Virtualization Project Final Report

Enhancing KVM Page Eviction: From FIFO to Approximate LRU

Aaron Ang, Alex Asch, Eric Huang, Justin Kaufman

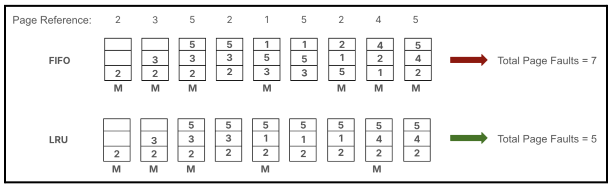
I. INTRODUCTION

Context and Motivation

Virtualization has proven to be an essential technology in modern computing, allowing multiple operating systems to run on a single physical machine. One of the most widely adopted virtualization technologies in the Linux ecosystem is Kernel-based Virtual Machine (KVM), which leverages a kernel module to turn Linux into a hypervisor. A key component of KVM is its Memory Management Unit (MMU), which is responsible for memory isolation and translating guest virtual addresses to host physical addresses.

Currently, the KVM MMU employs a First-In-First-Out (FIFO) page eviction mechanism, which removes the oldest page without considering its recency of use. This approach can result in suboptimal performance, as frequently accessed pages may be prematurely evicted, leading to increased page faults and performance overhead. Our project proposes an Approximate Least Recently Used (LRU) eviction policy to mitigate these inefficiencies, aiming to enhance memory reuse while minimizing unnecessary evictions. Through comparative analysis with FIFO within KVM, we aim to demonstrate the benefits of an LRU-based approach across various workloads.

Comparison FIFO vs. LRU



(Figure 1)

Figure 1 illustrates the fundamental difference between FIFO and LRU eviction strategies in the KVM MMU. The FIFO approach removes the oldest page in memory without considering its usage frequency, leading to unnecessary evictions and increased page faults. In contrast, LRU keeps track of recently accessed pages, ensuring that frequently used pages, such as page 2 and page 5, remain in memory longer. As a result, LRU reduces the total number of page faults (5) compared to FIFO (7), demonstrating its advantage in preserving memory locality and minimizing costly page replacements. This aligns with our project's goal of improving KVM's memory management efficiency by replacing FIFO with an Approximate LRU mechanism.

Scope and Objectives

Given the importance of memory management in virtualization, our project focuses on integrating an Approximate LRU eviction mechanism into KVM's MMU. Specifically, our objectives were:

1. Design and Implement an LRU Eviction Policy:

- Modify KVM's *mmu.c* file to track page access recency and replace

FIFO with an LRU-based eviction logic to incorporate LRU-based eviction logic.

- Adapt existing Linux kernel data structures to support an efficient Approximate LRU approach with minimal overhead

2. Validate Correctness Through Testing:

- Use KVM unit tests and custom workloads to verify correctness and stability of the new eviction policy.
- Monitor page eviction patterns using kernel tracing tools such as tracepoints and Kprobes.

3. Evaluate Performance Gains:

- Measure page fault frequency, CPU utilization, and memory overhead using benchmarking tools.
- Compare the Approximate LRU implementation against FIFO to determine efficiency gains.

4. Address Concurrency and Scalability:

- Ensure the LRU eviction mechanism remains thread-safe in multi-CPU environments.
- Validate that the solution scales well under high-memory workloads.

II. RELATED WORKS

Before diving into our implementation, it is important to address related works. Much of the prior research contrasting FIFO and LRU replacement policies is rooted in both theoretical and practical investigations. Notably, "It's Time to Revisit LRU vs. FIFO" highlights how large-scale caches, such as those in cloud object storage, experience significant metadata overhead when implementing LRU. Because storing and updating recency metadata can be expensive for massive caches, FIFO's simpler management can sometimes produce less overhead [2]. This is an important consideration for our work in cases where a large memory pool may affect CPU overhead and latency.

On the other hand, a formal analysis conducted in "LRU is Better Than FIFO" underscores the fact that LRU outperforms FIFO in paging systems, especially when workloads exhibit strong locality of reference. Their analysis results emphasize the idea that retaining least recently used pages is beneficial for maintaining "hot" data in cache [1]. This aligns with our project's objective to integrate a more locality-aware eviction policy (LRU) into KVM's MMU. Although, we must remain mindful of the potential overhead emphasized by "It's Time to Revisit LRU vs. FIFO".

Additionally, it is important to note development work within the context of KVM itself. The FIFO implementation of shadow page tables is a feature only used when ept hardware virtualization resources are not available within the environment. Due to this, within most use cases, KVM uses a TDP MMU implementation that leverages the given hardware, and is much more performant that shadow paging. Thus, while our work may show performance benefits over the old implementation, it is not a general improvement to the KVM project.

III. DESIGN OVERVIEW

System Architecture

At a high level, our modified virtualization environment retains the existing KVM framework, which provides the interface between guest virtual machines and host hardware. KVM operates in kernel space, exposing virtualization extensions to a user-space manager (e.g. QEMU) responsible for configuring virtual CPUs, devices, and memory. Under the hood, KVM implements its own Memory Management Unit (MMU) subsystem, managing page tables and address translations for each guest VM.

Our project inserts a custom eviction mechanism into this KVM MMU layer, specifically within the page table code that decides when and how to replace memory pages. The flow below outlines a simplified conceptual model of how KVM handles virtualization and memory:

1. Guest VMs issue memory operations (loads, stores), thinking they are running on real hardware

2. KVM traps these operations as needed, using the host CPU's virtualization capabilities

3. Page Table Structures in KVM's MMU track which guest pages are resident, which must be evicted, and handle faults

4. Host Kernel eventually services the page faults, interacting with actual RAM or swap

This flow is largely unchanged by our intervention except for the eviction logic, which now uses an approximate LRU-based approach. By preserving the rest of the hypervisor's design, we minimize disruption to the broader virtualization stack while still providing a novel improvement in page eviction decisions.

IV. IMPLEMENTATION

Developer Environment Set up

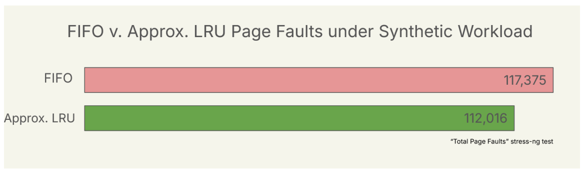
Our initial plan was to configure local development and run nested virtualization for testing KVM. However, hardware limitations proved challenging: only one team member had an x86 architecture laptop, while the others used Apple Silicon (ARM-based) MacBooks. Since KVM is natively supported on x86 but not on ARM, we had difficulties building and running custom kernels with KVM-enabled virtualization locally. We also encountered setbacks with VirtualBox's nested virtualization. Although theoretically supported, its performance and stability were inadequate for continuous kernel development. To address these limitations, we first experimented with UTM (Universal Turing Machine) on macOS, but it did not support nested virtualization on Apple Silicon hardware. Next, we attempted to launch a bare-metal AWS instance, discovering that the default plans available to us had insufficient vCPU allocations. Eventually, after discussions with our course instructor, we obtained access to a c5n.metal instance that met our requirements. There, we set up a full Linux environment with KVM enabled, allowing us to compile and debug custom kernel builds. On our one x86 laptop, we further performed local tests and kernel compilations, though we had to re-partition the Windows EFI space to accommodate larger kernels. All of this helped us establish a consistent development workflow, merging and testing changes in an environment representative of production KVM scenarios.

The primary modification to KVM's MMU involved adjusting the eviction logic to incorporate Approximate LRU. We introduced data structures to estimate page recency and modified the mmu_zap_oldest_pages() function to prioritize eviction based on access history rather than simple FIFO ordering. This required adding lightweight tracking metadata to each page table entry while ensuring that the eviction process remained efficient and scalable. Furthermore, we disabled Two-Dimensional Paging (TDP) MMU in our testing environment to directly observe and evaluate the impact of our changes.

Ensuring thread safety was a crucial aspect of our implementation. KVM operates in a multi-core environment where multiple virtual CPUs (vCPUs) can simultaneously access memory. To address potential concurrency issues, we incorporated synchronization mechanisms to prevent race conditions while maintaining performance efficiency. This involved carefully modifying memory structures to ensure safe updates without introducing unnecessary contention.

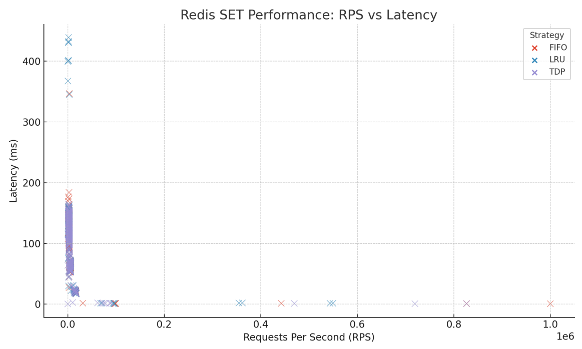
V. EVALUATION METHODS/RESULTS

Our evaluation methodology focused on assessing the effectiveness of the Approximate LRU implementation in improving memory efficiency, reducing page faults, and maintaining low latency under load, all while minimizing performance overhead. To achieve a comprehensive analysis, we used a combination of unit testing with KVM's built-in test suite, synthetic memory stress testing with stress-ng, performance profiling using flamegraphs, and application-level benchmarking with redis-benchmark. Key performance metrics included page fault frequency, latency, CPU overhead, and request throughput (RPS).



(Figure 2)

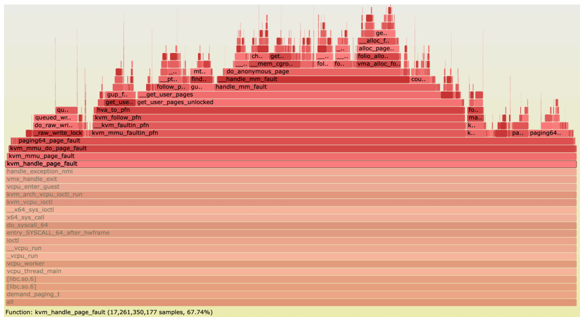
One of the most direct indicators of improved memory management is page fault frequency. Under stress testing with stress-ng, the traditional FIFO eviction policy resulted in 117,375 page faults. In contrast, our Approximate LRU implementation reduced this number to 112,016. This reduction highlights more efficient memory reuse and fewer unnecessary page replacements. Furthermore, performance profiling using flamegraphs revealed that Approximate LRU led to fewer costly MMU interventions, supporting the claim of reduced overhead and better memory locality.



(Figure 3)

Additionally, to evaluate real-world performance impact, we used the *redis-benchmark* tool, which simulates high-throughput client workloads. The resulting graph shows the relationship between RPS and latency for three page replacement strategies: FIFO, LRU, and TDP. LRU consistently delivers lower latency across a wide range of RPS values, highlighting its efficiency in handling frequent memory accesses. In contrast, FIFO exhibits increasing latency under higher loads, indicating poor adaptability under stress. While TDP shows some performance stability, it is more variable compared to LRU. Overall, this graph demonstrates that our Approximate LRU implementation offers clear advantages over FIFO by sustaining high throughput with low latency, validating its practical effectiveness in the KVM environment.

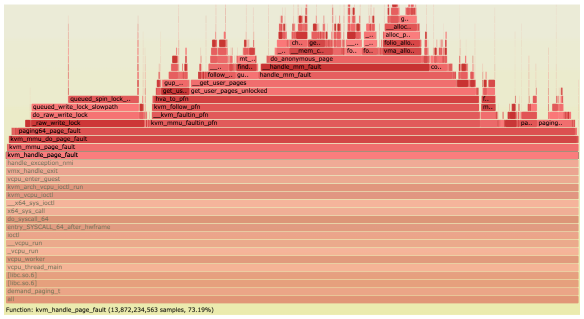
FIFO Flamegraph



Function: kvm_handle_page_fault (17,261,350,177 samples, 67.74%)

(Figure 4)

Approx. LRU Flamegraph



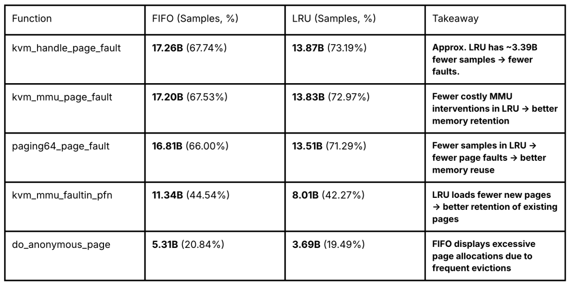
Function: kvm_handle_page_fault (13,872,234,563 samples, 73.19%)

(Figure 5)

The evaluation of function-level performance highlights the impact of Approximate LRU on page fault handling efficiency. As shown in Figure 4, key MMU functions, such as kvm_handle_page_fault and kvm_mmu_page_fault, executed fewer times under Approximate LRU compared to FIFO, indicating a reduction in overall page faults. Specifically, Approximate LRU resulted in 3.39 billion fewer samples in kvm_handle_page_fault, demonstrating its ability to retain frequently accessed pages and minimize unnecessary evictions. Similarly, the reduced execution of paging64_page_fault and kvm_mmu_faultin_pfn suggests improved memory reuse, as fewer new pages needed to be loaded.

However, while Approximate LRU reduces costly MMU interventions and improves memory retention, CPU profiling revealed a slight increase in execution time for MMU-related functions. This increase is attributed to the additional overhead introduced by tracking page access recency. Consequently, although Approximate LRU optimizes eviction decisions, it imposes a tradeoff between efficiency and computational complexity. This tradeoff must be carefully considered in scenarios where CPU overhead is a critical constraint. Nonetheless, the overall reduction in page faults confirms that Approximate LRU enhances memory management effectiveness compared to FIFO.

Performance Comparison



Function	FIFO (Samples, %)	LRU (Samples, %)	Takeaway
kvm_handle_page_fault	**17.26B** (67.74%)	**13.87B** (73.19%)	**Approx. LRU has ~3.39B fewer samples → fewer faults.**
kvm_mmu_page_fault	**17.20B** (67.53%)	**13.83B** (72.97%)	**Fewer costly MMU interventions in LRU → better memory retention**
paging64_page_fault	**16.81B** (66.00%)	**13.51B** (71.29%)	**Fewer samples in LRU → fewer page faults → better memory reuse**
kvm_mmu_faultin_pfn	**11.34B** (44.54%)	**8.01B** (42.27%)	**LRU loads fewer new pages → better retention of existing pages**
do_anonymous_page	**5.31B** (20.84%)	**3.69B** (19.49%)	**FIFO displays excessive page allocations due to frequent evictions**

(Figure 6)

Overall, our evaluation confirmed that Approximate LRU successfully mitigates the primary drawbacks of FIFO, reducing page faults and improving memory locality at the cost of a slight increase in CPU processing. These findings suggest that Approximate LRU offers a viable alternative to FIFO, particularly for workloads that benefit from improved memory retention and reduced eviction churn.

VI. CHALLENGES

Difficulties Encountered

Our initial difficulties were in relation to environment setup and understanding of the codebase. To begin, we had many challenges in getting a development environment in which we could test KVM and build the code within, since having much of our group on M1 mac, we needed an AWS bare metal environment to be able to develop. This added significantly to our time to start development. Additionally, understanding the large codebase of KVM, and the relatively large codebase of the MMU proved to be difficult. Between the two above steps, we encountered significant difficulties before beginning to implement our changes. Additionally, in benchmarking and testing, even after implementation and debugging, we had some stability issues, especially on larger tests. Due to this, our ability to gather additional information outside of our initial testing workloads was not possible.

What Didn't Work

Within our implementation, our first finding was that the TDP mmu was not able to be optimized in this way, and thus found that the analysis and optimization was to be done on the older and disabled by default shadow mmu. From this, our initial implementation was rather quick considering we implemented on the existing mmu, but debugging and ensuring stability were both notable challenges, some still pending.

What Did Work

Using a second change clock LRU algorithm, with new functions for tagging pages and a global clock hand proved to be our final implementation. While there are still stability concerns with this implementation in regards to deadlocking and faults, we generally find it performant and from the above results better than the old FIFO mmu.

VII. CONCLUSION

Summary of Contributions

In this project, we successfully implemented an Approximate LRU eviction policy to replace the existing FIFO policy within the KVM Memory Management Unit (MMU). Our modifications aimed to improve memory reuse, reducing page faults and enhancing overall system efficiency. To validate our changes, we conducted extensive testing in a controlled environment, using tracing, debugging, and benchmarking to analyze system behavior. Our results confirmed eviction method tradeoffs. Notably, approximate LRU reduces page faults and enhances memory efficiency, however FIFO incurs a lower CPU overhead. The code can be found at: https://github.com/aaron-ang/kvm.

Future Extensions

While our Approximate LRU implementation yielded positive results, there are several directions for future improvement:

1. Optimization: Fine-tune Approximate LRU parameters to achieve a better balance between eviction efficiency and system overhead, as well as further debug the aforementioned stability issues.

2. Additional Benchmarking: Evaluate the policy under a broader set of workloads and system configurations to ensure robustness across different environments.

3. Exploring Alternative Eviction Policies: Compare the Approximate LRU with other eviction policies, such as hybrid FIFO-LRU

or machine-learning-driven eviction mechanisms.

Reflection

This project demonstrated the feasibility of improving KVM MMU memory management by replacing FIFO with Approximate LRU. The process involved implementing the new policy, validating it in controlled virtualized and cloud-based environments, and analyzing its real-world performance. One of the key takeaways from this work is the importance of balancing efficiency and overhead, while more sophisticated eviction policies may further reduce page faults, they must remain lightweight to avoid introducing performance bottlenecks. Additionally, the adaptability of eviction strategies across different workloads remains an open challenge, making ML-driven approaches an exciting avenue for future research. Overall, this work contributes to the ongoing enhancement of KVM memory management, providing a foundation for future research in eviction policies and performance optimization in virtualization environments.

REFERENCES

[1] Chrobak, Marek & Noga, John. (1999). LRU is better than FIFO. Algorithmica. 23. 180-185. 10.1007/PL00009255.

[2] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. 2020. It's time to revisit LRU vs. FIFO. In Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '20). USENIX Association, USA, Article 12, 12.