Speed `read()`ing

Accelerating RocksDB Reads Using eBPF

Jeremy Carin and Tal Zussman

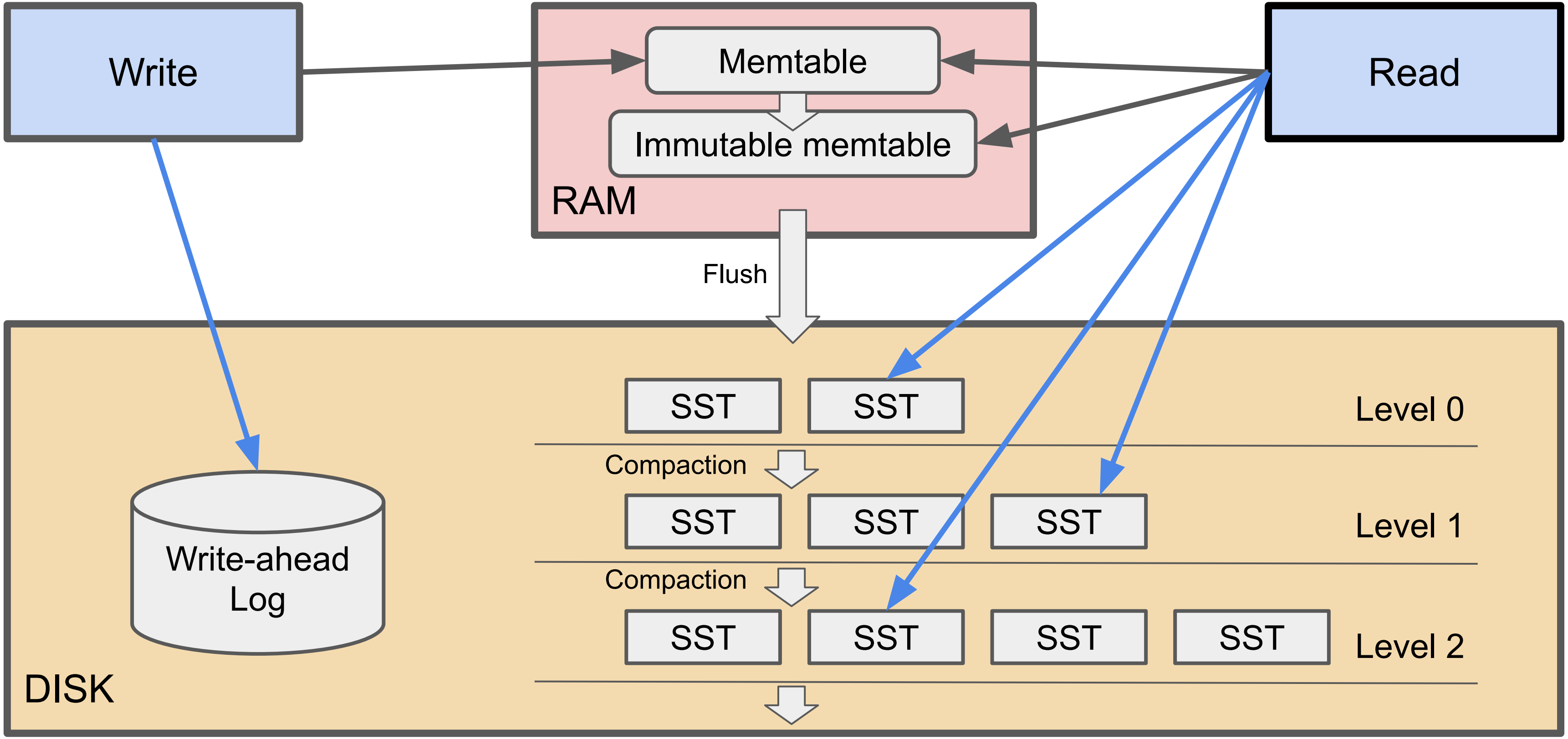
RocksDB

RocksDB

Background

- Popular embedded key-value storage engine
- Developed by Facebook, based on Google’s LevelDB
- Highly-optimized, written in C++, very fast

RocksDB: Log-Structured Merge (LSM) Tree



Data Blocks

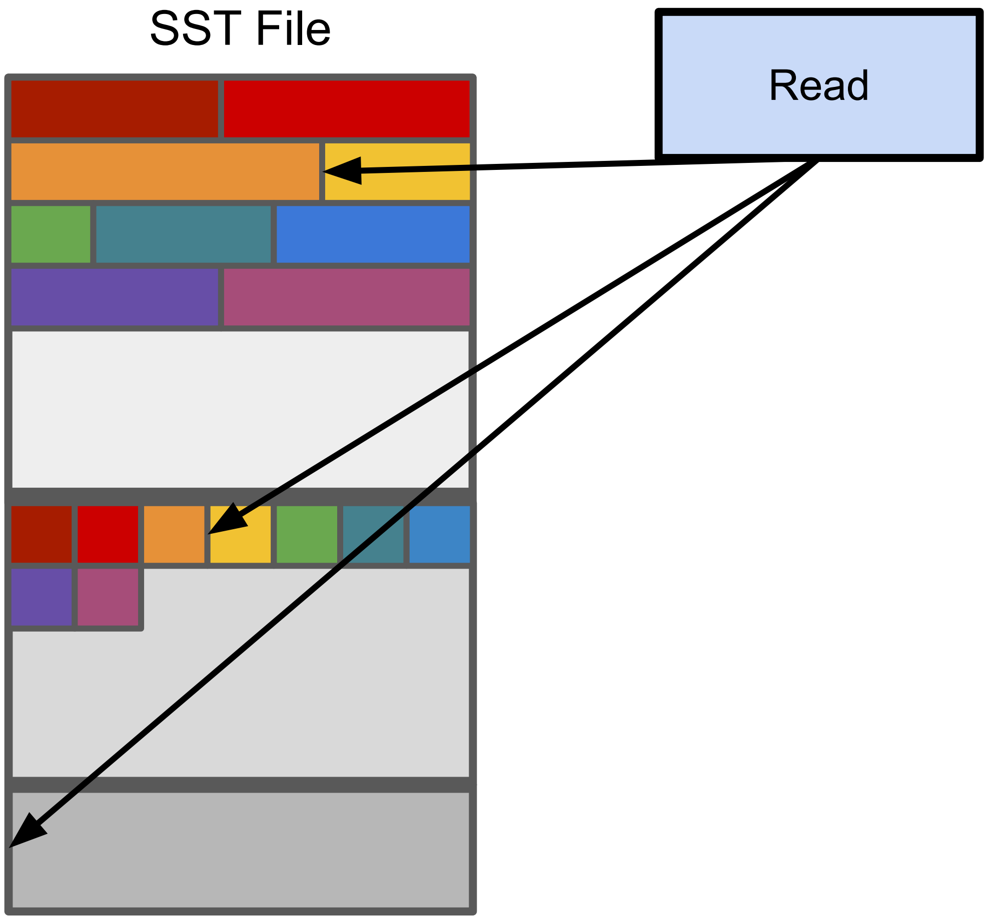
- Each block stores many key-value pairs in a range
- Key may or may not be in data block

Index block

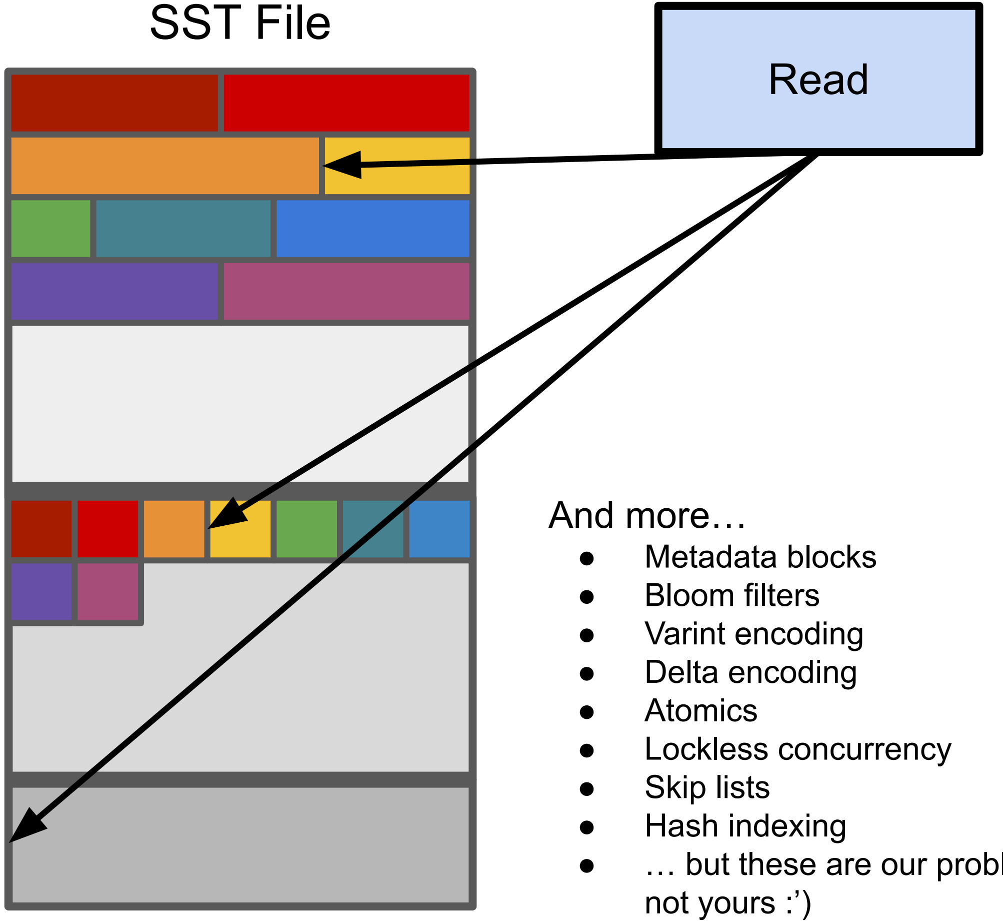
- Stores offsets of data blocks for key range

Footer

- Stores file metadata
- Usually cached



SST File



Data Blocks

- Each block stores many key-value pairs in a range
- Key may or may not be in data block

Index block

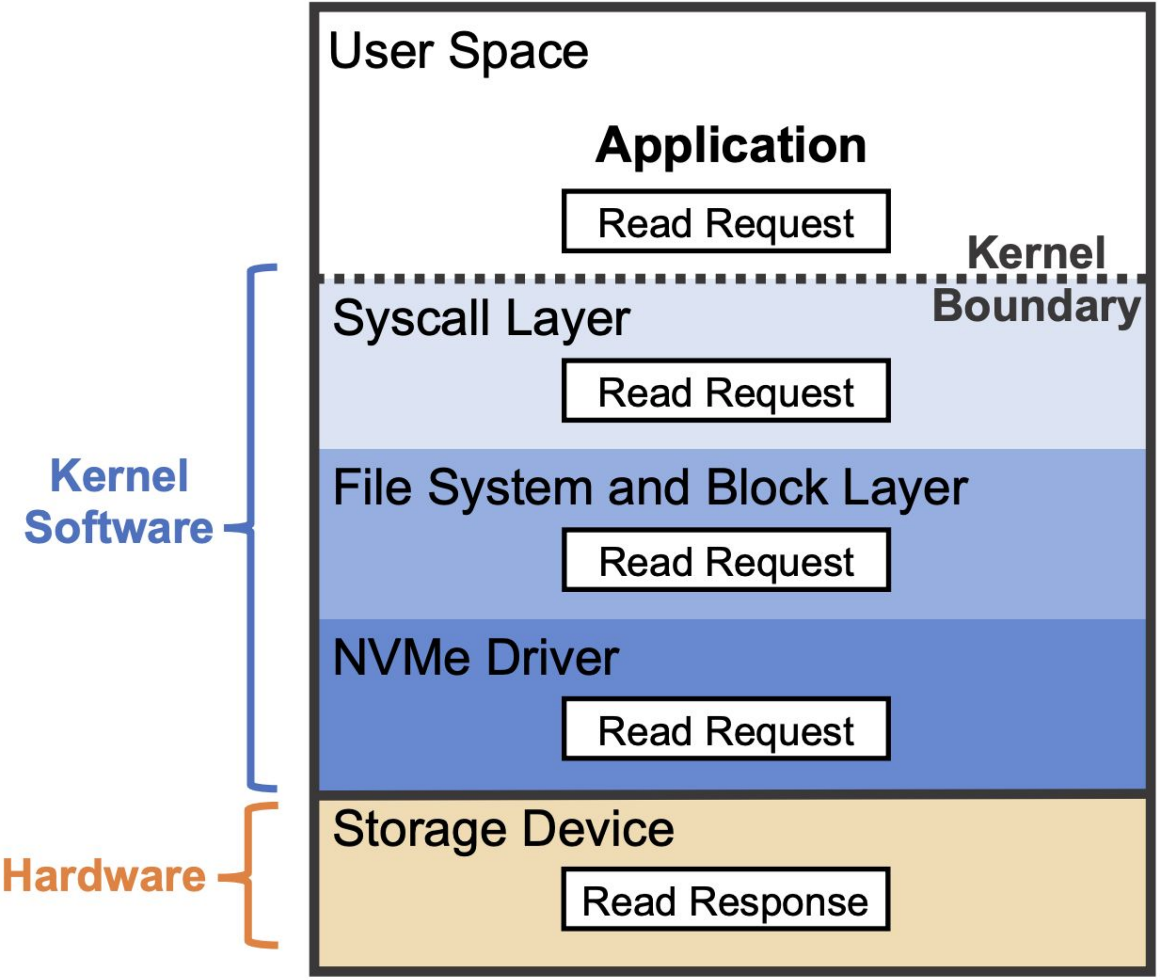
- Stores offsets of data blocks for key range

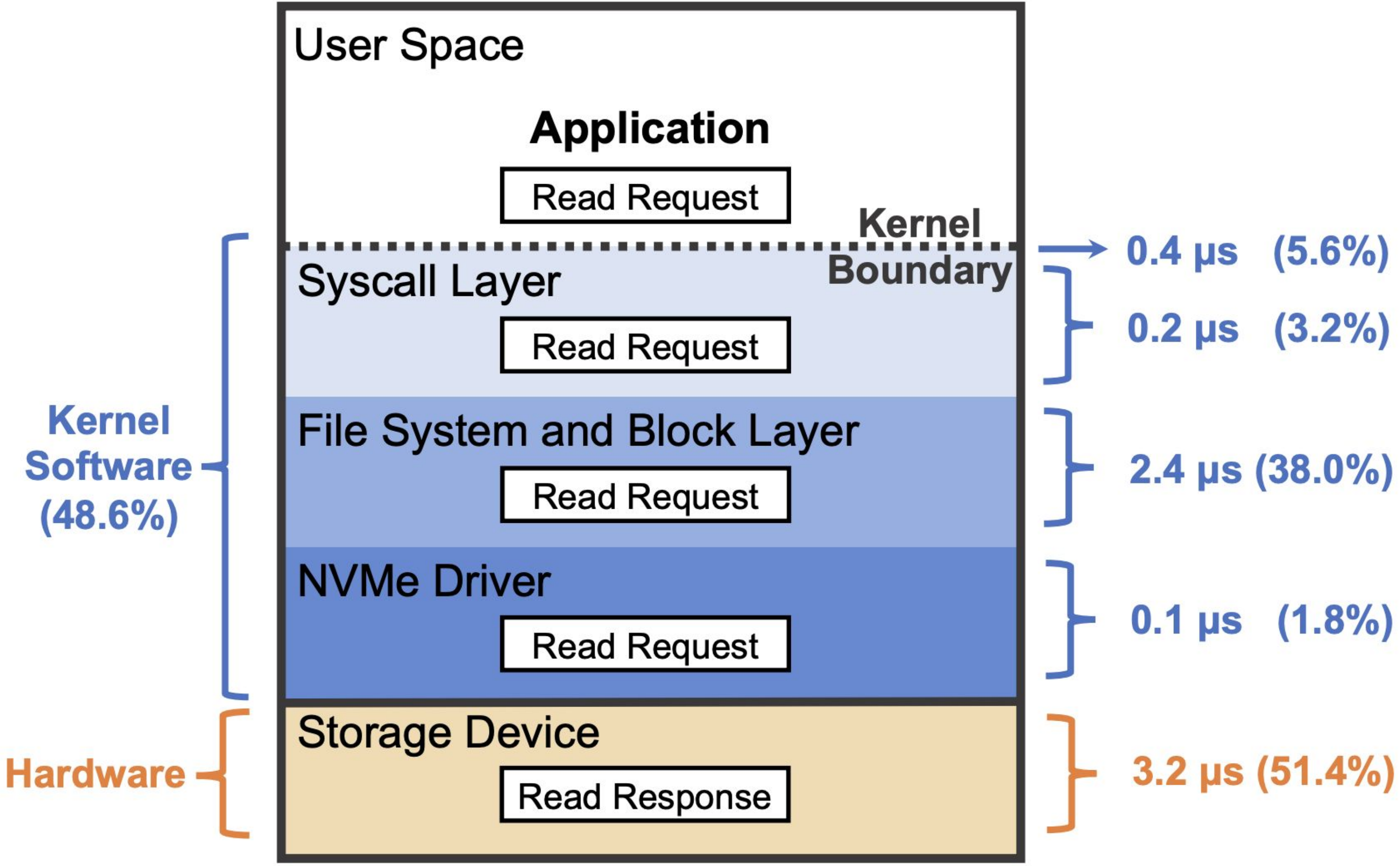
Footer

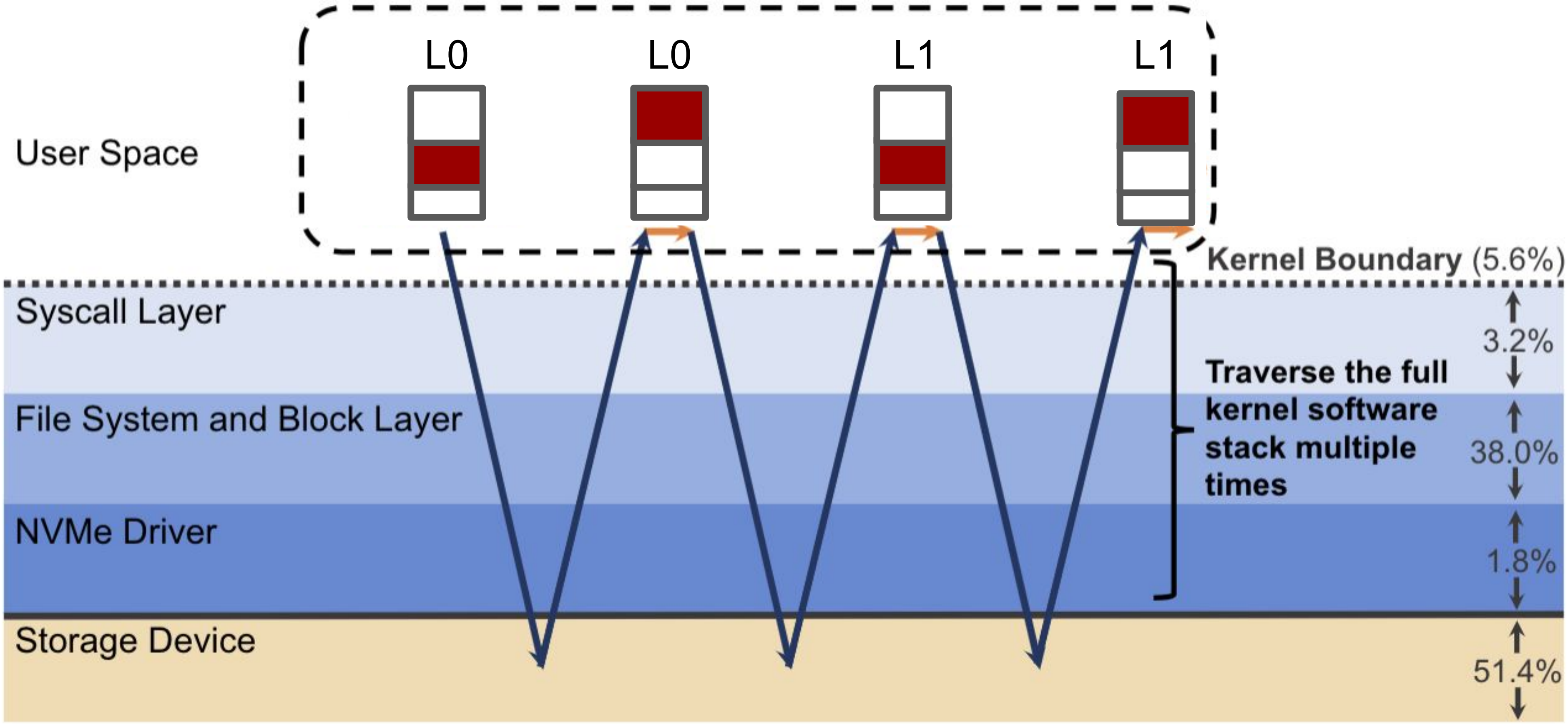
- Stores file metadata
- Usually cached

And more…

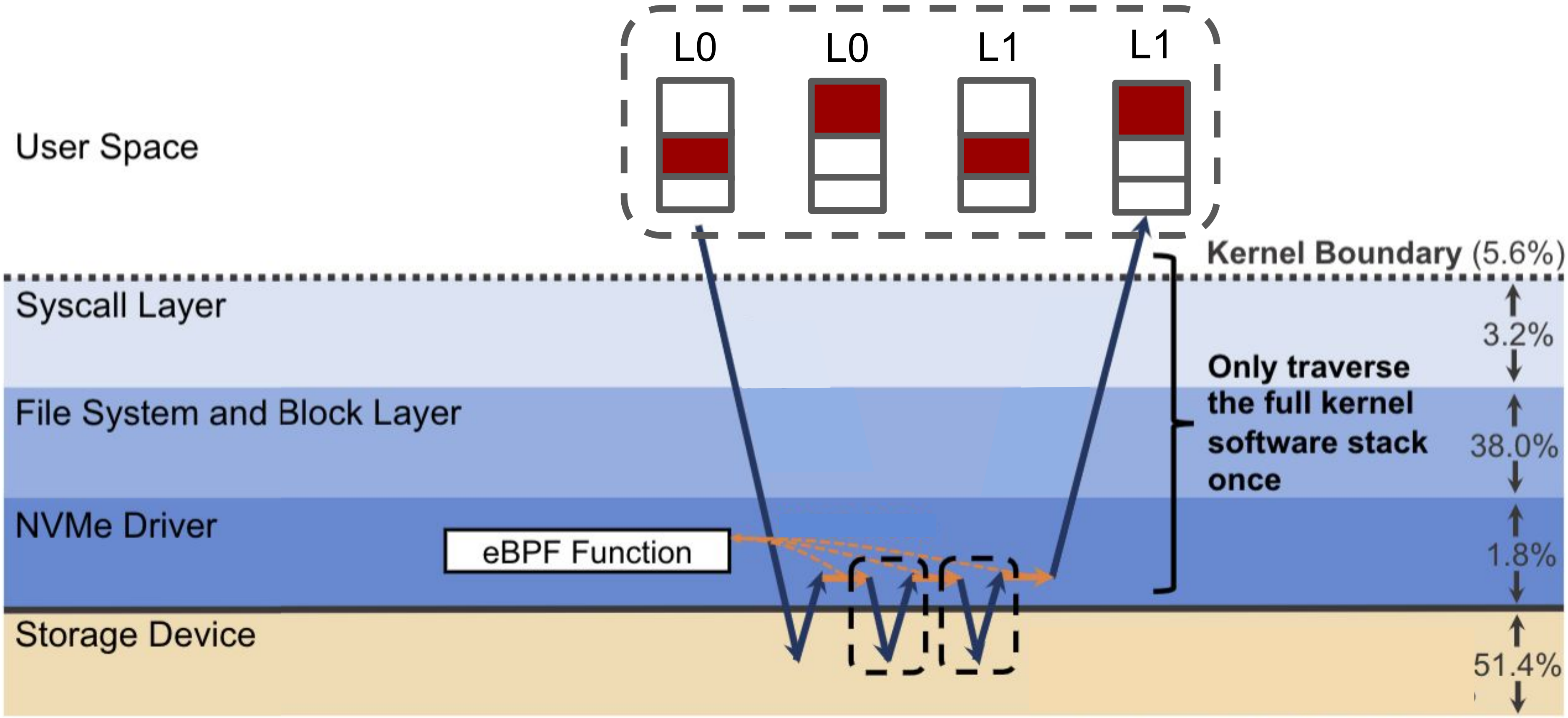
- Metadata blocks
- Bloom filters
- Varint encoding
- Delta encoding
- Atomics
- Lockless concurrency
- Skip lists
- Hash indexing
- … but these are our problems, not yours :')


User Space
Application
Read Request
Kernel
Boundary
Syscall Layer
Read Request
File System and Block Layer
Read Request
NVMe Driver
Read Request
Storage Device
Read Response
Kernel
Software
Hardware


User Space
Application
Read Request
Kernel
Boundary
0.4 µs (5.6%)
Syscall Layer
Read Request
0.2 µs (3.2%)
File System and Block Layer
Read Request
2.4 µs (38.0%)
NVMe Driver
Read Request
0.1 µs (1.8%)
Kernel
Software
(48.6%)
Storage Device
Read Response
Hardware
3.2 µs (51.4%)



L0
L0
L1
L1
User Space
Kernel Boundary (5.6%)
Syscall Layer
3.2%
Traverse the full kernel software stack multiple times
File System and Block Layer
38.0%
NVMe Driver
1.8%
Storage Device
51.4%


L0
L0
L1
L1
User Space
Kernel Boundary (5.6%)
Syscall Layer
3.2%
Only traverse the full kernel software stack once
File System and Block Layer
38.0%
NVMe Driver
eBPF Function
1.8%
Storage Device
51.4%

eBPF

II

eBPF

Background

- (Extended) Berkeley Packet Filter
- 1992: BPF developed for analyzing and filtering network traffic (packets)
- Early 2010s: Reworked in Linux, became eBPF (Linux 3.18)

- **Sandboxed programs in a privileged context (kernel)**

Running an eBPF function

- eBPF bytecode verified when loaded, and JIT-compiled when run
  - Instructions are simulated and change in VM state is observed
  - eBPF bytecode instructions –> modern assembly language instructions
- Use `bpf()` syscall (low-level interface) or `libbpf` (C/C++/Rust)

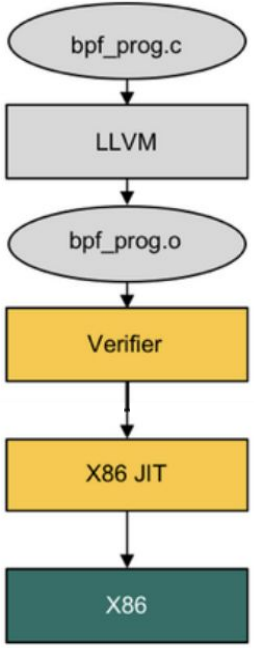


Image source:
https://cdn.open-nfp.org/media/documents/demystify-ebpf-jit-compiler.pdf

eBPF

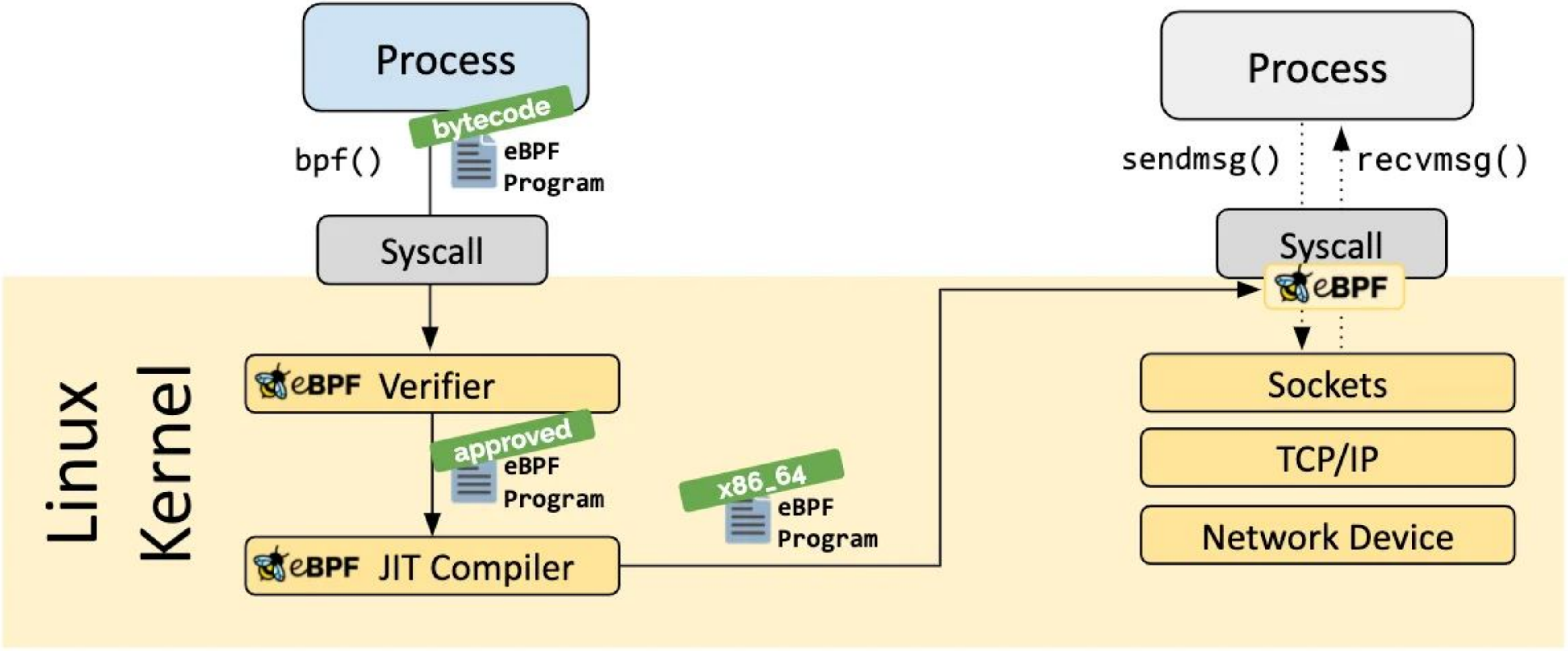


Image Source: https://ebpf.io/what-is-ebpf/

eBPF

Verifier

- Verification ensures:
  - Termination (no infinite loops)
  - Memory safety
  - No kernel crashes (assuming verifier is implemented correctly 💀)
- No false positives! But maybe false negatives…
- Two major stages
  - Control flow stage: DAG check for infinite loops, other CFG checks
  - Data flow stage: Simulates instructions and observes states
    - Pruning and liveness analysis: keep track of safe states to avoid re-simulating

Limitations

- Halting problem is undecidable + want fast verification
  - Instruction and jmp limit
  - Limited set of registers, small stack
- Safety
  - Memory access checks and no dynamic memory
  - Limited set of kernel functionality
- Used to write eBPF bytecode by hand
  - Can now be compiled from C
  - Type safety ensured at compile time

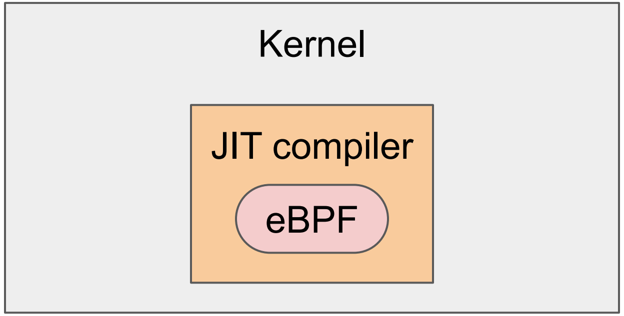
Kernel modules vs. eBPF functions

Kernel module

- Extension of kernel
- Exported symbols
- May break across kernel versions
- Unsafe – can crash kernel
- Requires root permissions (`CAP_SYS_MODULE`)

eBPF function

- In-kernel virtual machine
- Limited access to kernel functions
- Safe – sandboxed and verified
- Event-driven, flexible
- More granular permissions (capabilities)



Verifier Example

Verifier output is…

- Verbose, to say the least
- Not exactly elucidating…
- A work in progress…

XRP

III

XRP

[XRP: In-Kernel Storage Functions with eBPF]

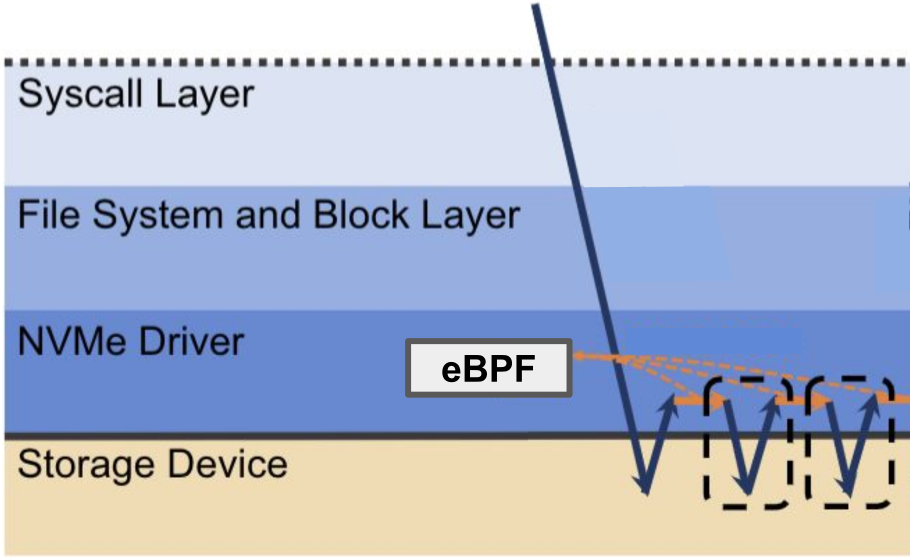
Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon.

Won Best Paper at OSDI 22.

XRP

Goal

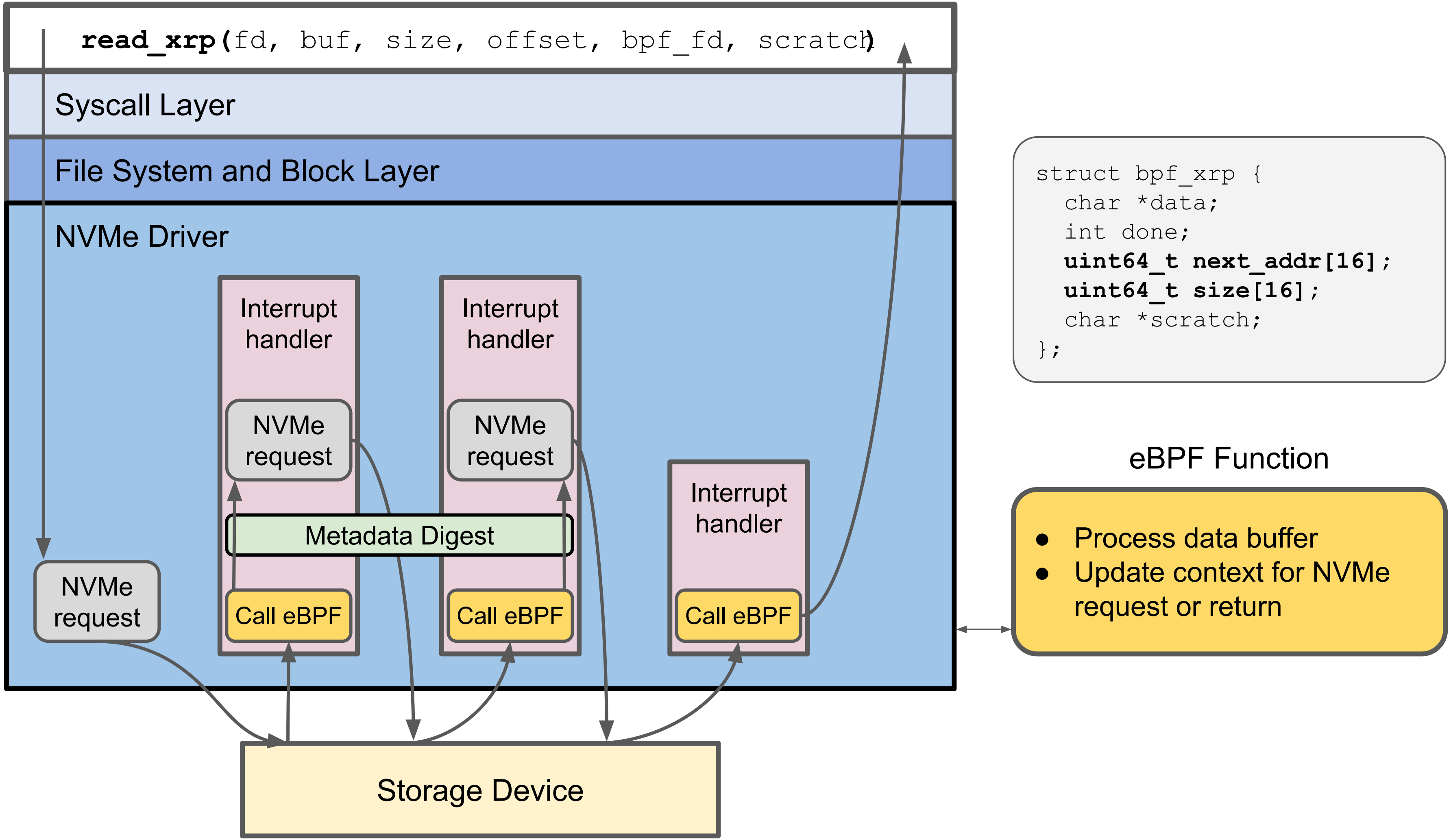
- For a dependent series of reads, do program logic in-kernel



XRP

Challenges

- Files contain logical offsets, NVMe driver only knows physical offsets
  - Need a method to convert
- Statefulness
  - Sequence of dependent reads
  - eBPF programs are usually stateless
- And much more…

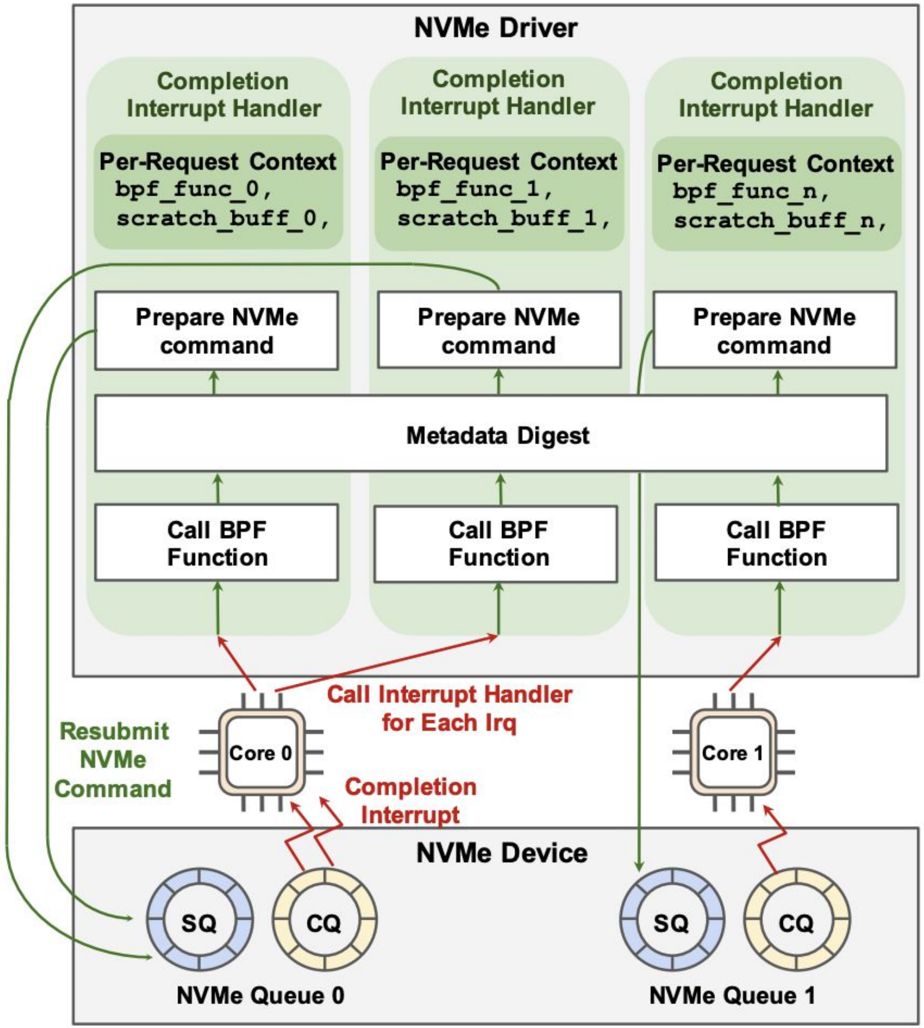


read_xrp(fd, buf, size, offset, bpf_fd, scratch)
Syscall Layer
File System and Block Layer
NVMe Driver
Interrupt handler
NVMe request
Interrupt handler
NVMe request
Interrupt handler
Metadata Digest
NVMe request
Call eBPF
Call eBPF
Call eBPF
Storage Device
struct bpf_xrp {
char *data;
int done;
uint64_t next_addr[16];
uint64_t size[16];
char *scratch;
};
eBPF Function
Process data buffer
Update context for NVMe request or return

IV

NVMe and Interrupts

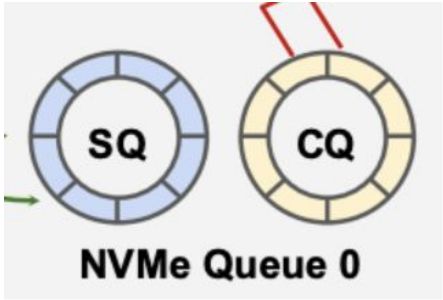
- NVMe: Non-Volatile Memory Express
  - Standard for accessing non-volatile storage (usually NAND flash drives)
  - Can access drives over PCIe (local) or through TCP or RDMA (network – NVMe-oF)
  - Created in 2011
- PCIe: Peripheral Component Interconnect Express
  - Standard for interface between motherboard and expansion cards (SSDs, GPUs, etc.)
  - Created in 2003


NVMe Driver
Completion Interrupt Handler
Per-Request Context bpf_func_0, scratch_buff_0,
Completion Interrupt Handler
Per-Request Context bpf_func_1, scratch_buff_1,
Completion Interrupt Handler
Per-Request Context bpf_func_n, scratch_buff_n,
Prepare NVMe command
Prepare NVMe command
Prepare NVMe command
Metadata Digest
Call BPF Function
Call BPF Function
Call BPF Function
Call Interrupt Handler for Each Irq
Resubmit NVMe Command
Core 0
Core 1
Completion Interrupt
NVMe Device
SQ
CQ
SQ
CQ
NVMe Queue 0
NVMe Queue 1

NVMe and Interrupts

Request Submission and Completion

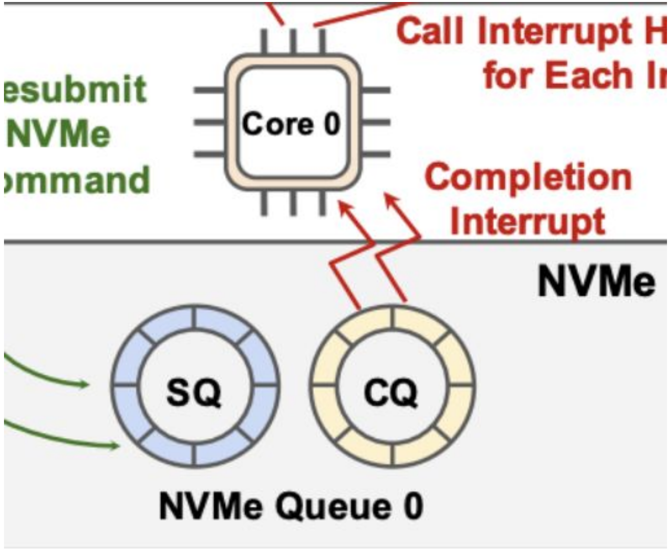
- Paired submission and completion queue (ring buffers)
- Driver:
  - Adds request to SQ and updates SQ tail
  - Rings hardware SQ doorbell (new tail)
  - Waits



NVMe and Interrupts

Request Submission and Completion

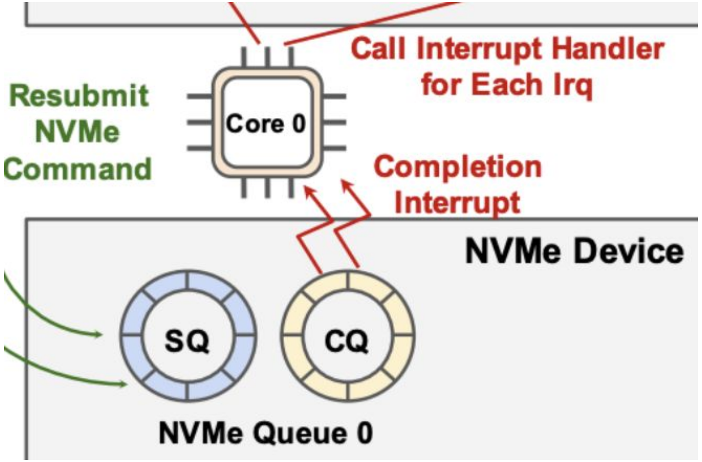
- Paired submission and completion queue (ring buffers)
- Hardware (controller chip):
    - Takes a request from SQ and updates SQ head
    - Handles request
    - Adds completed request to CQ and updates CQ tail
    - Generates interrupt
    - Repeat



NVMe and Interrupts

Request Submission and Completion

- Paired submission and completion queue (ring buffers)
- Driver (again):
  - Wakes up
  - Takes request from CQ and updates CQ head
  - Rings hardware CQ doorbell (new head)
  - Handles interrupt



NVMe and Interrupts

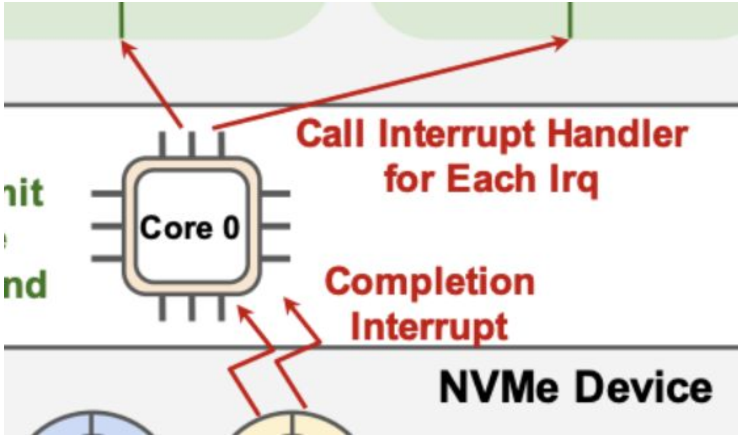
Request Submission and Completion

- Process that issued the I/O is sleeping, waiting for completion
  - Wait queue!
- **I/O in the kernel is inherently asynchronous**

NVMe and Interrupts

Interrupts

- Raised by hardware, handled by software
- For NVMe, indicates completed request - needs to be handled



NVMe and Interrupts

Interrupts

- In Linux, split into upper-half and lower-half
- Upper-half:
  - Preempts currently running process – runs in its execution context with higher privilege level
  - How does this affect scheduling timeslices? XRP Section 4.3
  - Keep it short and simple – avoid reentrancy and concurrency issues
  - **Do the absolutely necessary, urgent work, then “wake up” lower-half**

NVMe and Interrupts

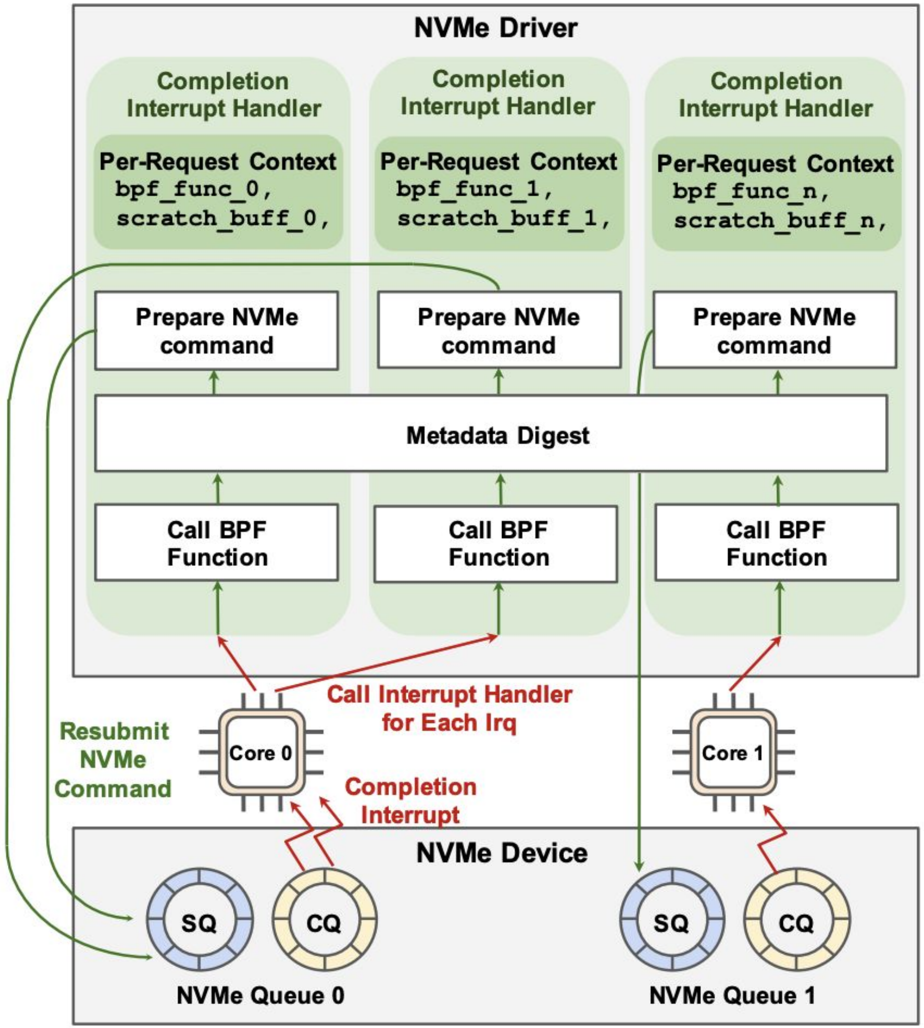
Interrupts

- In Linux, split into upper-half and lower-half
- Lower-half:
  - Lower-half “woken up” by front-half handler
  - Does the majority of the work
  - **Put eBPF function call in the lower-half interrupt handler**

NVMe and Interrupts

Interrupts

- In Linux, split into upper-half and lower-half
- NVMe example:
  - `pci_request_irq()`
  - `nvme_irq()`


NVMe Driver
Completion Interrupt Handler
Per-Request Context
bpf_func_0,
scratch_buff_0,
Completion Interrupt Handler
Per-Request Context
bpf_func_1,
scratch_buff_1,
Completion Interrupt Handler
Per-Request Context
bpf_func_n,
scratch_buff_n,
Prepare NVMe command
Prepare NVMe command
Prepare NVMe command
Metadata Digest
Call BPF Function
Call BPF Function
Call BPF Function
Call Interrupt Handler for Each Irq
Resubmit NVMe Command
Core 0
Core 1
Completion Interrupt
NVMe Device
SQ
CQ
SQ
CQ
NVMe Queue 0
NVMe Queue 1

Now, let’s get started!

Integration

- Kernel modifications
  - Index blocks can be big – how do we store one contiguously in memory?
  - Multi-file support for XRP
- eBPF implementation
  - Reimplement SST file parsing in eBPF
- Userspace modifications
  - Modify RocksDB to call `read_xrp()`
  - Set up multi-file read
  - RocksDB has a cache – how do we use it when we’re in the kernel?

Huge Pages

V

Huge Pages

What is a huge page?

- Larger-than-default page size
- Common sizes: 2MB and 1GB
- Supported by most modern operating systems
- Configurable by users and applications

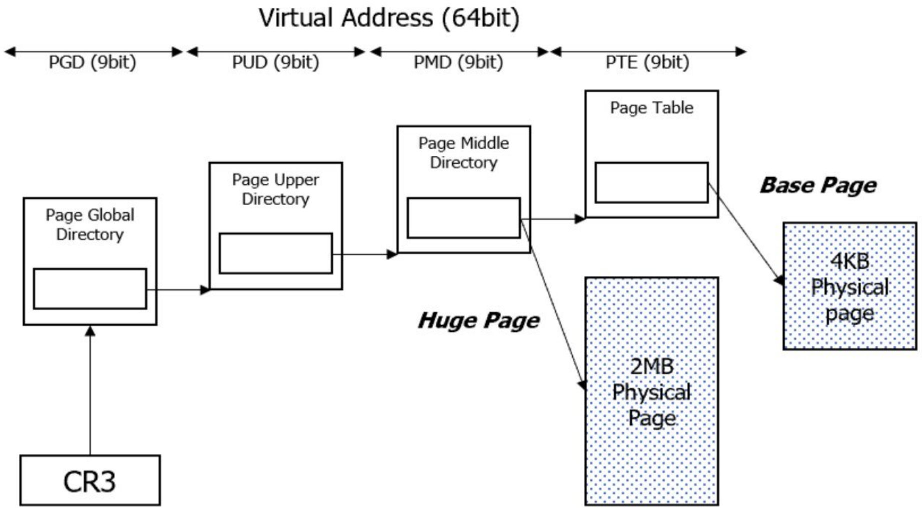


Image Source: https://dl.acm.org/doi/10.1145/3297280.3297425

Huge Pages

TLB: 1,500-2,000 entries per CPU core (typically)

With 4KB pages:

2,000 entries * 4 KB
= 8 MB

With 2MB pages:

2,000 entries * 2 MB
= 4 GB

Page table entries: 512x smaller for same working set

- Reduced memory usage for page tables
- Easier caching of page tables

Huge Pages

Pros

- Reduces translation lookaside buffer (TLB) overhead
- Improves performance for large memory workloads
- Decreased page table management overhead
- Enhanced cache locality

Cons

- Increased memory waste for small workloads
- Limited availability of large contiguous memory regions
- Fragmentation
- Kernel code complexity (compound pages)

Huge Pages

Acquisition

- Using `hugetlbfs`:
    - Pre-allocate huge pages with pseudo-filesystem
    - `mmap()` with `MAP_HUGETLB` flag and huge page size
- Using transparent huge pages:
    - Kernel dynamically allocates huge pages, if available
    - `posix_memalign()` allocates aligned memory
    - `madvise()` with `MADV_HUGEPAGE` flag on allocated, but unused memory

Interlude

VI

Interlude

Everything is broken

- Huge pages broke everything
- Our code did not work
- Kernel corruption, memory corruption, file system corruption
- Heretofore unseen dmesg output

Interlude

Two bugs

1. Can’t read more than 4096 bytes on resubmission
2. Reading garbage if we try to read the whole file initially

Block Layer

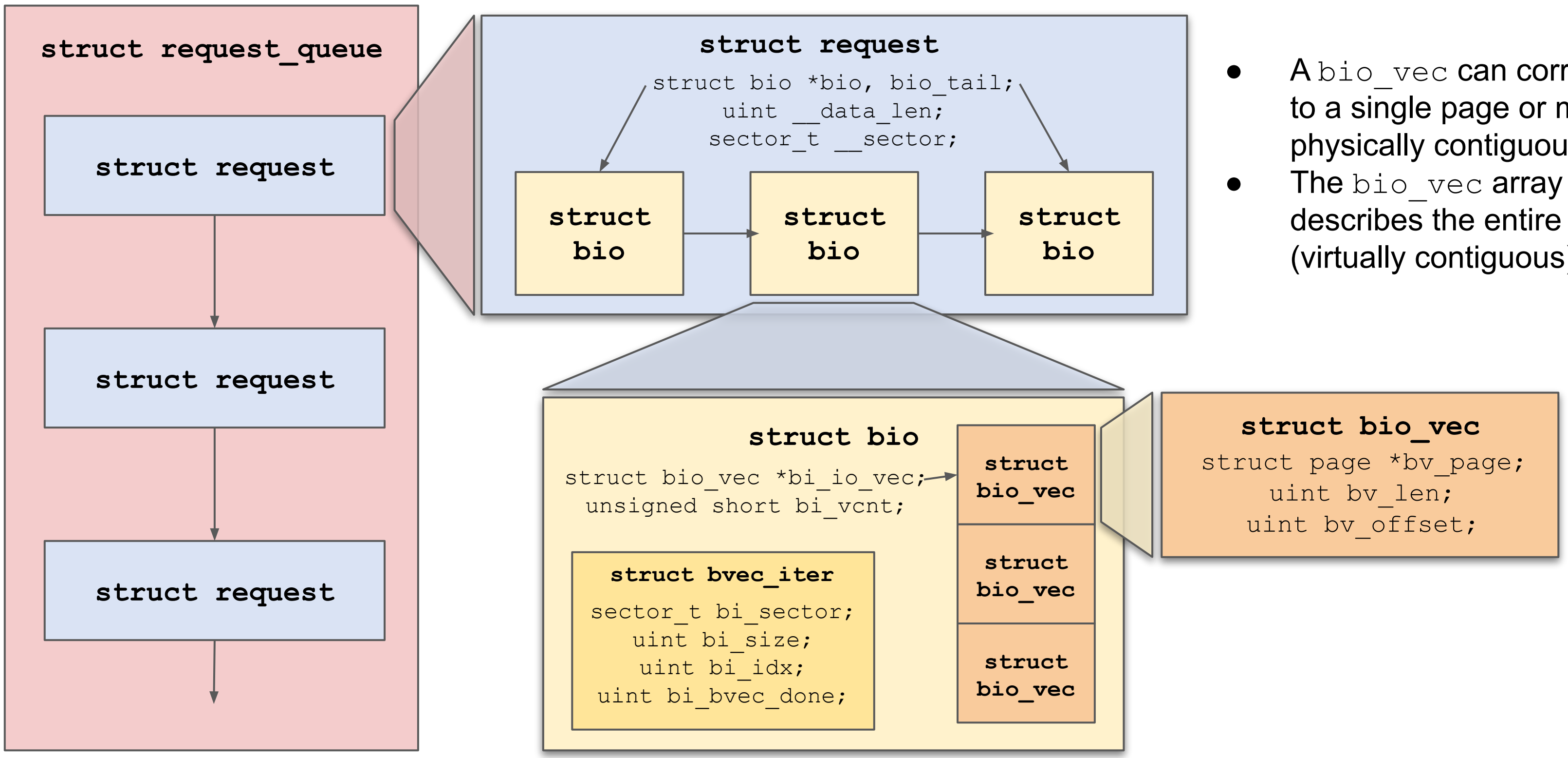
VII

Block Layer

What is it?

- Interface between filesystem and block device drivers
- All the code in Linux `block` subdirectory
- Two sub-layers: bio layer and request layer
    - bio layer: manipulate I/O requests, pass them to request layer
        - Thin layer, doesn't do much
    - Request layer: **schedule I/O requests** and pass them to driver

Block Layer



- A `bio_vec` can correspond to a single page or multiple physically contiguous pages
- The `bio_vec` array describes the entire buffer (virtually contiguous)

Block Layer

File System → Block Layer

- `submit_bio()`: Pass a created bio to the block layer
- `bio` vs. `buffer_head`
  - `buffer_head` is the original interface
  - `bio` represents an I/O operation, `buffer_head` represents a single buffer
  - `bio` is more lightweight, flexible - can represent multiple pages + disk blocks

Block Layer

But why does the driver care about this?

- Driver is below the block layer - why do we care about the internals of `struct request` beyond the disk offset and the size?

DMA Mapping

VIII

DMA Mapping

DMA: Direct Memory Access

- Allows devices to write data to memory without going through the CPU
- When a process wants to read:
    - Driver allocates a DMA buffer, tells hardware to write data there
    - Process sleeps
    - Hardware writes data to buffer, generates interrupt

DMA Mapping

NVMe DMA Mapping

- Our problem: **Can’t read more than 4096 bytes on resubmission**
- The NVMe driver uses the `struct bio` to figure out DMA mapping size
- When we resubmit, we use the original DMA buffer
  - If we resubmit with a larger size, the DMA buffer is too small
  - Need to allocate a new DMA buffer with new size
- *On resubmission, structs need to be updated to reflect new size*

DMA Mapping

NVMe DMA Mapping

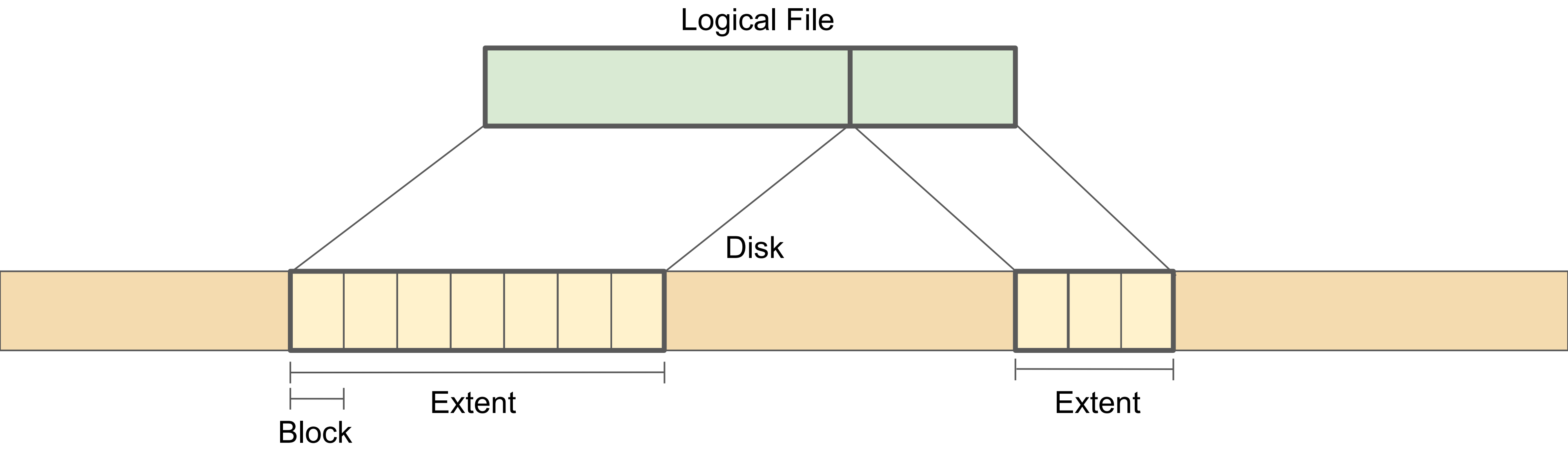
- Still seeing a kernel `BUG()` ...
- Kernel seems to expect consistency between original request and completed request…
- [`blk_mq_end_request()`]
- Save original values and reset when done

Extents

IX

Extents

- ext4 is an extent-based file-system
- File blocks not necessarily contiguous on disk



Extents

- NVMe driver can only read contiguous blocks in one request – extents at file system layer
- Why might we be seeing garbage?

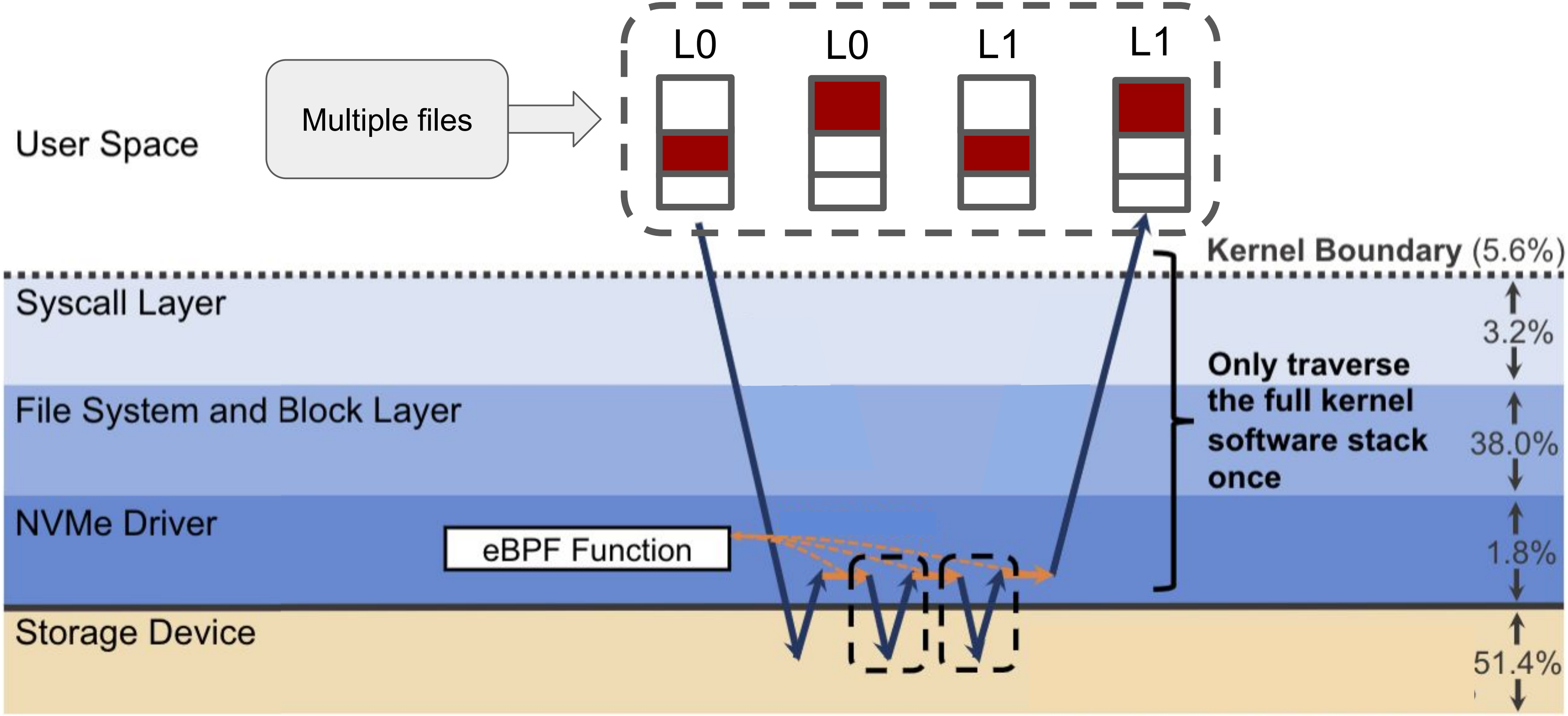
Extents

- Reads past extent boundary – garbage
- Need to update metadata digest query size
- If metadata digest tells us we’d be crossing an extent boundary, just give up

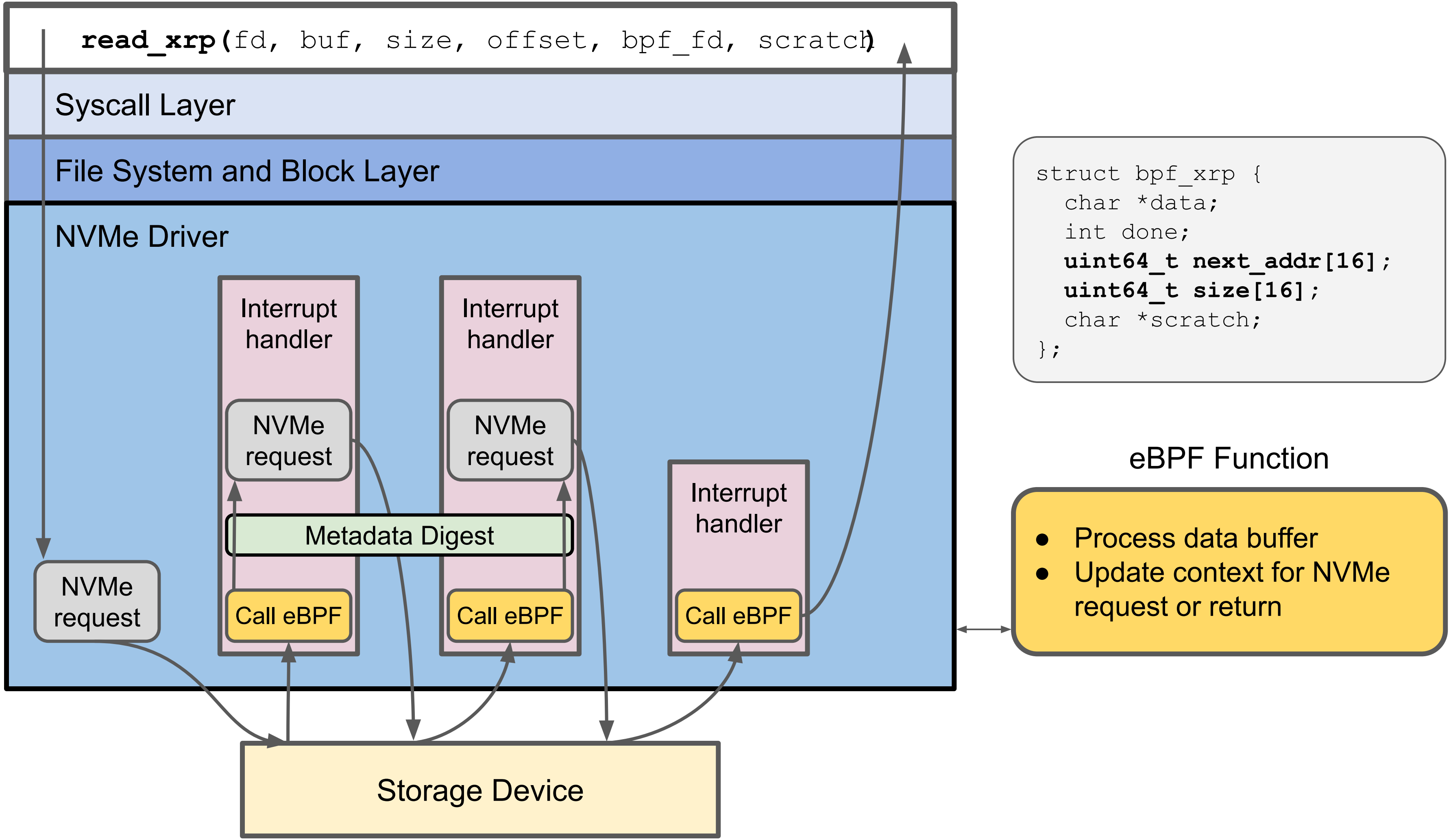
Multi-file support

X

To review…



User Space
Multiple files
L0
L0
L1
L1
Kernel Boundary (5.6%)
Syscall Layer
3.2%
Only traverse the full kernel software stack once
File System and Block Layer
38.0%
NVMe Driver
eBPF Function
1.8%
Storage Device
51.4%


read_xrp(fd, buf, size, offset, bpf_fd, scratch)
Syscall Layer
File System and Block Layer
NVMe Driver
NVMe request
Interrupt handler
NVMe request
Call eBPF
Interrupt handler
NVMe request
Call eBPF
Metadata Digest
Interrupt handler
Call eBPF
Storage Device
struct bpf_xrp {
char *data;
int done;
uint64_t next_addr[16];
uint64_t size[16];
char *scratch;
};
eBPF Function
Process data buffer
Update context for NVMe request or return

Multi-file support

Simplifying assumptions

- The eBPF program has an array of file descriptors
- When updating context for NVMe request, specify new `fd` in `struct bpf_xrp`

```
struct bpf_xrp {
  char *data;
  int done;
  uint64_t next_addr[16];
  uint64_t size[16];
  char *scratch;
};
```

→

```
struct bpf_xrp {
  char *data;
  int done;
  uint64_t next_addr[16];
  uint64_t size[16];
  uint64_t fd[16];
  char *scratch;
};
```

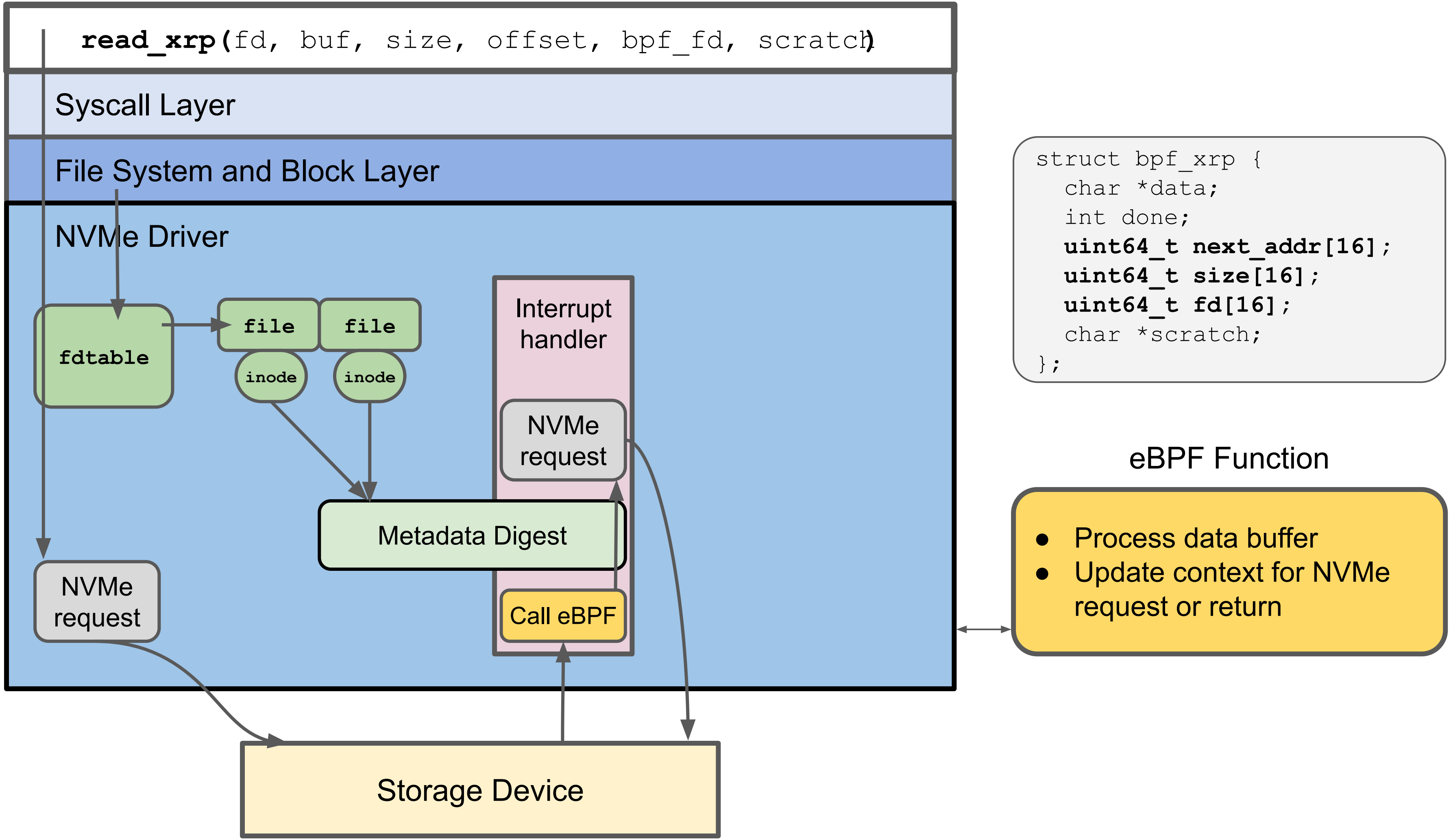
Multi-file support

Problem

- Metadata digest: uses inodes to translate logical to physical address
- Need to convert file descriptors to inodes

Solution

- Cache file descriptor table
- XRP runs in interrupt context – need to use cached version



read_xrp(fd, buf, size, offset, bpf_fd, scratch)
Syscall Layer
File System and Block Layer
NVMe Driver
fdtable
file
file
inode
inode
Interrupt handler
NVMe request
Metadata Digest
Call eBPF
NVMe request
Storage Device
struct bpf_xrp {
char *data;
int done;
uint64_t next_addr[16];
uint64_t size[16];
uint64_t fd[16];
char *scratch;
};
eBPF Function
Process data buffer
Update context for NVMe request or return

User space and eBPF

XI

eBPF

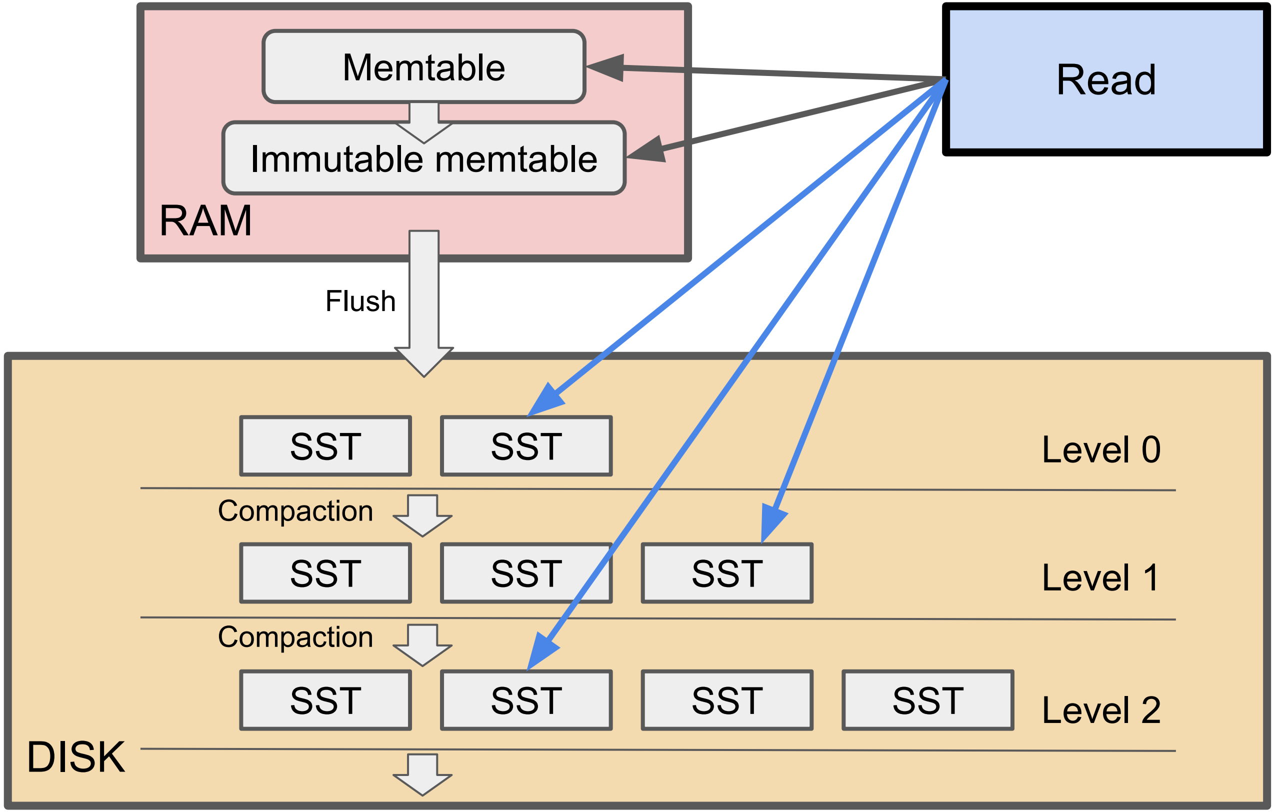
Implementation

- Replicate RocksDB logic to parse SST file in eBPF
- Limitations of eBPF code require
  - Function-by-function verification
  - Maximum bounds on loops – iterating over index block or data block
  - Reimplementation of simple functions like `strcmp()`
  - Pre-allocate potential dynamic memory buffers
  - And much more…

User space

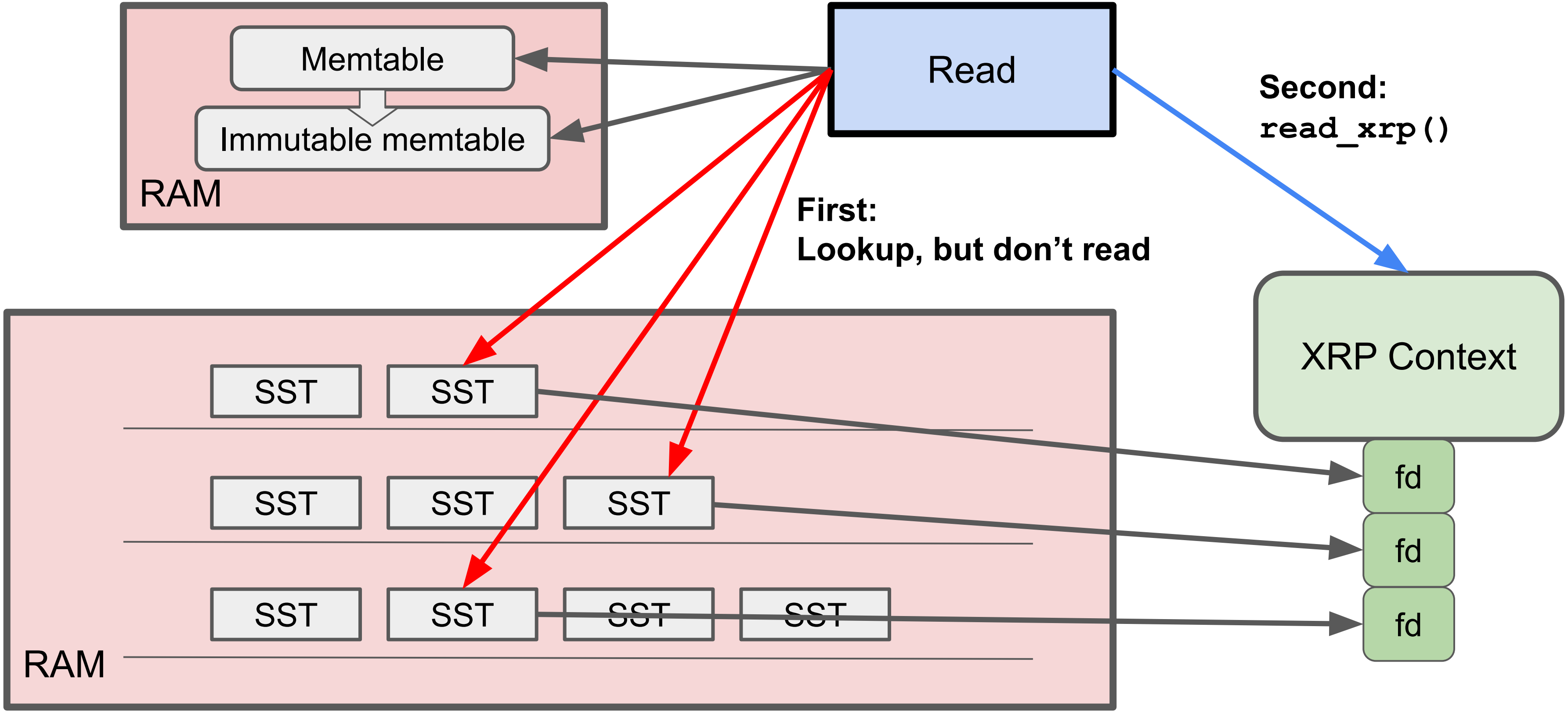
Implementation

- Earlier assumption – eBPF program is given array of file descriptors
- How do we build the array?



Memtable
Immutable memtable
RAM
Read
Flush
SST
SST
Level 0
Compaction
SST
SST
SST
Level 1
Compaction
SST
SST
SST
SST
Level 2
DISK

XRP support

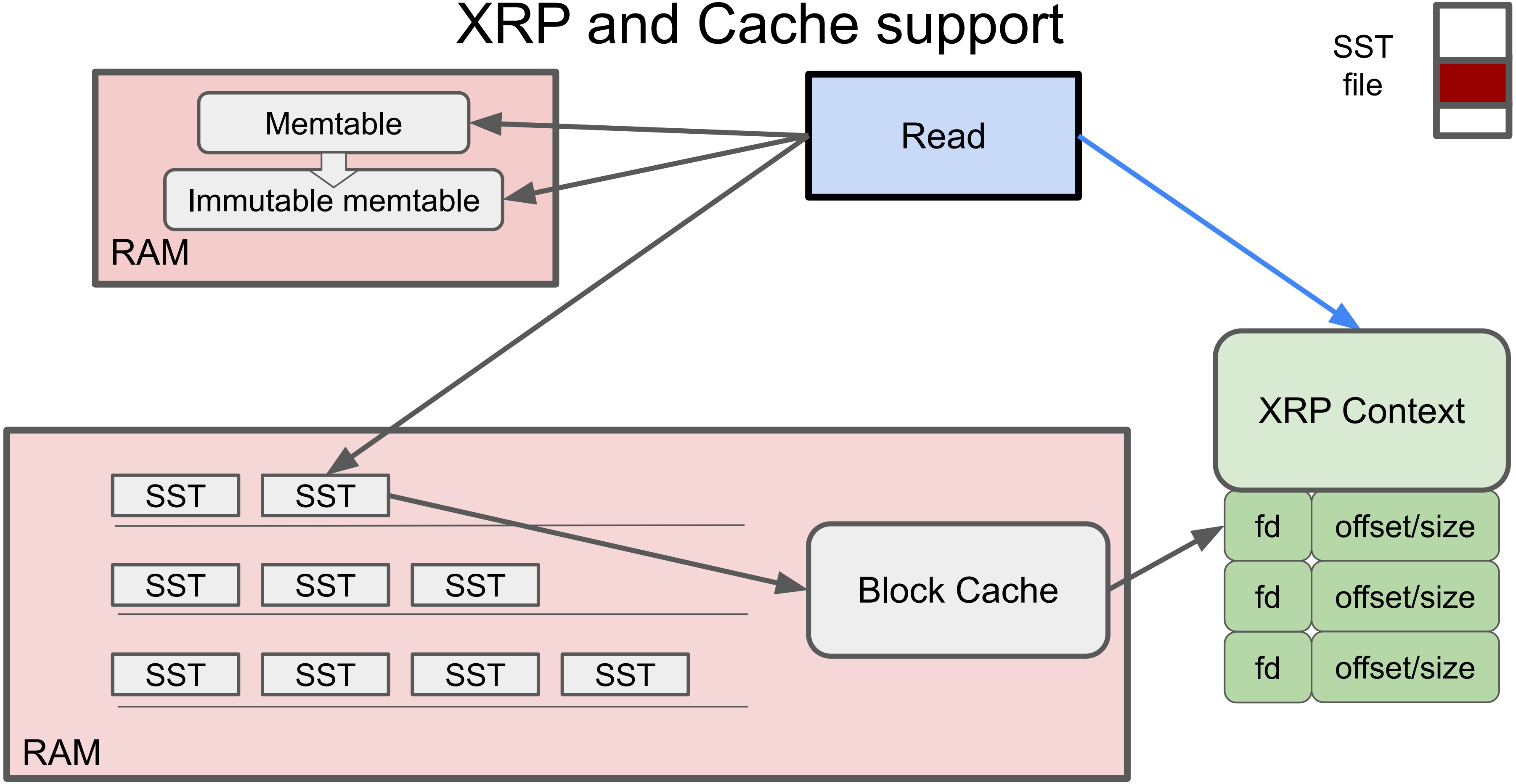

Memtable
Immutable memtable
RAM
Read
Second:
read_xrp()
First:
Lookup, but don't read
XRP Context
SST
SST
SST
SST
SST
SST
SST
SST
SST
fd
fd
fd
RAM

User space

Block Cache

- RocksDB has an in-memory cache for index and data blocks
- How can we use it?
- We can’t read the cache contents from eBPF

XRP and Cache support

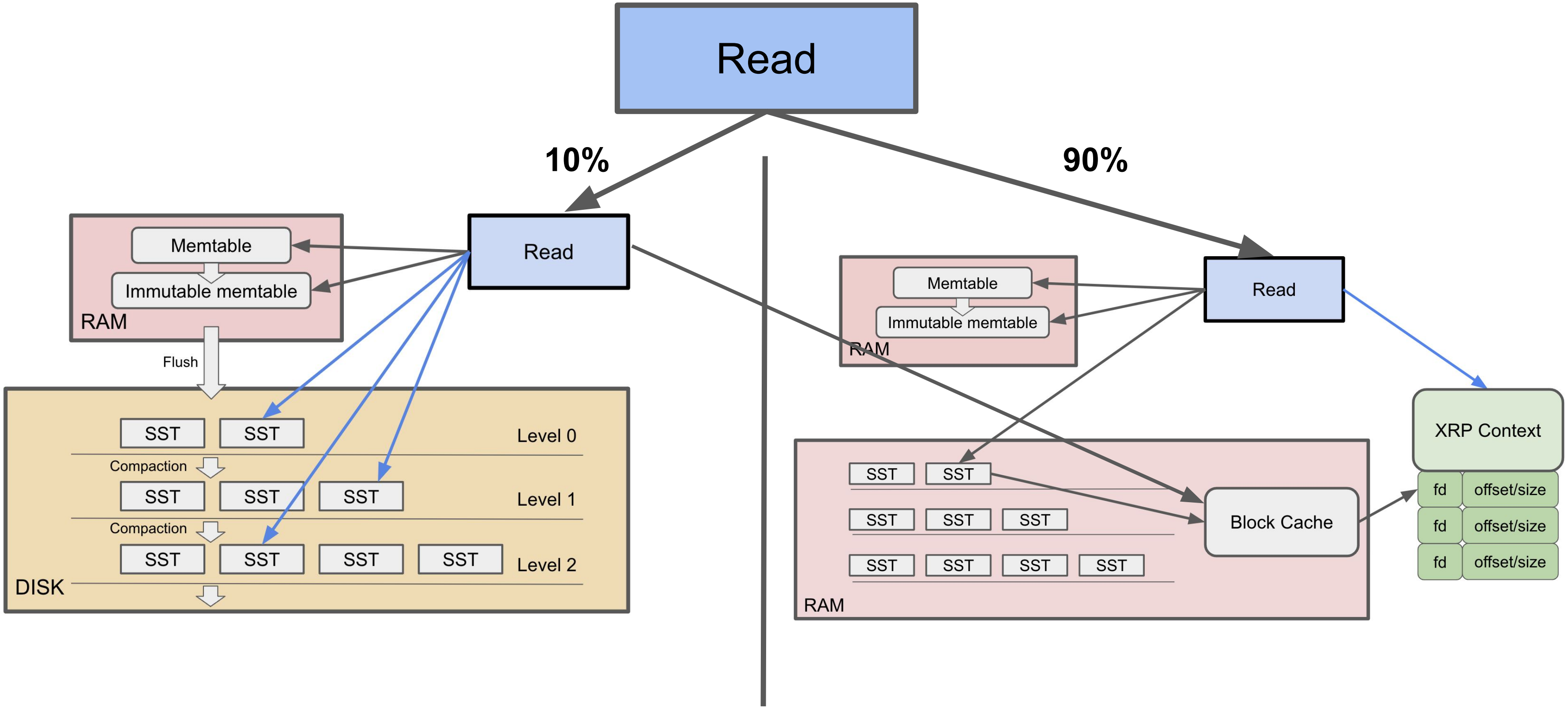


User space

Block Cache

- There's a problem: the block cache gets populated when RocksDB reads
- If we only use XRP, we have to
  - Transfer the blocks we read in-kernel back to RocksDB (hard!)
  - or, do something simpler:
  - Sampling – make a small percentage of reads using regular RocksDB

XRP and Cache support



Putting it all together

RocksDB makes a read

- In user space:
  - Collect array of file descriptors + offsets for XRP to use
  - Call `read_xrp()`, passing in eBPF context + metadata
- In kernel:
  - Prepare NVMe request
  - Call eBPF program, resubmit
- In eBPF:
  - Parse data buffer
  - Prepare next NVMe request (size, offset, fd), or return

Flame Graphs

- Uses Perf profiler to capture how long program spends inside functions
- Similar to ftrace, better for timing and visualization

Vanilla RocksDB

XRP RocksDB

Left Unsaid

There's a lot we don't have time to cover:

- Thread-local memory
- Direct I/O (I/O that skips the page cache)
- Page cache behavior
- I/O scheduling
- IOMMUs
- NVMe-oF
- XRP vs kernel bypass (SPDK)

Connections

HWs

- HW1 (linux-list): Modules vs. eBPF
- HW3 (multi-server): mmap(), thread-local storage, huge pages
- HW4 (tabletop): File descriptor table
- HW5 (fridge): Key-value stores, wait queues
- HW6 (freezer): Interrupts and scheduling
- HW7 (farfetchd): Huge pages, DMA mapping
- HW8 (pantry): Extents, block layer, page cache

Connections

Classes

- COMS 4111 Databases: LSM Tree
- COMS 4115 Programming Languages & Translators: eBPF JIT and verifier
- CSEE 4119 Computer Networks: eBPF use cases
- Some Computer Engineering class (probably): DMA, interrupts
- EECS E6897 Cloud Data Infrastructure: everything!

Acknowledgements

Ioannis Zarkadas, Sheng Jiang, and Yuhong Zhong, *Columbia University;* Md Ashfaqur Rahaman, *University of Utah;* Hubertus Franke and Jonas Pfefferle, *IBM;* Junfeng Yang, *Columbia University;* Kostis Kaffes and Amy Tai, *Google*; Ryan Stutsman, *University of Utah;* Asaf Cidon, *Columbia University*

Asaf is always looking for OS students to conduct research in his lab!

asaf.cidon@columbia.edu