Towards Programmable Memory Management with eBPF

Presented by Kaiyang Zhao <kaiyang2@cs.cmu.edu>

Carnegie Mellon University
Computer Science Department

Computer Architecture Operating Systems Group

Overview

- How eBPF can support programmable memory management
- Ample Memory Contiguity with Contiguitas
- Learned Virtual Memory

About me:

- I’m a researcher on memory management
- Trying to solicit feedback on the high-level idea

eBPF for Programmable Memory Management

- Programmability allows easy implementation of new ideas in MM
- eBPF is a good vehicle for programmability
- Implement the interfaces once, deploy many new MM policies later
- Related work includes sched_ext (scheduling) and TCP-BPF (TCP tuning)

eBPF for Programmable Memory Management

Many possibilities. Areas that can use flexible policies:

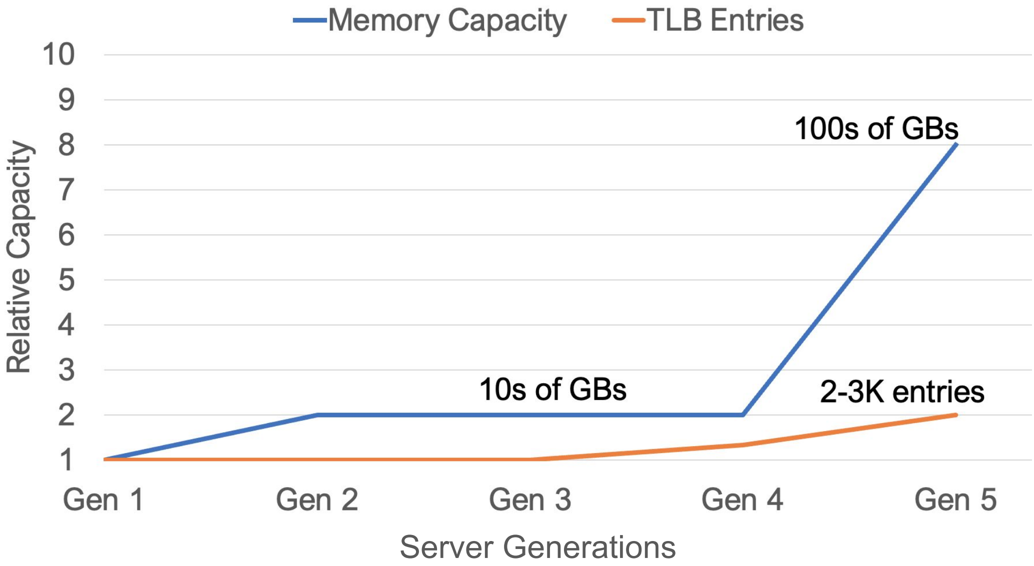
- Where do new page allocations go (NUMA node, tiered memory, etc.)
- How many huge pages to give to an app and of what size
- Memory reclaim (anon/file split, which process/cgroup to target)

In this talk:

- Two examples from our recent work that could benefit from programmable memory management

Why Memory Contiguity?

- Virtual memory overhead is severe and getting worse
- Up to 20% of CPU cycles have the page table walker active
- Most solutions to reduce the virtual memory overhead need contiguity



Why Memory Contiguity?

- Virtual memory overhead is severe and getting worse
- Most solutions to reduce the virtual memory overhead need contiguity

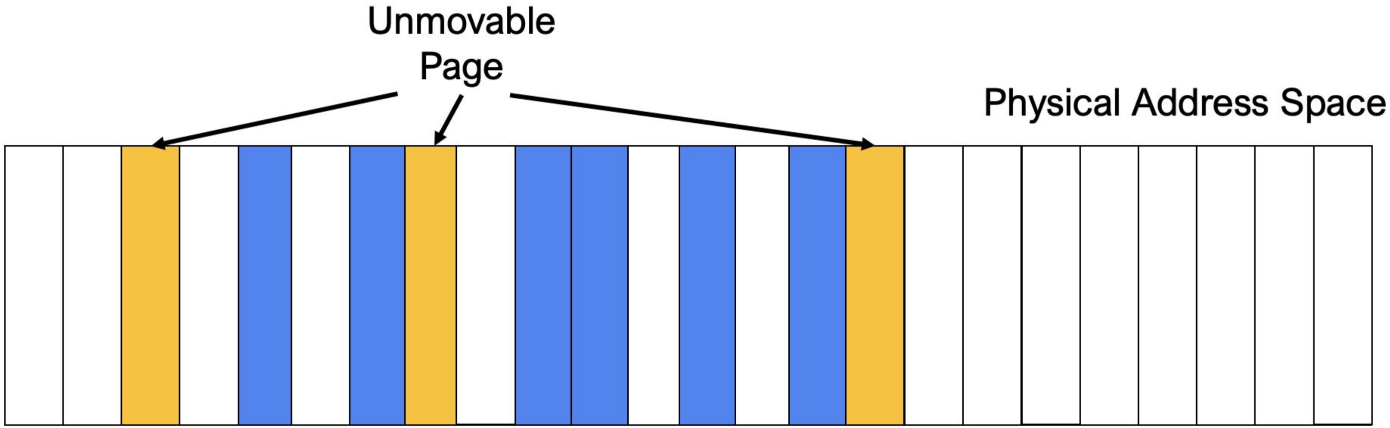
These solutions all need memory contiguity:

- Huge pages (THP, hugetlb)
- TLB coalescing (supported by ARM, RISC-V and AMD in current processors)
- Ongoing research in using contiguity to reduce the overhead

The goal:

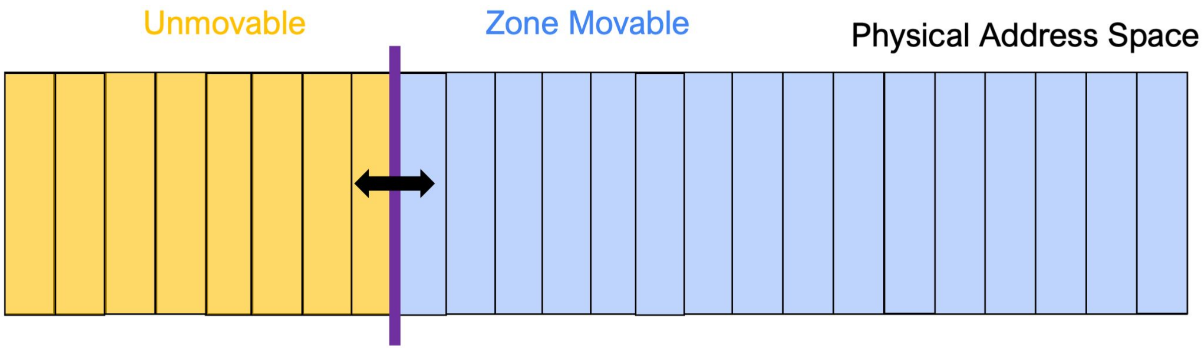
- Being able to obtain as-large-as-needed contiguous memory after compaction

Unmovable Pages Prevent Compaction



- One 4KB unmovable page makes compaction fail in a 2MB or 1GB range
- Only 0.19% of misplaced unmovable pages can block compaction completely
- They block contiguity from being recovered by compaction

Contiguitas – Improving ZONE_MOVABLE



- A movable zone where compaction will not be blocked by unmovables
  - We already have ZONE_MOVABLE
- Make ZONE_MOVABLE suitable for containing most of the memory
- Proactively migrate movable pages out of other zones to ZONE_MOVABLE
  - Compaction now can migrate pages to a different destination zone
  - Reduces the need for unmovable zones to grow

Contiguitas – Resizing

- Workload characteristics can change → may need to resize
- Empower an userspace agent to resize
  - Export # pages scanned on behalf of movable or unmovable allocations during reclaim
  - This approximates memory pressure. Can alternatively track memory pressure per type.
- Increasing the size of movable zone is best-effort
  - But unmovable zones can always back movable base page allocations – don't waste memory

Contiguitas – Results

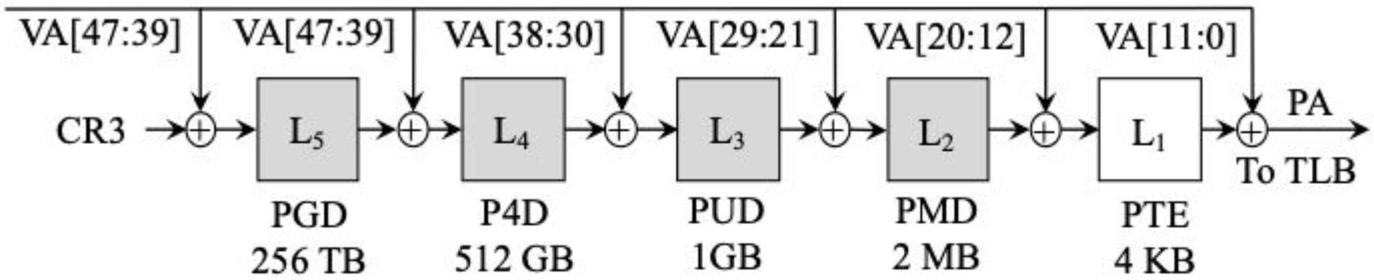
- 10% of memory is covered by the unmovable zone initially
  - Empirically determined by experiments at Meta
- Reliable THP and 1GB huge page allocation
- Up to 18% higher performance for Meta's production workloads
  - Web: +10% from the fragmented case using THP
  - Web: +18% when using 1GB huge pages
  - Cache A: +10% using THP
  - Cache B: +7% using THP
- Patches
  - https://lore.kernel.org/all/20230519123959.77335-1-hannes@cmpxchg.org/
  - https://lore.kernel.org/linux-mm/20240306041526.892167-1-hannes@cmpxchg.org/
  - https://lore.kernel.org/linux-mm/20240320024218.203491-1-kaiyang2@cs.cmu.edu/

* Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters, published on ISCA 2023

How Can MM Programmability Help

- A BPF program can hook to memory reclaim and decide whether to resize the regions and by how much
- A BPF program can classify the expected lifetime of allocations and direct the placement of pages such that fragmentation is reduced
- …

Learned Virtual Memory – Overview



- Linux assumes tree-like page table structures
- But recent work has shown that hash-based page table and more exotic paging schemes have great potential
- Difficult to evaluate novel paging designs with a realistic OS

Learned Virtual Memory – Overview

- An ongoing work of ours
- **Aims for single-memory-access page walk**
- Adapts to each application's mapped *virtual* address space with *learned indexes*
- Much higher coverage per byte of paging structure than radix page tables
    - Utilizes better the precious hardware page walk cache

How Can MM Programmability Help

The key is to get rid of the assumption of the tree-based page tables.

Interfaces for implementing new paging schemes in BPF:

- Establish a mapping (a virtual page becomes mapped)
  - An attachment point *mm_map_page(VA)*
  - BPF program returns a pointer to the page table entry of the newly mapped page
- Remove a mapping (a virtual page becomes unmapped)
  - An attachment point *mm_unmap_page(VA)*
- Find/update a mapping (returns the page table entry of a virtual page)
  - An attachment point *mm_get_pte(VA)*
  - Returns a pointer to the page table entry

How Can MM Programmability Help

An example of adapting existing code in the kernel to use the new interfaces

For __handle_mm_fault() that is part of the page fault handler, the new workflow becomes:

1. Call *mm_get_pte(VA)* to get the page table entry of the faulting address
   a. Currently it’s done by explicitly going level-by-level down the radix page table
   b. Turn the page table lookup into a black box provided by a BPF program
2. Identify the reason for the page fault (non-present page? write-protect? …)
3. Perform the appropriate next steps

- All code that interacts with page tables needs to be converted to use these 3 interfaces
- Anecdotally, it takes 10-20 person-months

How Can MM Programmability Help

- If these 3 interfaces are provided and used in Linux, the vast majority of novel paging schemes can be supported
- Tremendous benefits to the research community in prototyping and verifying novel designs on Linux

Some practical challenges remain:

- Many paging schemes (e.g., hashed page tables) require the allocation and management of a region of physical memory. Is it possible to support this in eBPF?
- How to allow a eBPF program to communicate with HW (e.g., set CR3)?

Summary and Questions

- Optimizing memory management is becoming increasingly important
- One line of research creates novel designs that don’t exist on commercial hardware
  - Being able to evaluate such designs on Linux is desirable
- Another line of research explores flexible policies in making decisions

1. Are more customizable policies and fewer assumptions about the paging scheme in MM possible to support in Linux?
2. Is eBPF the right vehicle for it?