}_i \quad (3)$$

These limits can be computed efficiently using Lagrange Multipliers. We raise the fraction to the power of 2 in equation 1 to amplify the impact of larger fractions. Since WSS estimation is computationally expensive and cannot be performed frequently [9], the page fault weight is essential for supporting latency-sensitive workloads.

We implement the page fault weight using a PID controller, where the error is the page fault count within a time window of W seconds, and all coefficients are set to 1:

$$\Delta\text{pf}_i(t) = \text{pf}_i(t) - \text{pf}_i(t-W) \quad (4)$$

$$\text{u}_i(t) = \Delta\text{pf}_i(t) + \int_{t-W}^{t}\text{pf}_i(\tau) - \text{pf}_i(t-W)d\tau + \frac{\Delta\text{pf}_i(t)}{W} \quad (5)$$

$$\text{pf\_weight}_i(t) = u_i(t) \times \text{page\_size}_i \quad (6)$$

3 Evaluation

We evaluate the performance of Maestro and Linux in managing multiple VMs under different levels of overcommitment. For Linux, we employ Multi-Gen LRU (MG-LRU) [3] with efficient secondary MMU aging [10] and THP [4]. Swapping and balancing are handled by placing all VMs in a single CGroup with a total memory limit, which forces Linux to swap memory under pressure. In contrast, Maestro uses the policy presented in section 2 to determine memory limits for each VM, aiming to minimize performance impact. Maestro guests perform strict 2MiB paging and swapping that we achieve with a user-space swapping framework [8].

Our setup involves launching 8 VMs, with two VMs assigned to each of the following workloads:

TPC-H [14] is benchmarked with a scale factor of 1. The TPC-H generated database is imported into MariaDB [6], with its contents stored in an in-memory table.

G500 [7] The graph500 reference implementation. We use scale 23. To decrease the execution time, we set the benchmark to do 2 BFS and 2 SSSP phases.

Matmul Matrix multiplication using OpenBLAS [16] dgemm. We use double precision matrices of size 14336×14336 for one iteration.

Redis [11] is benchmarked using *memtier* [12]. The database is initialized with a 2GiB dataset using small keys and 1kB data entries. After initializing the database, we execute the following access patterns in the following sequence: Gauss, Random, Sequential.

Each VM runs its workload for 30 minutes, during which we measure the average iteration time as a performance metric. We then calculate the average speedup compared to a non-swapping baseline. The overall speedup is determined using the geometric mean of the speedup across all VMs while the system is under pressure.

We compare these two approaches under increasing memory pressure (see fig. 1). Initially, we provide enough total system memory to accommodate all workloads, then gradually reduce the available system memory to less than the sum of the workloads' working set sizes, forcing the balancers to distribute memory between VMs.

Overall, we find that using Maestro for mixed workload VMs led to a 30% performance improvement compared to the kernel's MG-LRU-based memory balancing. The policy leverages information such as the WSS, which serves as the baseline for required memory for each guest, and utilizes the page fault weight to prioritize memory for guests that generate more page faults.

4 Conclusion

We observe some trends in hardware that make strict-2MiB swapping attractive: SSDs are getting faster and VM workloads prefer to be backed by Hugepages. Workloads exhibit different sensitivity to Hugepage swapping, which leads us to revisit memory balancing policies with Maestro, and we determine a first promising policy for a workload mix. We plan on generalizing this by evaluating different workloads and developing new policies.

References

[1] [n. d.]. Storage Performance Development Kit. https://spdk.io/ Accessed: 2024-02-05.

[2] Nadav Amit, Dan Tsafrir, and Assaf Schuster. 2014. VSwapper: a memory swapper for virtualized environments. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 349–366. https://doi.org/10.1145/2541940.2541969

[3] The Linux Kernel Development Community. 2025. *Multi-Gen LRU*. https://docs.kernel.org/admin-guide/mm/multigen_lru.html Accessed: 2025-01-30.

[4] The Linux Kernel Development Community. 2025. *Transparent Hugepage Support*. https://www.kernel.org/doc/html/next/admin-guide/mm/transhuge.html Accessed: 2025-01-30.

[5] Jonathan Corbet. 2018. LWN: The final step for huge-page swapping. https://lwn.net/Articles/758677/

[6] MariaDB Corporation. 2025. *MariaDB*. https://mariadb.org/

[7] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19, 45-74 (2010), 22.

[8] Milan Pandurov, Lukas Humbel, Dmitry Sepp, Adamos Ttofari, Leon Thomm, Do Le Quoc, Siddharth Chandrasekaran, Sharan Santhanam, Chuan Ye, Shai Bergman, Wei Wang, Sven Lundgren, Konstantinos Sagonas, and Alberto Ros. 2024. Flexible Swapping for the Cloud. arXiv:2409.13327 [cs.DC] https://arxiv.org/abs/2409.13327

[9] SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. 2022. DAOS: Data Access-aware Operating System. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (Minneapolis, MN, USA) *(HPDC '22)*. Association for Computing Machinery, New York, NY, USA, 4–15. https://doi.org/10.1145/3502181.3531466

[10] Axel Rasmussen, Guru Anbalagane, Wei Xu, and Yuanchu Xie. 2024. Multi-Gen LRU updates. In *Linux Plumbers Conference 2024*. Vienna, Austria. https://lpc.events/event/18/contributions/1781/

[11] Redis. 2025. *Redis*. https://redis.io Accessed: Nov 2024.

[12] RedisLabs. 2025. *Memtier Benchmark*. https://github.com/RedisLabs/memtier_benchmark Accessed: Jan 2025.

[13] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. AIFM:High-Performance,Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 315–332.

[14] Transaction Processing Performance Council. 2025. *TPC-H Benchmark*. http://www.tpc.org/tpch/

[15] Zhigang Wang, Xiaolin Wang, Fang Hou, Yingwei Luo, and Zhenlin Wang. 2016. Dynamic Memory Balancing for Virtualization. *ACM Trans. Archit. Code Optim.* 13, 1, Article 2 (March 2016), 25 pages. https://doi.org/10.1145/2851501

[16] Zhang Xianyi and Martin Kroeker. 2024. *OpenBLAS*. https://www.openblas.net

[17] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 37–56. https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong