# It's the Kernel's Fault! Custom Page Fault Handling With bpf_fault

Tal Zussman [1] Riju Dey [1] Hasan Zengin [1] Yiming Fang [1] David Hildenbrand [2,3] Asaf Cidon [1]

[1]Columbia University [2]Technical University of Munich [3]Arm Germany GmbH

## Problem Statement

- Page faults are traditionally handled entirely by the OS. However, many applications benefit from running **custom page fault handling logic**: VM live migration and snapshots, garbage collection, far and tiered memory, memory debugging, and more
- Linux's only mechanism for this is `userfaultfd` (`uffd`), which offloads fault handling to userspace
- Unfortunately, `uffd` suffers from high overhead, poor scalability, and a design that precludes its use in libraries

## Background: `userfaultfd`

- A process registers an address range with `uffd` and creates a dedicated *fault-handling thread*
- On a fault, the kernel blocks the faulting thread and enqueues a fault message
- The handler thread reads the message, prepares the page, and resolves the fault with an `ioctl` (e.g., `UFFDIO_COPY`)

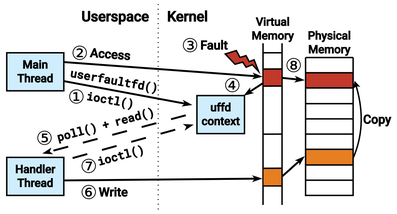



Figure 1. Handling page faults with `userfaultfd`.

## Limitations of `userfaultfd`

### Overhead

- Every fault requires context switches and extra kernel-user crossings: **70%** of a fault is spent on `uffd`-inflicted overhead
- A missing read fault takes 10.2 µs with `uffd` vs. 0.75 µs in the kernel (13.5× slower)

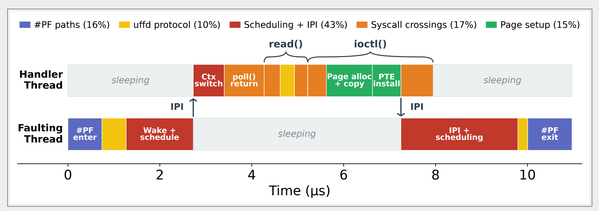



Figure 2. Critical-path breakdown of a single `uffd` missing fault.

### Scalability

- The single handler thread becomes a serializing bottleneck: from 1 to 512 threads, `uffd` takes 616× longer

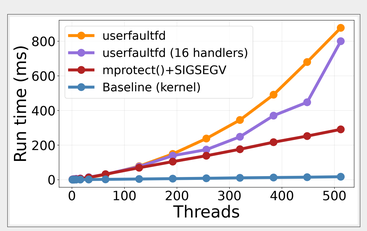



Figure 3. Scalability benchmark run time.

### Applicability

- The handler thread is not cloned on `fork()`, which precludes usage in libraries (e.g., the dynamic linker). No missing-fault support for file-backed memory.

### Security

- `uffd` can indefinitely block kernel execution, enabling TOCTOU exploits. Docker and Podman block it by default, and unprivileged `uffd` is now disabled by default.

## Our Approach: eBPF Fault Handlers

We want to run **custom fault handlers directly in the Linux kernel**, without sacrificing performance.

- eBPF allows userspace programs to run safely in the kernel. The kernel *verifies* them in advance to ensure they are bounded and memory-safe.
- We can use eBPF to run custom fault handlers in-kernel, resulting in a much simpler approach than `uffd` and fewer downsides.

## bpf_fault: Design and Implementation

- `bpf_fault` shifts the role of the fault-handling thread into the kernel as an eBPF `struct_ops` program
- Handlers run **in the faulting thread's context**: no context switches, handler thread, or extra data copies
- Supports anonymous and shmem missing, WP, and minor faults, as well as **file-backed memory**, which `uffd` does not support

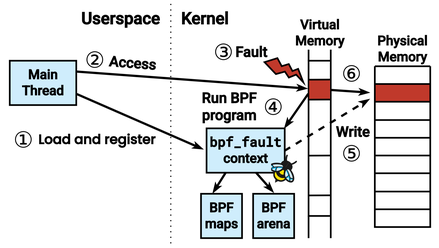



Figure 4. bpf_fault's fault-handling path.

### Interface

- One callback per fault type, invoked with the faulting address and the page
- Zero-copy: handlers write directly into the new page, with verifier-enforced bounds
- Returning zero installs the page. A non-zero return delivers `SIGBUS`, unlike `uffd`, where a failed handler leaves the faulting thread blocked.

```
struct bpf_fault_ops_ctx {
  unsigned long address;
  unsigned long real_address;
  __u8 fault_type;
};

struct fault_ops {
  int (*handle_missing_fault)(
      struct bpf_fault_ops_ctx *ctx,
      unsigned char *page);
  int (*handle_minor_fault)(...);
  int (*handle_wp_fault)(...);
  int (*handle_fork)(
      struct bpf_fault_fork_ctx *ctx);
  ...
};
```

Figure 5. bpf_fault's `struct_ops` interface.

### Registration and Fault Path

- An application attaches its handlers to one or more address ranges with a single libbpf call
- On a fault, the kernel releases the VMA lock, allocates a page, and invokes the eBPF program. It then re-locks, validates the VMA, and installs the PTE.
- VM events (e.g., `fork()`, `munmap()`) become in-kernel callbacks, and children can inherit their parent's handlers

### Security

- The eBPF verifier checks handlers at load time, guaranteeing bounded loops and provably safe memory accesses
- Handlers are **non-sleepable by default**, closing off common `uffd` TOCTOU exploits
- Sleepable handlers are opt-in and much harder to exploit: the blocking operations available to eBPF are bounded and cannot wait on an attacker-controlled event
- A slow handler only delays the process that installed it

## Microbenchmarks

| | `uffd` | | `bpf_fault` | | Baseline | |
|---|---|---|---|---|---|---|
| Fault type | Avg | p99 | Avg | p99 | Avg | p99 |
| Missing (anon) | 11.6 | 16.8 | 2.7 | 7.8 | 2.0 | 7.0 |
| Write-protect | 10.8 | 13.9 | 1.6 | 1.8 | – | – |
| Minor (shmem) | 10.3 | 13.8 | 3.7 | 8.8 | 2.6 | 7.6 |

Table 1. Per-fault latency (µs) for write-triggered faults.

- 2.8–6.7× lower fault latency than `uffd`, with only 1.4× overhead over the kernel
- `bpf_fault` scales linearly: at 512 threads, it takes 25 ms vs. 859 ms for `uffd` (34× faster)

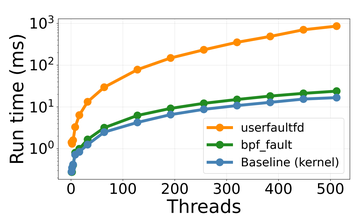
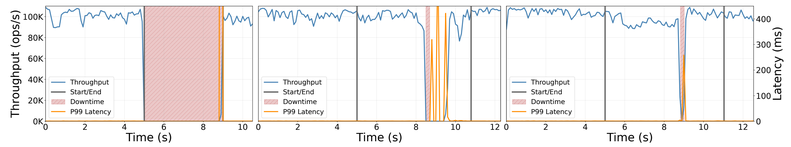



Figure 6. `uffd`, bpf_fault, and baseline scalability.

## VM Live Snapshots

- Live snapshots write-protect guest RAM and save pages in the background. With `uffd`, vCPU writes to unsaved pages block until the handler thread saves them.
- With `bpf_fault`, the eBPF handler copies the page's pre-image into a ring buffer and returns immediately, so **the vCPU never blocks**
- We integrated `bpf_fault` with live snapshots in **Firecracker** and **QEMU**



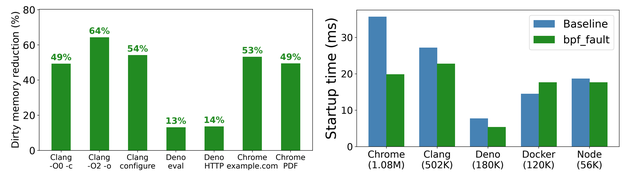

(a) Synchronous. (b) `uffd` live. (c) bpf_fault live.

Figure 7. Redis throughput and tail latency while taking a snapshot of an 8 GB Firecracker VM.

- In Firecracker, `bpf_fault` achieves **2–4× higher throughput** and **5–111× lower average latency** than `uffd` during snapshots

## Lazy Dynamic Linking

- Linux's dynamic linker eagerly applies relative relocations at load time, dirtying pages and increasing startup time for relocations that may not be used
- We use `bpf_fault` to **defer relocations to fault time**: on first access to a page, the eBPF handler applies only that page's relocations
- Integrated into the glibc dynamic linker (`ld.so`), with no changes to the ELF format. Impossible with `uffd`, which lacks file-backed fault handling and requires a handler thread.
- `ld.so` registers each object's writable segments in a per-process **interval tree** in a BPF arena, which the handler searches on each fault




(a) Dirty memory reduction. (b) Application startup time.

Figure 8. Dynamic linking with bpf_fault.

- Chrome: **50% less dirty memory** (95 MB) and **44% faster startup**. Uses 23% less physical dirty memory than Chrome's zygote process.
- Clang: 49–64% less dirty memory and 7–11% faster compilation

## More Applications

**Garbage Collection**: We push the compaction phase of a JVM garbage collector into the kernel, **cutting GC pauses by up to 42%** while exceeding stop-the-world throughput.

**Memory Debugging**: Our electric fence memory debugger is **up to 4.4×** faster than `uffd`'s handler mode, while always correctly handling kernel-originated faults.

## Conclusion

`bpf_fault` enables fast, scalable, and safe custom page fault handling in Linux using eBPF, improving application performance, lowering resource usage, and enabling novel use cases. More broadly, `bpf_fault` provides a platform for experimenting with memory management policies that were previously fixed in the kernel or required sacrificing performance.
Full paper and code at `github.com/bpf-fault/bpf-fault`