Copy-on-Write (COW) Optimization Study

CSC5031 Operating Systems Project — Topic 7

Yutong Liu
Student ID: 225040151
Requirement 1: Latency Measurement
Benchmark Design & Experimental Evaluation

Longjie Su
Student ID: 225040158
Requirement 2: Pre-Copy Implementation
Kernel Build & Infrastructure

***Abstract*—Copy-on-Write (COW) accelerates the `fork()` system call by deferring page duplication until a write occurs, but write faults remain expensive. This project presents an end-to-end study of COW behavior in Linux 6.8: we instrument the kernel to measure fault latency with nanosecond precision, then design a speculative pre-copy optimization based on spatial locality. Measurement reveals that a 4KB page copy takes only ∼350 ns (5% of total fault cost), while trap handling and TLB maintenance account for 95%. The pre-copy mechanism reduces runtime by 17% for sequential workloads and 15% for random workloads, halving the number of COW faults in the best case. The optimization is safe, runtime-toggable, and demonstrates that eliminating page faults yields far greater benefit than accelerating the copy itself.**

***Index Terms*—Copy-on-Write, COW, fork(), page fault, memory management, Linux kernel, pre-copy, speculative execution**

I. INTRODUCTION

The `fork()` system call is the standard UNIX mechanism for creating new processes. A naive deep copy of the parent address space is both slow and wasteful, especially because most children immediately call `exec()`, discarding the duplicate memory. Copy-on-Write (COW) elegantly solves this problem by marking shared pages read-only; a private copy is created only when a process attempts a write. However, the resulting page fault still incurs significant overhead from hardware exception handling, kernel trap entry, page table manipulation, and TLB flushes.

This project pursues two complementary objectives:

1) **Latency measurement:** precisely characterise the cost of a COW fault in the Linux kernel, from the hardware trap to the return to user space.
2) **Speculative pre-copy:** exploit spatial locality to pre-copy the neighbouring page during a fault, thereby reducing the total number of faults and improving throughput.

We modify the Linux 6.8 kernel (Ubuntu 6.8.0-106-generic tree, compiled as 6.8.12-cow-topic7) with instrumentation in the page fault handler, a `/proc` statistics interface, and a runtime-toggable pre-copy mechanism. Experiments under sequential, random, and single-page workloads provide a comprehensive performance evaluation.

II. BACKGROUND

A. Virtual Memory and COW Mechanism

Modern operating systems use virtual memory to provide per-process address spaces. The hardware MMU translates virtual addresses through multi-level page tables. COW exploits this indirection: after `fork()`, the parent and child share the same physical pages, which are marked read-only in both page tables. The moment a write is attempted, the hardware raises page fault exception #14 (x86). The kernel then allocates a new frame, copies the 4KB content, updates the PTE to writable, and resumes the faulting instruction.

B. Linux COW Fault Path

The relevant kernel code resides in `mm/memory.c`. The call chain is:

`do_page_fault()` → `handle_pte_fault()` → `do_wp_page()` → `wp_page_copy()`

The function `wp_page_copy()` is responsible for the actual page allocation and copy; it is the precise location of our instrumentation and pre-copy logic.

C. Motivation for Pre-Copy

Our early measurements (Section VI) show that a bare-metal 4KB memcpy takes roughly 350 ns, while the full write fault costs about 6.8 µs. In other words, 95% of the latency comes from factors other than the copy itself. If spatial locality holds—a process that writes page N often writes $N+1$ next—we can pre-copy page $N+1$ inside the current fault handler, turning a future expensive trap into a low-overhead in-kernel copy.

III. IMPLEMENTATION

All modifications are applied to Linux 6.8.12 based on the Ubuntu 6.8.0-106-generic tree and centre on `mm/memory.c`.

A. Requirement 1: COW Latency Measurement

1) Measurement Design: We wrap the actual copy operation inside `wp_page_copy()` with `ktime_get()` calls. The following lock-free atomic64 counters are maintained:

- `cow_fault_count`: total COW faults handled.
- `cow_copy_total_ns`: cumulative copy time.

- `cow_copy_max_ns`: maximum single-copy latency, updated via a CAS loop to avoid contention.
- `cow_precopy_attempts` & `cow_precopy_success` (for Requirement 2).

Derived metrics such as `cow_copy_avg_ns` are computed on read.

```
/* Inside wp_page_copy() */
ktime_t __cow_start = ktime_get();
err = __wp_page_copy_user(&new_folio->page,
    vmf->page, vmf);
s64 __cow_ns = ktime_to_ns(ktime_sub(ktime_get
    (), __cow_start));

atomic64_inc_return(&cow_fault_count);
atomic64_add(__cow_ns, &cow_copy_total_ns);

/* Lock-free maximum update */
s64 __prev_max = atomic64_read(&
    cow_copy_max_ns);
while (__cow_ns > __prev_max &&
!atomic64_try_cmpxchg(&cow_copy_max_ns, &
    __prev_max, __cow_ns))
;
```

To avoid flooding `dmesg`, `printk` is emitted only once every `cow_sample_interval` faults (default 100).

3) Proc and Sysctl Interface: Three runtime interfaces are provided:

1) `/proc/cow_stats` — read for cumulative statistics; write any value to reset all counters.
2) `/proc/sys/vm/cow_precopy` — 0 to disable, 1 to enable the pre-copy optimization (Requirement 2).
3) `/proc/sys/vm/cow_sample_interval` — controls the printk sampling rate.

A typical interaction looks like:

```
$ cat /proc/cow_stats
faults: 49242  total_ns: 16543417  max_ns:
    12924
precopy_attempts: 26580  precopy_success:
    24580

$ echo 1 > /proc/sys/vm/cow_precopy
$ echo 1 > /proc/cow_stats   # reset counters
```

B. Requirement 2: Speculative Pre-Copy Optimization

1) Pre-Copy Algorithm: We introduce `cow_try_precopy()`, called after a successful COW copy on page N. The function:

1) Computes the address of page $N + 1$.
2) Validates that $N + 1$ is a COW candidate (five safety checks, see below).
3) If valid, allocates a new page, copies the content, and installs a writable PTE for $N + 1$.
4) If any check fails, returns silently without affecting the current fault.

2) Safety Checks and Race Handling: All five conditions must hold:

1) `next_addr < vma->vm_end` (inside VMA).
2) Same PMD range as the faulting page.
3) PTE present and read-only.
4) Page is anonymous (not file-backed).
5) Page is non-exclusive (truly shared for COW).

To guard against concurrent modifications, we obtain a reference on the old page via `folio_get()` before releasing the PTL, then re-verify the PTE with `pte_same()` after re-acquiring the lock. If the PTE has changed, the newly allocated page is freed and the pre-copy is abandoned.

```
static void cow_try_precopy(struct
    vm_area_struct *vma,
struct mm_struct *mm,
pmd_t *pmd,
unsigned long fault_addr);
```

The function adds about 80 lines to `mm/memory.c` and is guarded by a global `cow_precopy_enabled` flag.

IV. Kernel Build and Deployment

The custom kernel was built on an AMD 16-core machine with 16 GB RAM running Ubuntu 22.04.5 LTS. The procedure:

1) Download the Ubuntu Linux 6.8 source tree.
2) Patch `mm/memory.c` with measurement and pre-copy code.
3) Configure via `make olddefconfig`.
4) Compile with `make -j16` and install modules.
5) Generate initramfs and update GRUB.
6) Port the AIC8800 USB WiFi driver to maintain remote SSH access.
7) Boot into `6.8.12-cow-topic7` and verify through `/proc/cow_stats`.

V. Experimental Setup

A. Environment

- CPU: AMD x86_64, 16 cores
- RAM: 16 GB
- OS: Ubuntu 22.04.5 LTS
- Modified kernel: `6.8.12-cow-topic7`
- Page size: 4 KB (base pages)

B. Benchmark Programs

Two user-space tools were developed:

- `cow_test.c`: verifies COW memory isolation by having parent and child modify alternating pages and checking invariants.

- `cow_bench.c`: measures single-page, sequential, and random write workloads after `fork()`, using `clock_gettime()` for wall-clock timing.

C. Methodology

For each workload, we first run with `/proc/sys/vm/cow_precopy`=0 to obtain a baseline, then enable pre-copy and collect the same metrics. Every test is repeated 5 (16 MB) or 3 (64 MB) times; average values are reported.

VI. EXPERIMENTAL RESULTS

A. COW Latency Measurement

1) Full Fault Latency: End-to-end user-space measurement of a single COW write fault yields an average of **6,782 ns** (∼6.8 µs), with a range of 5,801–12,212 ns. The anomalous maximum is attributable to occasional scheduling preemption.

2) Kernel Copy Latency: The pure `__wp_page_copy_user()` operation takes **320–400 ns** on average, contributing only about 5% to the total fault cost.

TABLE I
COW COPY LATENCY PERCENTILES (NS, 4KB PAGE)

Metric	Value
Min	90
P50	361
P90	631
P95	711
P99	862
Max	1,082

3) Copy Latency Distribution: The tight distribution confirms that the copy itself has very predictable performance. Occasional cache misses push the tail beyond 1,000 ns.

4) Overhead Breakdown:

- 4KB page copy: ∼350 ns (5%)
- Trap entry/exit, page table walk, rmap accounting, TLB flush: ∼6,450 ns (95%)

5) Workload Scaling: Random access shows about 8–10% higher latency than sequential access because of poorer TLB and cache locality. Increasing the working set from 16 MB to 64 MB raises user-space fault latency only slightly; the kernel copy component remains stable at 320–400 ns.

B. Pre-Copy Performance

1) Sequential Workloads: For 16 MB (4096 pages): runtime drops from 8.989 ms to 7.462 ms (**-17.0%**); for 64 MB (16384 pages): from 36.692 ms to 30.169 ms (**-17.8%**). COW faults are reduced by **49.7%** and the pre-copy hit rate reaches **99%**. Practically every speculatively copied page is used.

2) Random Workloads: For 16 MB: 9.433 ms → 8.024 ms (**-14.9%**); for 64 MB: 39.003 ms → 33.291 ms (**-14.6%**). Faults drop by **36.3%** with a hit rate of **57%**. Even though the access order is randomised, more than half of the pre-copies are eventually written, yielding a net speedup.

3) Single-Page Write: A workload that touches only one page after `fork()` sees a 17% overhead (7,935 ns vs. 6,782 ns) because pre-copy always tries and fails. This overhead is fully amortised in any bulk scenario.

4) Tail Latency Improvement: Pre-copy reduces the P99 copy latency by 14% (862 → 741 ns) and the maximum copy latency by 28% (1,082 → 781 ns), as pre-copied pages avoid their own fault and the associated scheduling jitter.

C. Correctness Verification

`cow_test.c` confirms that parent and child see distinct pages after writing, that data written by one does not leak to the other, and that no kernel panic, OOPS, or corruption occurs when pre-copy is active.

VII. ANALYSIS AND TRADE-OFFS

A. Why Pre-Copy Works

The central insight is that the cost of a full page fault (∼6.8 µs) dwarfs the cost of an in-kernel copy (∼550 ns). Under sequential access, spatial locality is almost perfect: writing page N strongly implies $N+1$ is next. Pre-copy turns a future expensive trap into a cheap copy, halving the number of faults.

Even with random access, after `fork()` all anonymous pages are COW- shared, so every page will eventually be written. The 57% hit rate means more than half of the speculative copies are used before exit; the remaining 43% wasted copies are still affordable given the net gain.

B. Cost-Benefit Economics

TABLE II
PRE-COPY COST-BENEFIT BREAKDOWN

Item	Latency (ns)
Full fault trap	6,800
In-kernel pre-copy	550
Savings per hit	6,250
Break-even hit rate	8.5%
Actual hit rate (seq)	99%
Actual hit rate (rand)	57%

The break-even point is only 8.5%, which is far below the measured hit rates. Hence pre-copy delivers a robust net benefit across all bulk workloads.

C. Performance by Access Pattern

Sequential: 99% hit rate → 50% fewer faults → ∼17% faster. Nearly ideal scenario.

Random: 57% hit rate → ∼36% fewer faults → ∼15% faster. Waste is tolerable.

Single page: always fails → +17% overhead per fault, but such sparse writes are rare and unaffected when pre-copy is disabled.

D. Memory and Overhead

Pre-copy allocates one extra 4KB page per attempt. In high-hit scenarios this memory is immediately productive; in memory-constrained environments the sysctl toggle allows the admin to disable the feature. The per-fault overhead consists of PTE lookup, allocation, and a memcpy—all bounded and small relative to the fault trap cost.

E. Robustness Guarantees

The five safety checks, combined with `folio_get()` pinning and post- allocation PTE re-verification, ensure that pre-copy never violates process isolation, COW semantics, or kernel stability. Extensive testing confirmed zero crashes or data corruption.

F. Key Takeaways

1) **Fault count dominates performance.** Eliminating one trap saves 18 times the cost of a speculative copy.
2) **Spatial locality is highly exploitable.** Simple 1-page look-ahead achieves near-optimal gains for sequential workloads.
3) **Speculation can be safe.** With careful VMA/PMD checks and double-checked locking, kernel-level speculation is practical.

VIII. LIMITATIONS

- Only 4KB base pages are supported; Transparent Huge Pages (2MB) not yet handled.
- Pre-copy depth is fixed at one page; deeper look-ahead not explored.
- Evaluation covers single-child `fork()`; multi-process and multi-threaded scenarios remain untested.
- Pages crossing a 2MB PMD boundary are conservatively skipped.
- The heuristic assumes spatial locality, which may not hold for all applications.

IX. FUTURE WORK

1) **Adaptive depth**: track per-VMA hit rates to dynamically adjust the number of pages pre-copied.
2) **Huge page support**: break a THP into 4KB sub-pages on partial write to reduce large-copy latency spikes.
3) **Workload detection**: auto-enable pre-copy for sequential VMAs and disable it for truly random patterns.
4) **Delayed allocation**: explore deferring copy until page reclaim, as an alternative COW strategy.
5) **Extended benchmarking**: investigate multi-child fork, KVM, and container workloads.

X. CONCLUSION

This project delivered a complete measurement and optimisation study of Copy-on-Write in the Linux 6.8 kernel. Precise instrumentation showed that the 4KB page copy accounts for only 5% of the fault latency, while trap handling and TLB maintenance dominate. Based on this observation, a speculative pre-copy optimisation was implemented that reduces runtime by 15–17% for typical bulk workloads by eliminating nearly half of the page faults. The mechanism is safe, runtime-controllable, and demonstrates that reducing the number of fault traps is far more effective than accelerating the copy operation itself.

XI. ACKNOWLEDGMENT

We thank the instructor and teaching assistants of CSC5031 for their guidance and support.

REFERENCES

[1] Linux Kernel Organization, *Linux Kernel Source Code: Version 6.8*, 2024.

[2] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, 2005.

[3] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer Manuals*, 2024.

[4] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Wiley, 2018.

[5] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Communications of the ACM*, vol. 15, no. 3, pp. 135–143, 1972.

[6] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed. O'Reilly Media, 2005.

[7] Linux Kernel, `mm/memory.c`, functions `do_wp_page` and `wp_page_copy`, v6.8.