Improving gVisor Memory Subsystem Performance

Xiaohan Fu, Bili Dong, John Hwang
University of California, San Diego

Abstract

Among current virtualization technologies, containers are often closely connected with high performance, low overhead, as well as weak security isolation. gVisor is a technology developed by Google which attempts to improve container security through sandboxing. In this project, we analyzed the performance of the gVisor memory management subsystem, starting from benchmarking malloc and ending up focusing on mmap. We further profiled mmap performance within gVisor and identify its bottlenecks. We proposed an optimization in free page searching algorithm of virtual memory space within gVisor instead of the original plain linear search. This optimization shortens the search time to $O(\log n)$ with a close to constant performance in practice, in contrast to the original $O(n)$ linear scan which scales poorly with the number of existing mappings. This patch has been submitted to the official gVisor repository as a pull request and finally merged to the master branch[1].

1 Introduction

OS-level virtualization thrives in recent years with the popularization of commercially successful containers like Docker, but security has remained a major issue since the birth of this technology [2]. gVisor, open sourced by Google in 2018 [4], is designed to address this container security issue. Unfortunately, there exists an unavoidable trade-off between security and performance, and gVisor is no exception. Previous studies have revealed gVisor's performance penalties in multiple subsystems including memory management, network stack, and others [5]. This work inspired us to attempt optimizations in these subsystems.

Before diving into details, it helps to categorize the performance overhead of gVisor as structural costs and implementation costs [3]. Structural costs are those imposed by high level design choices, like the general virtualization architecture and the adopted programming language. Implementation costs are those related to the actual implementations, like the system calls in the libOS. Reducing structural costs requires extensive understanding of the whole system, and an effort far larger than we could afford. For our purposes, it's only realistic to focus on reducing implementation costs.

Therefore in this project, we aim to identify and improve the performance of gVisor from the perspective of implementation costs without sacrificing its security. Specifically, we focus on improving the memory subsystem performance by optimizing the mmap syscall within gVisor, which is one of the major bottlenecks in memory operations. Alternatively, we also identified netstack related improvements for non-datacenter use cases as a backup project. Fortunately, the mmap project went well, and we never need to look further into the netstack project.

The motivation for us to study and optimize the memory subsystem of gVisor is straightforward. Memory management is one of the most fundamental components of every operating system. gVisor, as a libOS adding one more layer of indirection for security, will definitely have management and implementation overhead in this subsystem. As any container's workload will inevitably involve a large number of memory operations from the its startup to the initiation of programs within, even small improvement in performance of the memory management subsystem would have an obvious and positive impact on the whole system performance. Hence memory management subsystem is a very cost-effective choice of entry point to improve the overall efficiency of gVisor.

The rest of this paper is organized consistent with the workflow we conducted this project, as following. We describe how we narrow down the optimization target from the whole memory management subsystem to a single mmap syscall specifically through various experiments and benchmarks in Section 2. After that, significant code reading and literature research is done to understand how mmap is implemented within gVisor and how it diverges from regular linux, which is explained in Section 3. Then we introduce the method we used to identify the bottleneck within a mmap syscall in Section 4. Finally, after figuring out the dominating source

[1] https://github.com/google/gvisor/commit/059879e

of overhead, we got our hands dirty with the source code of gVisor and optimized the free-space searching algorithm in virtual memory space. The implementation of our optimization is explained briefly in Section 5. For curious readers, more details can be found by reading the commit itself. Evaluation of the cost and effect of this optimization is shown in Section 6. The Challenges, unexpected behaviors, and experiences we faced and obtained throughout this process are shared in Section 7.

2 Problem Narrowing and Definition

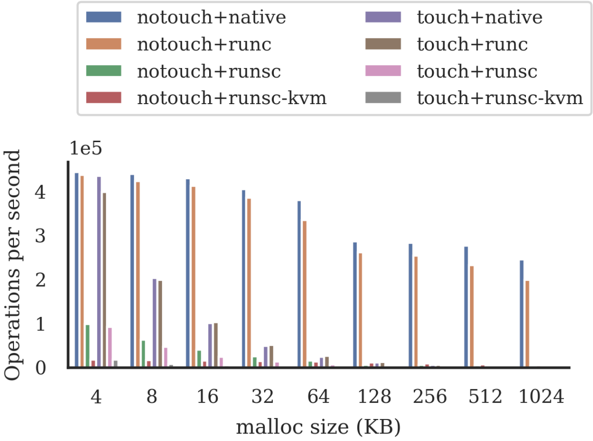

Figure 1: Performance of malloc in the nofree case. This figure is comparable to Figure 4 in [5], and shows similar trends.

As gVisor is under active development, previous research on gVisor performance [5] might be outdated already. So we first reproduced and extended their malloc benchmark to see if their results still hold. These benchmarks are performed on a Google Cloud Platform (GCP) n1-standard-4 instance with 4 vCPUs and 15 GB memory, running a Ubuntu 18.04 OS. (The same measurement is done on a bare metal as well for comparison).

For each benchmark, we call malloc repetitively and measure the number of malloc operations per second. We ran each benchmark with 4 different runtimes: **native** is the native Linux runtime, **runc** is the normal Docker runtime, **runsc** is the gVisor runtime in ptrace mode, **runsc-kvm** is the gVisor runtime in KVM mode. We performed 2 types of experiments characterized by **touch/notouch**. In the context of this paper, we refer to touch as accessing the memory immediately after allocation. To stress the memory system, we do not free the memory after allocation. This is the same setup as [5].

The results are presented in Figure 1. In general, we see the same trends as in [5]. For the notouch case, the native and runc results are fairly constant when the malloc size is in between 4KB to 64KB, and then decrease from 128KB and keep constant again, which can be explained by switch from brk to mmap (proved by strace log). However, the throughput inside gVisor is already much lower than runc or native when in 4KB to 64KB region and comes incomparable to runc or native when the allocation size is larger or equal to 128KB. This gives us the intuition that both brk and mmap in sentry have degraded performance with the latter case being extremely severe.

From these experiments we could see that there is a large performance overhead in gVisor when mmap is involved for memory allocation. So we decided to focus on mmap for further investigation and optimization. We performed similar benchmarks for mmap alone to further quantify its performance. These benchmarks are presented in Section 6 as they are also used as an evaluation method.

3 MMAP in Linux and gVisor

`mmap`, as described in its man page, "creates a new mapping in the virtual space of the calling process" [1]. Classically, there are 3 modes of `mmap` that in both linux and gVisor, as shared, private, and anonymous. Subtle differences between the linux and gVisor implementations cause the anonymous configuration to have significantly different performance overhead from the other configurations. The general workflow is described as follows.

3.1 Linux

In linux, the `mmap` call creates a mapping called the Virtual Memory Area (VMA) and typically points to some offset in a real file on disk or a zero-filled ephemeral file (anonymous mapping) as used by `malloc`.

Note that with the linux system call, the file mapped is not immediately put into the physical memory, which means the virtual to physical page mapping does not necessarily exist after the `mmap` call. Instead, this mapping occurs upon access to the allocated virtual address as a page fault, whose handler maps physical memory to the file descriptor of the VMA. Then, a page table entry which maps virtual addresses to physical addresses is computed based on the above two mappings.

To summarize, the mmap workflow can be expressed as 3 steps where steps 2 and 3 occur only after a page fault. This behavior is remains consistent among the previously mentioned configurations.

1. **Create VMA:** Maps virtual address to offset of file.

After triggered by a page fault

2. **Create filemap:** Maps file offset to physical address
3. **Create PTE:** Maps virtual address from 1. to physical address from 2.

3.2 gVisor

In gVisor, the difference in mmap workflow is in the form of two additional steps. First, the sentry intercepts `mmap` system calls and takes on the responsibility of mapping the virtual address to the file by creating a sentry VMA. Second, pgalloc is used to map the sentry file and offset to a temporary host memory file and offset. The workflow is summarized below.

1. **Create sentry VMA:** Maps virtual address to offset of file in sentry (instead of host kernel). <= createVMALocked()

After triggered by a sentry page fault (VA accessed for 1st time)

2. **Create sentry filemap:** pgalloc is used to map file and offset in sentry to file and offset on host. <= getPMAsLocked()
3. **Create host VMA:** Maps virtual address from 1. to file and offset on host from 2. by calling the host mmap syscall. <= mapASLocked()

After triggered by a host page fault (VA acessed for 2nd time)

4. **Create host filemap:** Maps file and offset on host to physical address.
5. **Create PTE:** Maps virtual address from 1. to physical address from 4.

Although steps 1 and 2 are unique to gVisor, steps 3-5 maps to steps 1-3 in a normal linux mmap call. Note step 1, 2 and 3 corresponds to createVMALocked(), getPMAsLocked() and mapASLocked() in gVisor respectively.

One optimization made by gVisor is to eagerly create host VMA for anonymous mmap, which means it will step through the first three steps even when there's no page fault at all. This optimization saves the overhead of one time switching to sentry to handle the page fault. Hence, anonymous mmap in gVisor performs two more steps compared to shared/private mmap when it's called. **This gives us the intuition that an anonymous mmap should be slower than file-backed mmap as it does more work.**

4 Identifying the Bottleneck of MMAP

After understanding the implementation of MMAP in gVisor, we tried to figure out the major source of overhead within it. Usually, implementation cost is particularly obvious under stress tests. Therefore we designed two groups of benchmarks to put the system under pressure: one is to test the average latency of a mmap call vs. total iterations, the other is to test the average latency of a mmap call vs. the mmap size. We refer to these benchmarks as Exp1 and Exp2 respectively. Note as shared and private mmap show same pattern in all benchmarks, we typically only shows the result for private one.

4.1 Average Latency vs. Total Iterations

The idea behind Exp1 is to see if number of existing allocated mappings has any effect on the average latency. If some implementation in gVisor scales with the total mappings, there must be some positive correlation. We fix the mmap size to be 4KB, 16KB and 64KB respectively and span the iteration from 100000 to 500000. These specific numbers allow sufficiently large test sizes and iterations within the limitations of our total memory. We perform the test on a Linux bare metal with 64GB ram and 3GHz 8-core CPU. We do the test for 12 trials with first 2 as warming up and 2 seconds internal warmup time in each trial.

The test result for anonymous (private) case is shown in Figure 2. **Both in gVisor (runsc-kvm) and in normal docker (runc) the average latency is nearly constant, which implies that there is no overhead that scales with increasing iterations or existing mappings in anonymous case.**

However, when we do the same benchmark for non-anonymous file-backed mmap, i.e. private or shared mmap, our test can't even run to completion. This shows 100000 iterations is far too large and we reduce our tests to 5000, 10000, and 25000 iterations instead. The result is shown in Figure 3. In contrast to the previous case, we see a clear pattern of the average latency increasing linearly with total iterations. **This implies the cost of each file-backed mmap is linearly related to the number of existing mappings ahead of it** and the reason our test can't scale to 100000 iterations is because the total time would thus be quadratic to total iterations. Also, it deserves to note that the latency is much larger than anonymous case, which contradicts our intuition in previous section because of this linear performance.

This is an interesting behavior which is likely to be caused by gVisor's implementation. To identify the parts and functions of gVisor that cause this pattern, we make use of perf, the linux performance analysis tool to observe the cpu consumption pattern by each function under different iterations' test. The flame graph generated by the perf data is shown in Figure 4 and Figure 5 for 10000 iterations and 25000 iterations respectively. It's clearly noted that the utilization of findAvailableLocked function increases from 40% of the total createVMALocked() in 10000 iterations case to 82% in 25000 iterations case. This hints us findAvailableLocked may be the source of overhead.

However, perf is based on CPU sampling which is not really accurate and can only show percentage change of CPU utilization. To have a more quantitative analysis, we add two pieces of time tracers assembly code to read the TSC register from CPU directly before and after the call of findAvailableLocked inside MMap() in gVisor to record the time consumption. This method gives minimal overhead and the most accurate result at unit of cycle compared to normal clock_get_time() syscall based time library functions. We obtain a chart of the average latency of n-th mmap call in cycle vs. n in Figure 6.

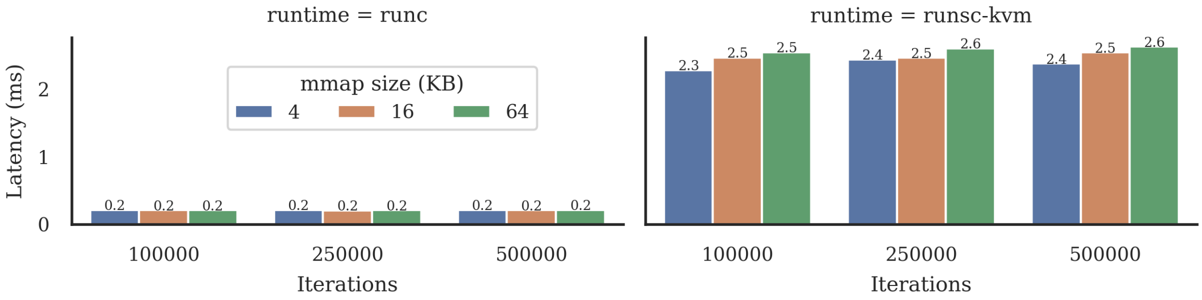

Figure 2: Anonymous mmap latency vs. iterations.

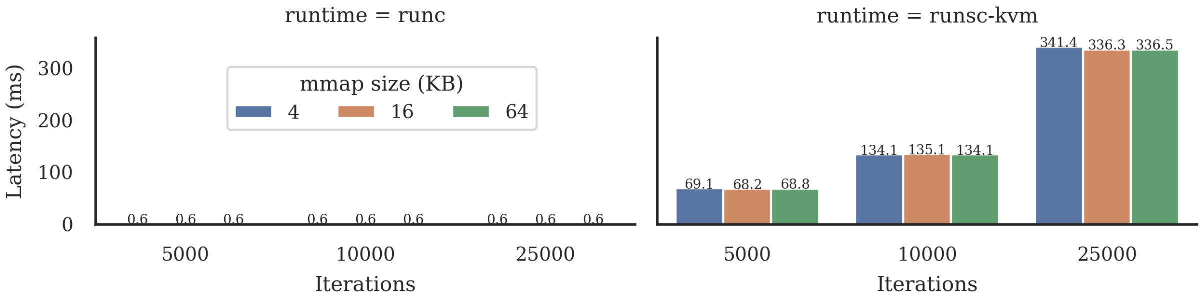

Figure 3: Private mmap latency vs. iterations.

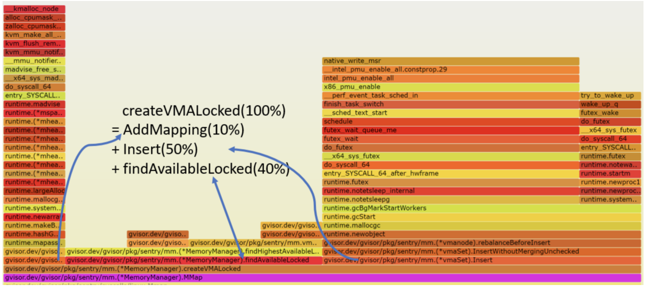

Figure 4: Flamegraph for createVMALocked with 10000 iterations.

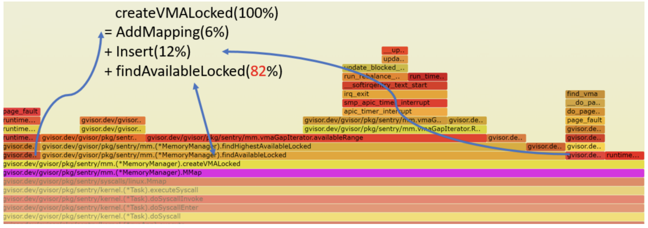

Figure 5: Flamegraph for createVMALocked with 25000 iterations.

A clear and obvious linear relation is shown.

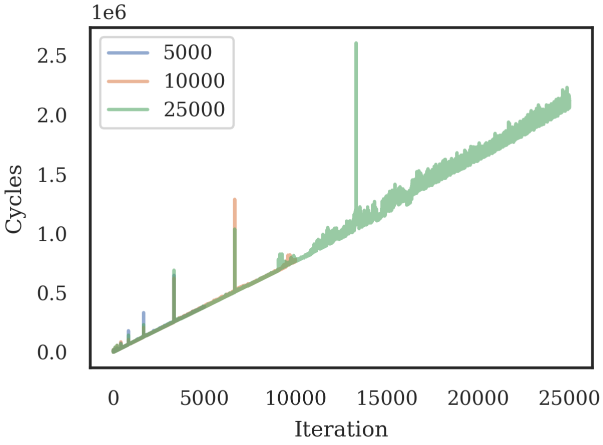

Figure 6: Time consumption of findAvailableLocked in cycles vs. existing number of mappings ahead.

As the functionality of findAvailableLocked is to find available large enough gap within the virtual memory space, such pattern suggests a linear scan algorithm is used there, which can be a practical optimization point. Reading the code of this function shows this is correct. However, the problem is not ended. Careful readers may wonder why this linear pattern only happens on shared/private mmap but not on anonymous mmap? This question actually confuses us for quite a long period and will be explained in Section 5.

4.2 Average Latency vs. Mapping Size

The idea behind Exp2 is to see if the size of mapping has any effect on the average latency. By intuition, larger mappings might cause more trouble. We fix the iteration to 25000 and span the mapping size from 1KB, 2KB to 1MB. The number 25000 is specifically chosen for two reasons: to reach a good balance at a large enough iteration which can still cover large mmap size under the limitation of total memory and to have a reasonable test time, especially because the non-anonymous case in Exp1 does not scale well. We perform the test for 12 trials with 2 trials for warming up and 2 second internal warmup in each trial on the same Linux bare metal used in Exp1.

The test result for anonymous (private) case is shown in Figure 7. In normal Docker (runc) the average latency is constant no matter how large the mapping is. **But in gVisor (runsc-kvm), the average latency of the anonymous case increases in relation to size after 64KB. In contrast, the results for non-anonymous case maintains a constant average latency**, as shown in Figure 8.

We want to understand what's happening in anonymous case. Again, we use perf and generate flame graphs for MMap size at 4KB and 512KB. In 4KB case, we can already see that mapASLocked() which calls the host mmap syscall is taking a dominating (97%) percentage of whole mmap call. In the 512KB case, it reaches 100%. This implies that the latency of mapASLocked is likely increasing with mapping size. But if you look the upper parts of the call stack shown in the flame graph, most time is consumed by KVM operations which are unrelated to gVisor functions.

Similarly, to have a more quantitative analysis, we measure the average time consumption of createVMALocked, getPMAsLocked, and mapASLocked versus different mapping sizes. The same warmup strategy above is still used and we only take the mean of the real benchmark data. The result is shown in Figure 9. Just as hinted by the flame graph, the mapASLocked function, which is calling the host mmap syscall, is the major cause of the increasing latency versus mmap size, especially when larger than 64KB. After discussing with the google developers during the community meeting, **we kind of agree that this is caused by KVM's superpage involvement after a specific value of total size mapped. This is not something relevant to gVisor and can't be optimized.**

Summary Through these two experiments, we figure out a valid optimization point: the virtual memory space free space searching algorithm. More details about how we optimize it is discussed in next section.

5 Optimization

To answer the question why anonymous mmap "does not suffer" from the effects of the linear free space search algorithm and to actually implement optimizations, we need to understand how virtual space is structured within gVisor.

5.1 Virtual Memory Space in gVisor

The virtual memory space inside gVisor, is a set sorted by the starting point of each allocated memory "segment". The set is implemented as a B-tree, rather than the RB-tree used by Linux. Each node of the tree contains from 2 to 5 segments as keys (except the root node). While it's very efficient to add/remove segments into B-tree, the current implementation uses a linear scan with a primitive NextGap() to iterate over and find sufficiently large unallocated gaps for the new allocation. Such search is of time complexity O(n) where n is the total number of segments inside this set.

Insertion and removal of segments decides the total number of segments. When a new mapping is inserted to this set, an optimization is done to see if the previous or next allocated segments have adjacent virtual addresses. If so, the mapping contents are checked and if they are the same thing, these two mapping will be merged. No modification to the tree structure would be done if there's a successful merged insertion.

Now we can answer the mystery question above. Anonymous mappings all have the same file identity "nil" and sequential anonymous in our benchmark would actually always merge successfully. Thus the tree is actually only added one new segment (key) only once throughout the process and the search time is constant. However, for the non-anonymous case, even though all the mappings are mapped to the exact same file, since they are referencing the same region of the file, they don't actually have the same identity because of increased reference count each time and can't be merged. Thus a new node is added to the tree each time there's a new mapping making the searching time increase linearly.

5.2 Solution

Our strategy to optimize the above mentioned search algorithm is to add a new attribute - maxGap within each node to record the longest gap under one node. Actually gaps are defined as non-negative intervals in between segments. So the gaps under one node can viewed as the gaps in between all segments living in the tree rooted but this node, including the heading and trailing one which are shared with this node's parent (if applicable). maxGap stores the maximum one in this closure.

With the help of maxGap attribute, we are able to develop two new member functions of gap: FindNextLargeEnoughGap and FindPrevLargeEnoughGap to replace the NextGap and PrevGap primitives used for linear scan . These functions

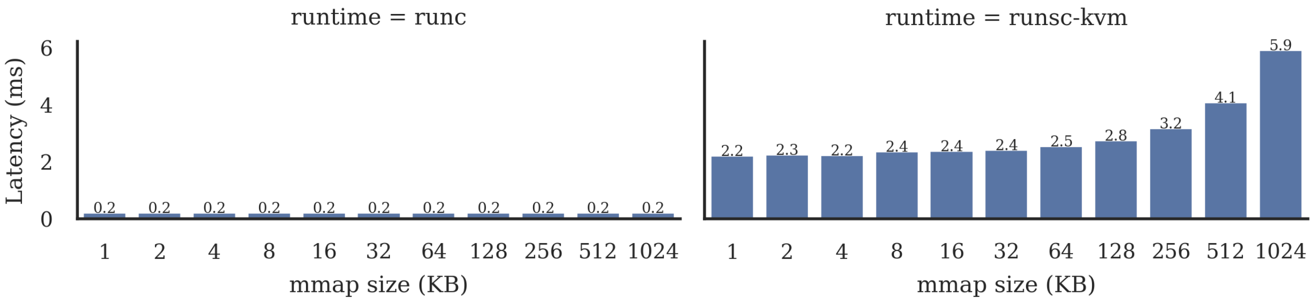

Figure 7: Anonymous mmap latency vs. mmap size.

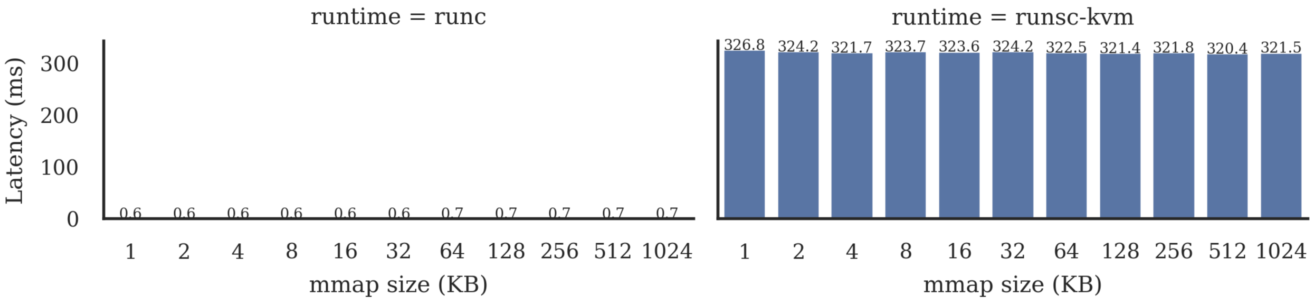

Figure 8: Private mmap latency vs. mmap size.

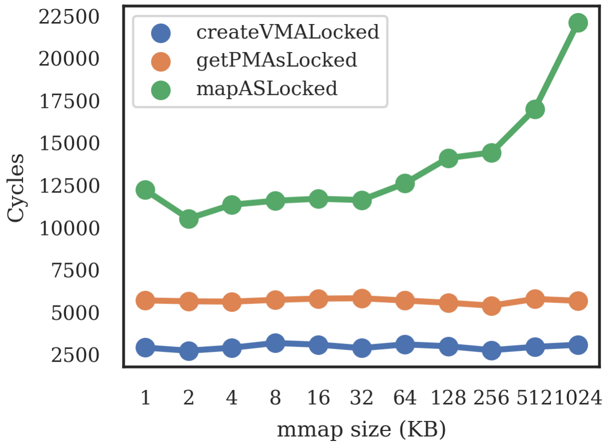

Figure 9: Average cost of subfunctions vs. mmap size.

will return the most adjacent large enough gap immediately if there's one. It only iterates and goes deep into one subtree when the root node has a maxGap attribute larger than the required minSize. The time complexity is reduced from $O(n)$ to $O(\log n)$ as we no longer need to iterate every node.

To avoid violating B-tree invariants (i.e. every leaf node is on the same level and every node must have a certain range of children) the tree structure undergoes rebalancing on insertions and removals. The maintenance of maxGap during insertion, removal and merge is tedious. This is especially true during rebalancing before insertion and rebalancing before removal; there are quite a few various corner cases to consider. A large amount of effort is put there to ensure correctness, but for limitation of pages, no details will be discussed here. If you are interested, you may look into the code base directly. The cost of maintenance of maxGap is still $O(\log n)$, which is identical to the cost of insertion and removal. This means we are only adding a small constant in front of the total time complexity and should be negligible.

To summarize, the cost of this optimization is expected to be very limited while the benefit should be significant. If we reperform Exp1 and Exp2 in Section 4, we may see the latency of anonymous case increased slightly because of the cost of maintenance and we should see the latency of shared/private case dramatically dropped and no longer increasing with iterations in Exp1. Evaluation in practice verifies this expectation and is discussed in Section 6.

6 Evaluation

To evaluate the performance of the optimization, we performed exactly the same suite of benchmarks for mmap with different runtimes, similar to what we did in Section 4 in the notouch and nofree case. Again, **runc** is the normal Docker runtime, **runsc-kvm** is the original gVisor runtime in KVM mode, **runsc-dev** is the gVisor runtime after optimization also in KVM mode. These experiments are performed in Xiaohan's personal computer with bare-metal Linux (the same one as above).

In all cases shared mmap results are very similar to the private mmap results, so we only keep private mmap results in the main text and save shared mmap results in Appendix A.

6.1 MMAP Latency vs. Total Iterations

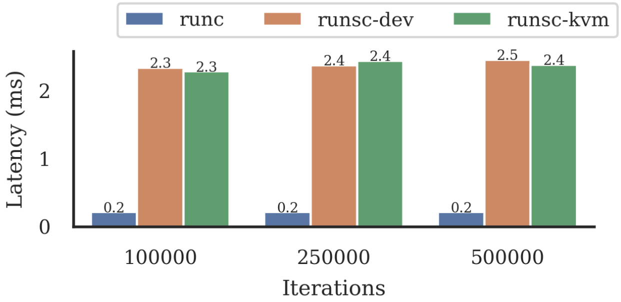

Figure 10: Anonymous mmap latency vs. iterations.

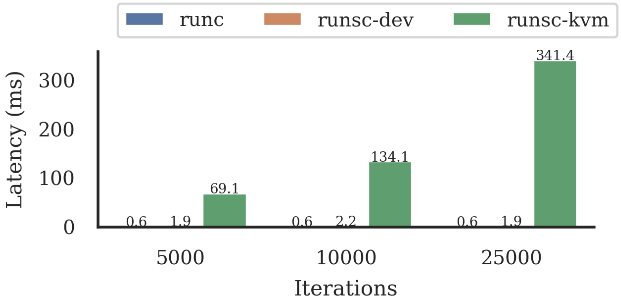

Figure 11: Private mmap latency vs. iterations.

We fix the mmap size to be 4KB, 16KB and 64KB respectively and span the iteration from 100000 to 500000. To save space, we only display the 4KB case here because they all have very close pattern. The results are presented in Figure 10, 11 and 14 for anonymous, private and shared mmap respectively.

In the anonymous case, there's barely difference between the average latency before and after the optimization, which means introduction of maxGap does not add any notable maintenance cost.

Looking at the private or shared case, a significant improvement can be found.You may note that we are now doing consistent iterations as anonymous case because the average latency dramatically dropped. More importantly, the latency is slightly smaller than anonymous case, which makes much more sense compared to the previous case. And now the average latency is close to constant and is no longer increasing by iterations.

6.2 MMAP Latency vs. Mapping Size

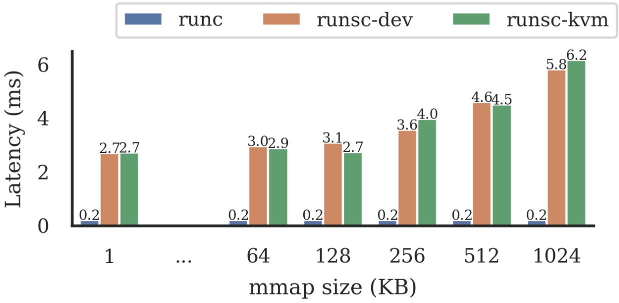

Figure 12: Anonymous mmap latency vs. mmap size.

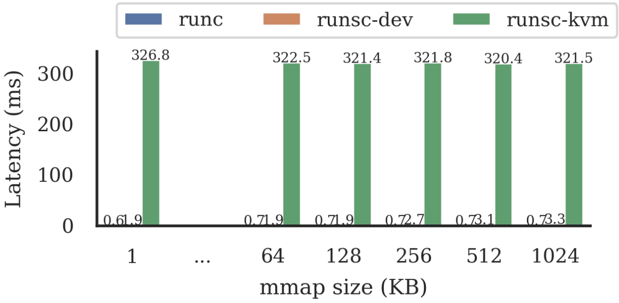

Figure 13: Private mmap latency vs. mmap size.

The second set of experiments is again to see how mmap latency scales with mmap size after the optimization. Iterations are still fixed to be 25000 in these experiments. The results are presented in Figure 12, 13 and 15 for anonymous, private and shared case respectively.

Looking at the anonymous case, you could still see there's very limited difference between the average latency before and after the optimization, which means our introduction of maxGap does not add any significant maintenance cost.

Looking at the private or shared case, along with the obvious improvement on latency, we should note that there's a slight increasing pattern with mapping size. This is reasonable as when the required size gets larger, there would be more recursion and jumps across the tree to find a large enough one. The latency is still smaller than anonymous case.

7 Discussion & Experiences

In this section, we'd like to share the precious experiences we obtained from the challenges we met and mistakes we made.

7.1 Challenges

One of the most challenging task in this project was to get familiar with the toolchains of gVisor. The only related materials or resources available online are the github repository and official website documentation while they are mostly designed for users rather than developers. It took us a long time to figure out how to use the profiling and debugging tools embedded with gVisor. This turned out to not be very helpful and in the end we instead decided to use perf and code injection to perform the benchmark.

It's nevertheless not a easy task to configure perf correctly with gVisor at all. Adin, one of the gVisor developers, said he never succeeded in configuring the perf guest bits correctly. I spent a large amount of time reading the documentation and tutorial of perf and search related materials to finally figure out a way to set it correctly and get things work. To summarize here are three key points.

First, the gVisor container should be directly spawned by runsc instead of forwarded to runsc by dockerd. This means you should first understand OCI and then write a correct configuration file for your container. I tried to write one but it was not able to allow the container to create file descriptors such that shared/private tests can't be monitored. To resolve that, I decided to "steal" the exact configuration file sent by dockerd. However, since late 2019, docker no longer explicitly stores the config file. It's now stored in a mysterious temporal location only when the container is running, which is rarely discussed on the Internet and was hard to find. Second, runsc must be run in KVM mode to produce a useful result as ptrace mode is actually running as a separate process. Third, perf events option must add correct flag to monitor cycles of both host and guest os for KVM hyper-vised platform i.e. -e cycles:HG.

Injecting assembly code into gVisor was also non-trivial. Since the gVisor project is using bazel to manage build and test processes, we had to figure out how to add our new package to the build chain of the bazel project. It took numerous attempts and the help of Ian, another google engineer, to finally able to compile gVisor with the timer.

Another notable obstacle was gVisor being written in Golang, a programming language unfamiliar to us. It took significant practice, reference to tutorials, and looking through the source code before we were able to write and add optimizations.

One another interesting finding during the mmap benchmark is the total number of memory mappings is limited by a system parameter vm.max_map_count, which might be exceeded and cause your mmap to fail silently.

7.2 Mistakes

The most critical mistake we made, is the benchmarking method we initially used. We didn't apply any warmup trials or iterations for the measurement, leading to results that seemed somewhat random and non-deterministic. We wasted a large amount of time understanding those weird patterns and didn't consider the possibility of our benchmark approach itself being incorrect.

However, we later noted the warmup time option inside the benchmark toolset lmbench, and tried running with some warmup trials and iterations which resulted in much more stable and reasonable results. Through tests and trials, we figured out the best warmup scheme to be 2 warmup trials and 2 seconds' warmup time within each trial. This taught us the important of building a stable testing environment and was necessary to remove disturbance from things like caching effects, scaling clock cycles, and so on. Generally, adding warmup, increasing iterations and trials, pinning cpu core and running alone in the system would be essential and helpful.

Another notable mistake we made happened during the optimization. When designing the optimization algorithm, we didn't define the meaning of maxGap clearly enough. The problem lied in whether the two ends of the gaps within a node should be included. Despite this, I started coding which resulted in a lot of bugs in the first version of code. After going back and designing carefully, the code ran much better. This teaches me again, that a thorough high level design of the algorithm, like the choice of invariant, is more important than the implementation itself.

8 Conclusion

The purpose of this project was to identify causes of performance degradation in gVisor's memory management as well as propose approaches to mitigate their effects. We target the mmap command specifically due to the significant slowdown and poor scaling exhibited. After exploring the source code of gVisor and running tons of testbenches, we dissect the implementation of mmap and identify edge cases in its behavior which we walk through in this paper. We find and attempt to address scaling issues in certain edge cases of mmap.

We end by proposing and implementing an optimization to improve mmap performance and scaling by avoiding linear scanning in finding gaps to allocate memory by introducing a new attribute, maxGap, to track gaps. This turns out to be a valuable pull request which is finally merged to the master branch.

Acknowledgments

We are particularly grateful for Yiying Zhang, the course instructor, for giving us feedback and directions as the project progresses. We thank Ethan from University of Wisconsin, Madison for answering questions on paper [5] he authored. We also thank the gVisor development team, especially Adin, Jamie and Ian for their helpful suggestions and comments along the way. Also special thank given to Yihao Liu for his help on perf and bazel configuration.

Availability

The benchmark and evaluation code and data are publicly available at https://github.com/291j-gvisor/gvisor-mem-perf. The commit to the official gVisor repository can be viewed at https://github.com/google/gvisor/commit/059879e.

References

[1] Linux Programmer's Manual - MMAP(2). http://man7.org/linux/man-pages/man2/mmap.2.html, 2019-10-10.

[2] Aaron Grattafiori. Understanding and hardening linux containers. Technical report, NCC Group, 2016. https://www.nccgroup.trust/us/our-research/understanding-and-hardening-linux-containers/.

[3] Google gVisor Team. gVisor Documentation - Performance Guide. https://gvisor.dev/docs/architecture_guide/performance/, accessed January 2020.

[4] Nicolas Lacasse. Open-sourcing gVisor, a sandboxed container runtime. https://cloud.google.com/blog/products/gcp/open-sourcing-gvisor-a-sandboxed-container-runtime, May 2018.

[5] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The true cost of containing: A gVisor case study. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'19, page 16, USA, 2019. USENIX Association. https://dl.acm.org/doi/10.5555/3357034.3357054.

A Shared MMAP Evaluation Results

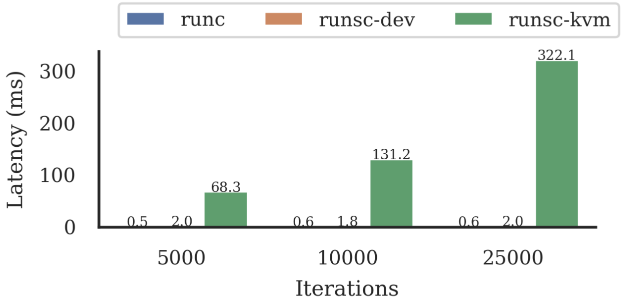

Figure 14: Shared mmap latency vs. iterations.

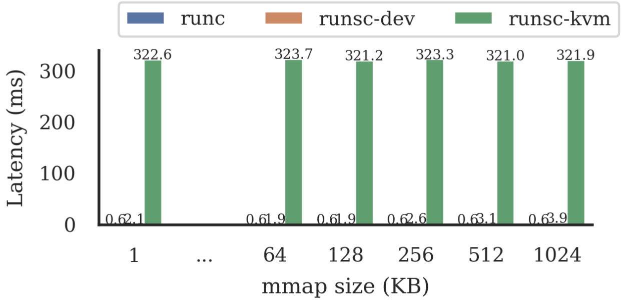

Figure 15: Shared mmap latency vs. mmap size.