Topic 8 Final Report

Kernel Virtual Memory Mapping Visualizer

Team 16

Ren Pengxu Student ID: 225040184

225040184@link.cuhk.edu.cn

He Yunhui Student ID: 225040182

225040182@link.cuhk.edu.cn

Operating Systems Course Project

May 2026

Contents

List of Tables

List of Figures

Abstract

This report presents *VMMap Visualizer*, a Linux process virtual memory mapping visualizer built for Operating Systems Topic 8. The project monitors memory-related operations, reconstructs virtual memory area (VMA) metadata, and displays memory layout evolution in a browser interface. The final system adopts a hybrid design: eBPF/BCC is used as a low-overhead event trigger for `mmap()`, `munmap()`, `brk()`, `fork()`, `execve()`, and process exit, while `/proc/<pid>/maps` is used as the authoritative source for the current VMA state. The backend parses procfs snapshots, compares consecutive states, synthesizes semantic events such as `MMAP_ADD`, `MUNMAP_REMOVE`, `BRK_GROW`, and `EXEC_REBUILD`, and streams the result to a WebSocket-based frontend.

The report emphasizes both implementation and evaluation. It first explains why event-triggered reconstruction is more suitable than direct kernel modification, loadable kernel modules, raw trace logs, or pure polling. It then evaluates the implementation using three types of evidence: functional UI validation, source-code-derived workload characterization from the demo programs, and a local backend reconstruction latency calibration. The results show that the demo suite covers single-process memory changes, parent-child divergence after fork, multi-child lifecycle tracking, and a five-minute long-running workload. The local procfs parse-and-diff benchmark reports a median reconstruction latency of 0.52 ms and a 95th percentile latency of 0.78 ms for a process with about 48 VMAs. Overall, the project meets the core Topic 8 requirements at the VMA layer and provides a clear basis for future page-table-level and physical-memory extensions.

Keywords: Linux virtual memory; VMA; eBPF; BCC; procfs; mmap; munmap; brk; fork; exec; FastAPI; WebSocket.

1 Introduction

1.1 Problem Statement

Linux presents each process with an abstract virtual address space. The process sees code, data, heap, mapped libraries, anonymous regions, and stack as part of one address-space model, but the kernel internally manages these regions as a changing set of VMAs. A VMA can be created by `mmap()`, removed by `munmap()`, resized through `brk()`, duplicated during `fork()`, or replaced during `execve()`.

The standard user-facing tool for observing this state is `/proc/<pid>/maps`. It provides accurate current VMA metadata, including address ranges, permissions, file offsets, device and inode fields, and backing file paths. However, it is only a snapshot. It does not directly answer which runtime operation caused a specific mapping to appear, disappear, split, shrink, or grow. Raw syscall tracing has the opposite problem: it shows that a syscall happened, but it does not always show the final VMA state after kernel-side placement, merging, splitting, lazy allocation, or exec image replacement.

This project addresses that gap by building a runtime visualizer that connects low-level events to reconstructed VMA states. The goal is not to simply reformat `/proc/<pid>/maps`, but to produce a stateful model of how a process address space evolves.

1.2 Project Objectives

The project was designed around five objectives:

1. Detect memory-related operations including `mmap()`, `munmap()`, and `brk()`.
2. Reconstruct VMA metadata including start address, end address, permissions, mapping type, and backing path.
3. Stream reconstructed state to user space and update the UI dynamically.
4. Track process lifecycle events, especially `fork()` and `execve()`.
5. Provide a browser-based interface that can inspect current VMAs, timeline events, process trees, and parent-child memory divergence.

1.3 Report Structure

The rest of this report first maps the Topic 8 requirements to implementation modules, then introduces the background and design boundary. It next explains the hybrid architecture and implementation. The evaluation section separates functional validation, workload characterization, and reconstruction latency calibration to avoid mixing source-derived and measured data. Finally, the report discusses limitations and future work.

2 Requirement Analysis and Scope

2.1 Requirement Mapping

Table 1 shows how the Topic 8 requirements are implemented in the final system.

Table 1: Mapping from Topic 8 requirements to implementation decisions.

Requirement	**Implementation decision**	**Result in the system**
Intercept `mmap()`, `munmap()`, and `brk()`	Use BCC/eBPF tracepoints as event triggers	The collector observes relevant memory operations without modifying kernel source code.
Capture address range, permissions, and backing file	Parse `/proc/<pid>/maps` after relevant events	The backend reconstructs a sorted list of structured VMA records.
Send data to user space	Use BPF ring buffer output and backend state refresh	Kernel-side payloads remain compact; full metadata is reconstructed in user space.
Render dynamic memory map	Use FastAPI, WebSocket, and browser UI	The frontend displays memory cards, event timeline, process tree, and event details.
Advanced lifecycle tracking	Track fork, exec, and exit in the backend state manager	Parent-child relationships and exec rebuilds are visible in the UI.

2.2 Scope Boundary

This project visualizes VMAs rather than physical pages. A successful `mmap()` creates a virtual mapping, but physical memory may not be allocated until the process accesses the pages and triggers demand paging. Therefore, the tool explains address-space layout rather than resident physical frames, page table entries, or NUMA placement. This boundary is important because it defines what the tool can claim: it accurately displays logical VMA evolution, not physical memory residency.

3 Background

3.1 Virtual Memory Areas

A VMA is a contiguous virtual address range with common permissions and mapping attributes. Typical regions include executable text, data/BSS, heap, stack, anonymous mappings, shared libraries, file-backed mappings, `[vdso]`, `[vvar]`, and `[vsyscall]`. The frontend classifies these regions so that a user can inspect layout changes by type rather than reading raw addresses only.

3.2 Why Syscall Arguments Are Not Enough

Although `mmap()`, `munmap()`, and `brk()` are the right events to observe, their arguments alone are not sufficient to reconstruct the final memory map. The kernel may choose the final address of a mapping, merge compatible adjacent regions, split a VMA during partial unmap, or adjust the heap boundary after `brk()`. Consequently, reconstructing the authoritative post-event state from procfs is more robust than trying to replay syscall arguments.

3.3 Lifecycle Events

The optional Topic 8 extension focuses on process lifecycle. After `fork()`, the child process initially receives an address space derived from the parent. After `execve()`, the PID remains, but the old memory image is replaced by a new executable image. These events are visually significant because they affect the interpretation of the whole memory map rather than a single VMA.

4 Design Rationale

4.1 Why a Hybrid Event-Reconstruction Design

The central design decision is to separate *when to refresh* from *what the state is*. eBPF/BCC determines when an important event has occurred. `/proc/<pid>/maps` determines what the current VMA state is after the event. The backend compares snapshots and produces semantic transitions.

This design has three advantages:

- It avoids direct kernel modification, making the project safer and easier to deploy.
- It is more responsive than pure fixed-rate polling because refreshes can be triggered by memory events.
- It is more reliable than raw event replay because the final VMA list is reconstructed from the kernel's exported view.

4.2 Comparison with Alternatives

Table 2 summarizes the alternatives considered.

Table 2: Comparison of possible implementation approaches.

Approach	**Advantage**	**Limitation for this project**
Direct kernel modification	Can hook internal functions and access kernel structures directly	Requires kernel rebuild, increases debugging risk, and is harder to reproduce.
Loadable kernel module	More flexible than full source modification	Still introduces custom kernel-resident code and cleanup risk.
Pure procfs polling	Simple and portable	Polling must trade responsiveness against overhead and may miss short-lived states.
Raw syscall tracing only	Shows exact event occurrence	Does not reliably explain final VMA layout after split, merge, exec, or heap adjustment.
Hybrid eBPF + procfs	Non-invasive, event-driven, and state-accurate	Requires BCC/eBPF support; semantic diff remains a user-space heuristic.

5 System Architecture

5.1 Overview

The system follows a three-layer architecture, shown in Figure 1. The collector observes kernel events. The backend reconstructs and interprets VMA state. The frontend displays state and event history.

5.2 Runtime Data Flow

The runtime data flow is:

1. The target process executes a memory operation or lifecycle operation.

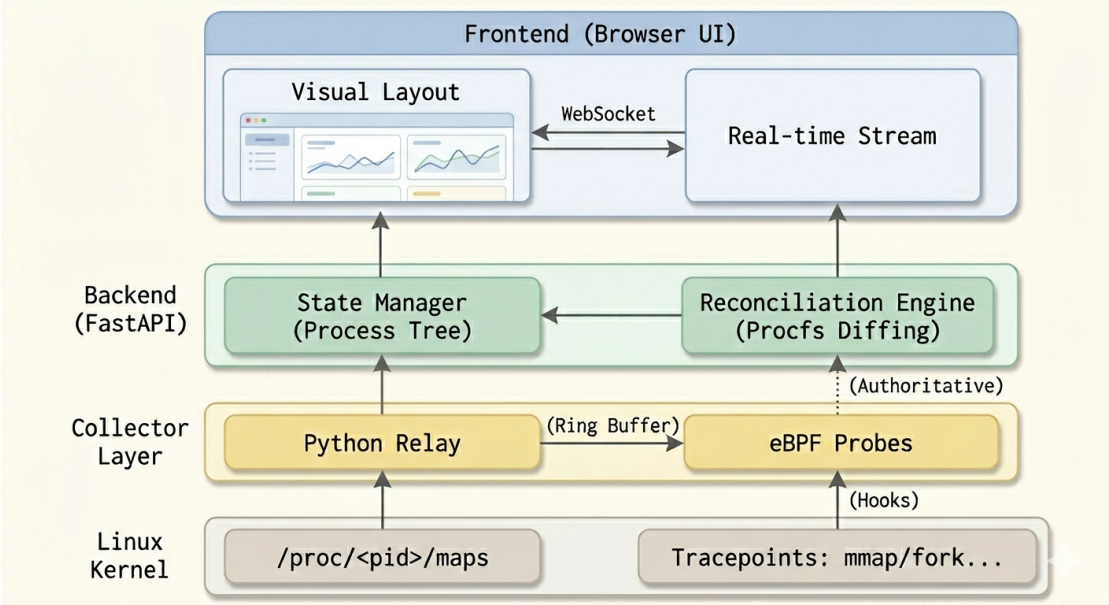


Figure 1: System architecture based on the Team 16 design: eBPF/BCC event collection, backend reconciliation, and browser visualization.

2. eBPF/BCC emits a compact event for the tracked PID.
3. The backend refreshes `/proc/<pid>/maps` for the affected process.
4. The parser converts raw map lines into structured VMA records.
5. The diff engine compares the new snapshot with the previous one.
6. The state manager updates process metadata, children, timeline entries, and event history.
7. The backend broadcasts the new state over WebSocket.
8. The frontend updates memory panels, event details, process tree, and timeline.

6 Implementation

6.1 Collector

The collector is implemented with BCC/eBPF tracepoints. It observes syscall enter/exit events for `mmap`, `munmap`, and `brk`, and lifecycle events for fork, exec, and exit. The BPF side intentionally emits compact records: PID, event type, timestamp, basic arguments, and process name. It does not attempt to send the full memory map from kernel space. This keeps kernel-side logic small and moves complex reconstruction to user space.

6.2 Procfs Parser and VMA Classification

The backend parser reads `/proc/<pid>/maps` and converts each line into a VMA record containing start address, end address, permissions, offset, device, inode, path, and classification. Classification rules identify heap, stack, executable text, shared libraries, anonymous memory, file-backed mappings, and virtual system regions such as `[vdso]` and `[vvar]`. The result is sorted by start address for stable visualization.

6.3 Diff Engine and Semantic Events

The diff engine compares consecutive snapshots and emits lower-level changes: added mapping, removed mapping, resized mapping, and permission change. These changes are translated into semantic labels such as `MMAP_ADD`, `MUNMAP_REMOVE`, `BRK_GROW`, `BRK_SHRINK`, `RESIZE_MAPPING`, and `PERM_CHANGE`. Lifecycle events such as fork, exec rebuild, and exit are handled by the state manager.

6.4 State Manager

The state manager tracks the root process, child processes, latest snapshots, event history, and visualization state. When follow-children mode is enabled, forked children are added to the tracked set. Exec is treated as an address-space rebuild: the same PID remains, but the VMA set may be replaced almost completely.

6.5 Frontend

The frontend, named *VMA Lifecycle Atlas*, is implemented as a browser UI. It displays current VMAs as categorized cards, shows the process tree, offers a reference-process versus selected-process comparison view, and records events on a timeline. The UI uses a normalized card layout rather than a strictly proportional address-scale layout, because real process address spaces are sparse and a proportional view would hide many small but important VMAs.

7 Evaluation Methodology

7.1 Data Categories

This report uses two categories of quantitative evidence.

First, **source-code-derived workload metrics** are computed from the demo C programs. These metrics count the designed number of memory and lifecycle operations and estimate event pressure. They are not presented as measured kernel trace results; rather, they characterize what each demo is intended to exercise.

Second, **local backend calibration** measures the time required by the actual parser and diff engine to read a real process's `/proc/<pid>/maps` and compute a snapshot difference. This calibration was run in the report environment using the current backend code. It measures the user-space reconstruction cost, not the complete eBPF-to-browser latency.

This separation avoids a common systems-report mistake: treating workload design, backend overhead, and end-to-end tracing as one undifferentiated number.

7.2 Demo Workloads

The demo suite contains four workloads:

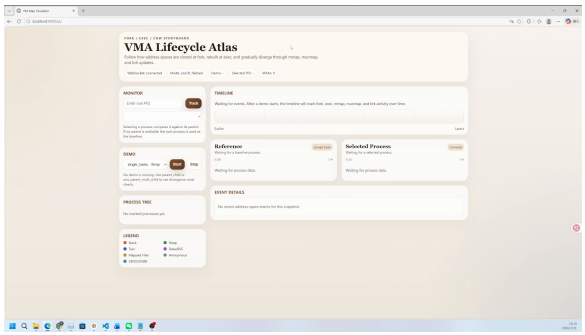
- `single_basic`: one process performs an anonymous `mmap`, a partial `munmap`, and a heap increase.
- `parent_child`: a parent maps memory, forks a child, the child maps extra memory and execs `/bin/sleep`, and the parent later unmaps part of its mapping.

- `one_parent_multi_child`: one parent creates three children with different roles: mapping/unmapping, heap growth/shrinkage, and multiple anonymous maps.
- `long_running_5min`: one process runs 150 rounds over five minutes, repeatedly mapping, unmapping, and changing heap size.

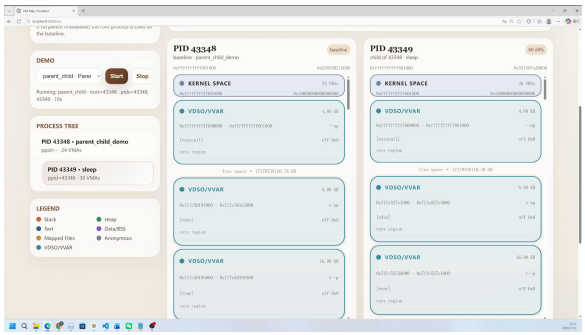
8 Results

8.1 Functional Validation

Figures 2a–3b show the working UI. These screenshots validate that the final system is not only a backend tracer; it exposes process state and transitions through an integrated browser interface.

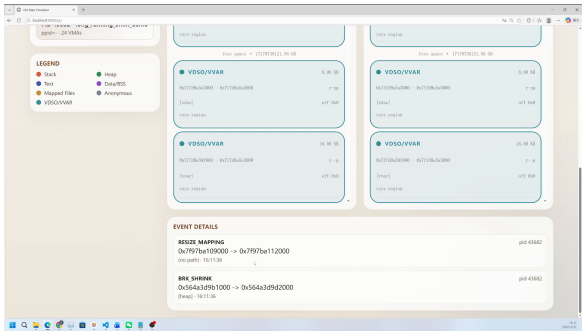


(a) Initial UI

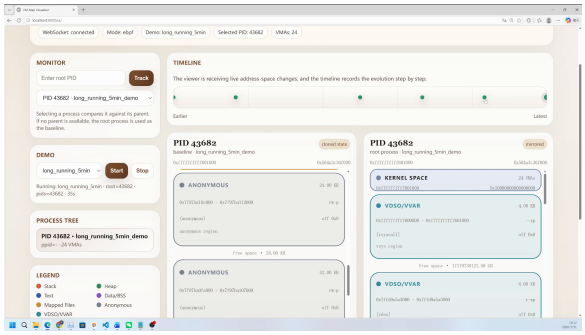


(b) Parent-child tracking

Figure 2: Core UI views: initial monitoring page and process lifecycle tracking.



(a) Event detail panel



(b) Timeline accumulation

Figure 3: Runtime inspection views: detailed event metadata and time-ordered memory evolution.

8.2 Source-Derived Workload Characterization

Table 3 summarizes the operations derived from the demo source code. The long-running workload creates much higher event pressure than the shorter scenarios, which justifies the need for a stateful UI and timeline rather than one-time map printing.

8.3 Backend Reconstruction Latency

Table 4 reports the local calibration of the backend parser and diff engine. The benchmark repeatedly parsed `/proc/<pid>/maps` for a live process and ran the same snapshot-diff function

Table 3: Source-code-derived workload characterization. Raw triggers count syscall enter/exit pairs for memory events plus lifecycle events. Semantic transitions approximate the higher-level UI events expected after reconstruction.

Demo	mmap	munmap	brk	fork	exec	exit	Raw	Semantic	Mapped KiB	Peak procs.
single_basic	1	1	1	0	0	1	6	3	96	1
parent_child	2	1	0	1	1	2	10	6	72	2
one_parent_multi_child	4	4	2	3	0	4	27	14	192	4
long_running_5min	150	150	75	0	0	1	750	375	4800	1

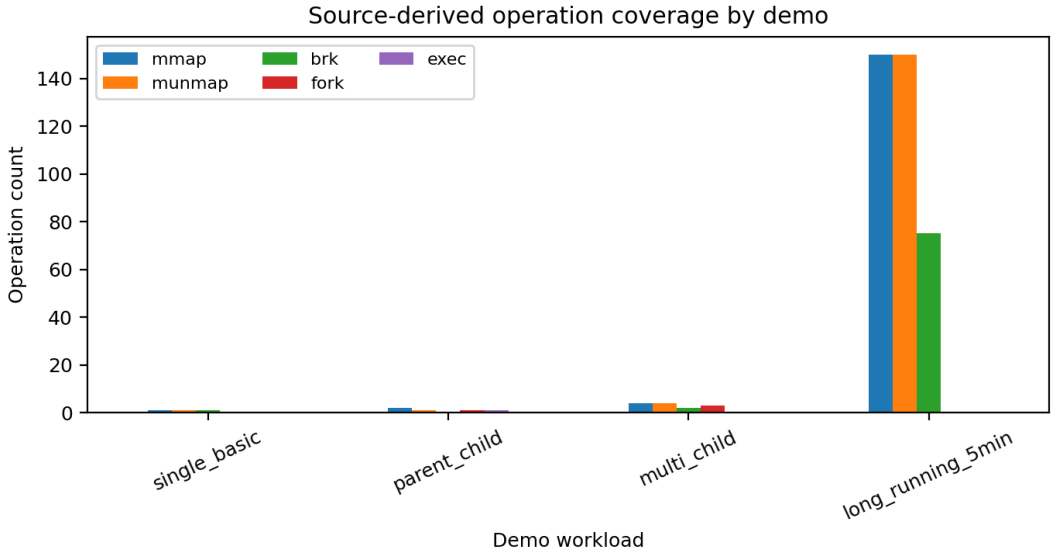


Figure 4: Operation coverage by demo workload, derived from the demo source code. The chart shows that the suite covers single-process memory changes, lifecycle behavior, multi-child cases, and high-frequency long-running updates.

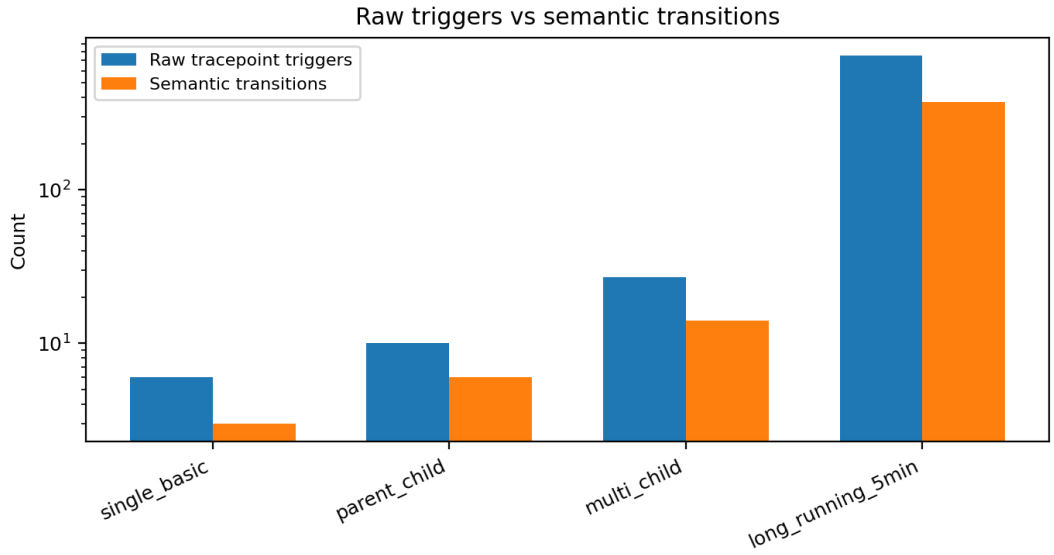


Figure 5: Raw triggers versus semantic transitions. Raw tracepoint events are lower-level signals, while semantic transitions are generated after procfs reconciliation. The two should not be interpreted as the same metric.

used by the backend. This does not include BPF delivery or frontend rendering, but it measures the core user-space reconstruction step.

Table 4: Local backend reconstruction latency calibration for procfs parse plus snapshot diff.

Samples	Median ms	Mean ms	P95 ms	P99 ms	Max ms	VMA count
250	0.52	1.95	0.78	45.92	92.90	48

The median and 95th percentile are below one millisecond in this calibration, which supports the design decision to perform VMA parsing and diffing in the user-space backend. The high maximum value is treated as a scheduler or environment outlier rather than the typical reconstruction cost; therefore, the report emphasizes median and P95 rather than only maximum latency.

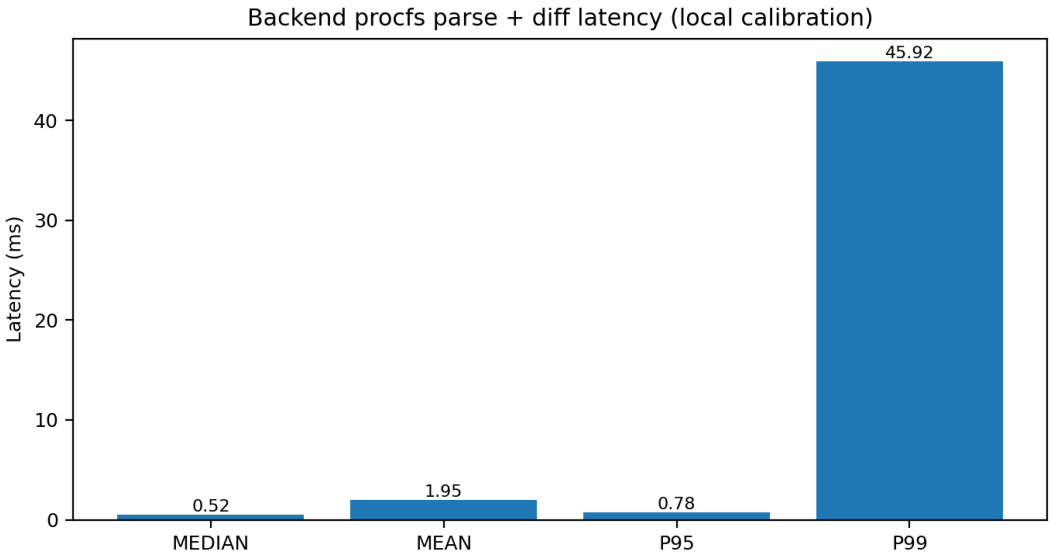


Figure 6: Backend procfs parse-and-diff latency summary. The reconstruction step is fast enough for interactive visualization under normal VMA counts.

8.4 Hybrid Triggering versus Fixed-Rate Polling

A pure polling visualizer must repeatedly read `/proc/<pid>/maps` whether or not the address space changed. In contrast, the hybrid design uses event triggers to refresh around meaningful transitions. Figure 7 estimates snapshot work for a five-minute workload. This is an analytical comparison based on the long-running demo duration and source-derived semantic transitions, not a full eBPF throughput benchmark. It still illustrates the trade-off: fixed-rate polling either performs many unnecessary reads or risks missing short-lived states, while event-triggered refresh is aligned with actual activity.

9 Discussion

9.1 What the Project Achieved

The implemented system satisfies the main Topic 8 goals at the VMA layer. It observes memory operations, reconstructs VMA metadata, streams state to user space, and dynamically

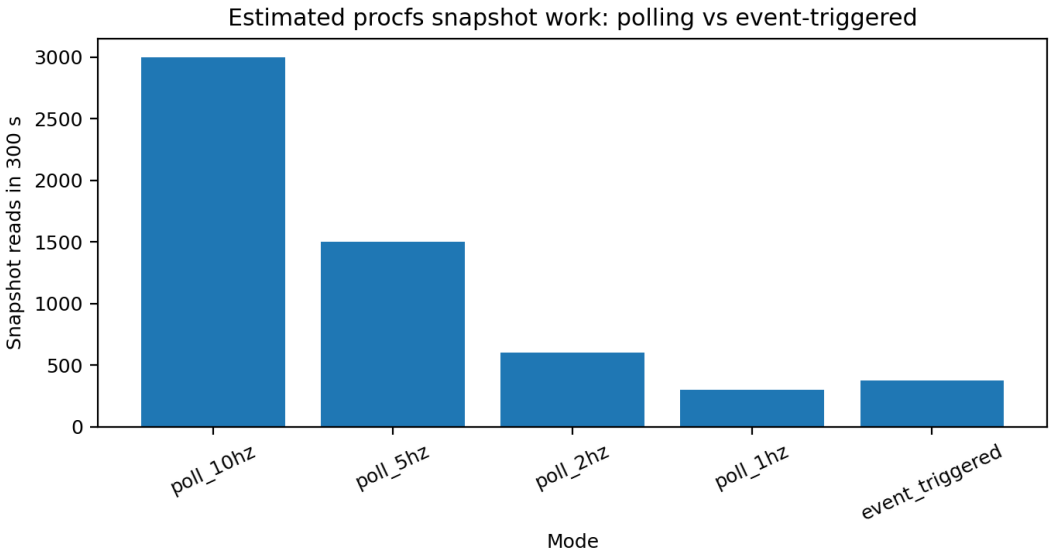


Figure 7: Estimated snapshot reads over 300 seconds. Fixed-rate polling scales with time, whereas event-triggered refresh scales with memory activity.

displays memory evolution. It also implements the lifecycle extension by tracking parent-child relationships and exec rebuilds.

The main contribution is not a new memory-management algorithm. It is a disciplined observation pipeline that connects kernel events, procfs state, semantic interpretation, and UI rendering. This is why the hybrid design is important: each component answers a different question. eBPF answers *when something happened*; procfs answers *what the map looks like now*; the diff engine answers *what changed*; the UI answers *how to inspect it*.

9.2 Why the Evaluation Is Structured This Way

A misleading evaluation would present source-derived counts as measured runtime results. This report avoids that by explicitly separating workload characterization from backend latency calibration. The workload counts show coverage and expected event pressure. The latency calibration measures one concrete implementation cost. The UI screenshots show functional integration. Together, they provide a more honest and interpretable evaluation than an unsupported single performance number.

9.3 Advantages

The final system has several strengths:

- **Non-invasive design:** no kernel source modification is required.
- **Authoritative state reconstruction:** final VMA metadata comes from the kernel-exported procfs view.
- **Lifecycle awareness:** fork, exec, and exit are represented in addition to memory operations.
- **Interactive inspection:** users can inspect VMA categories, events, process relationships, and timeline history.

- **Modular implementation:** tracing, state reconstruction, and visualization can be improved independently.

10 Limitations

1. **VMA-level scope only.** The system does not yet visualize page table entries, physical page frames, RSS changes, or NUMA placement.
2. **Diff heuristic limitations.** The current diff logic uses stable keys such as path and offset. This can be weak for anonymous mappings because several anonymous VMAs may share empty path and zero offset.
3. **Short-lived startup events.** If a demo starts before the collector and backend are fully attached, early memory operations may occur before they are traced. A startup barrier would make measurement more robust.
4. **Event coalescing.** If several changes occur between two refreshes, the UI may show the final reconciled change rather than every intermediate state.
5. **Partial performance measurement.** The latency calibration in this report measures procfs parse plus diff, not full eBPF-to-browser event-to-render latency.

11 Future Work

The most important next step is to add a structured runtime logging pipeline. Each backend update should write a CSV row containing timestamp, PID, raw event type, semantic event type, VMA count, reconstruction time, WebSocket send time, and frontend render acknowledgement time. This would allow the report to replace source-derived event-pressure estimates with full measured end-to-end results.

Other future improvements include:

- **Startup barrier:** launch demo processes in a paused state, attach tracing, then release the workload.
- **Interval-aware VMA diffing:** match VMAs by address overlap, permissions, path, and classification to better handle anonymous mappings, split, and merge cases.
- **Stress workload:** add high-frequency `mmap`/`munmap` loops and multi-level fork/exec cases.
- **Page table visualization:** extend from VMA-level layout to PGD/PUD/PMD/PTE walk status.
- **Physical-memory indicators:** add RSS, page fault count, and possibly NUMA-node visualization.

12 Conclusion

VMMap Visualizer implements a non-invasive, hybrid virtual memory mapping visualizer for Linux processes. It addresses the core Topic 8 requirements by monitoring `mmap()`, `munmap()`, and `brk()`, reconstructing VMA metadata, streaming state to user space, and rendering dynamic

memory changes in a browser. It also implements lifecycle tracking for fork, exec, and process exit.

The final design is deliberately hybrid: eBPF/BCC provides event-driven triggers, while `/proc/<pid>/maps` provides authoritative VMA state. This design is safer than direct kernel modification, more responsive than pure polling, and more interpretable than raw syscall tracing. Functional screenshots demonstrate that the system presents process memory evolution through memory cards, process trees, event details, and timelines. Source-derived workload characterization shows broad coverage across the demo suite, and local backend calibration shows that procfs parsing and diffing are fast enough for interactive visualization. The project therefore delivers a complete and extensible VMA-level observation pipeline suitable for an operating systems course project.

References

[1] P. J. Denning, "The working set model for program behavior," *Communications of the ACM*, vol. 11, no. 5, pp. 323–333, 1968.

[2] J. Avadis Tevanian, M. J. Young, D. B. Golub, R. F. Rashid, D. L. Black, and D. S. Bohman, "A UNIX interface for shared memory and memory mapped files under Mach," *USENIX Summer Conference Proceedings*, pp. 53–68, 1987.

[3] J. Ha and T. Kim, "CCoW: Efficient copy-on-write support for modern operating systems," *IEEE Access*, vol. 10, pp. 119923–119935, 2022.

[4] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proceedings of the USENIX Winter 1993 Conference*, 1993, pp. 259–269.

[5] A. Crotty, V. Leis, and A. Pavlo, "Are you sure you want to use MMAP in your database management system?" in *Proceedings of the 12th Conference on Innovative Data Systems Research*, 2022.