Efficient User-level Physical Memory Management on seL4

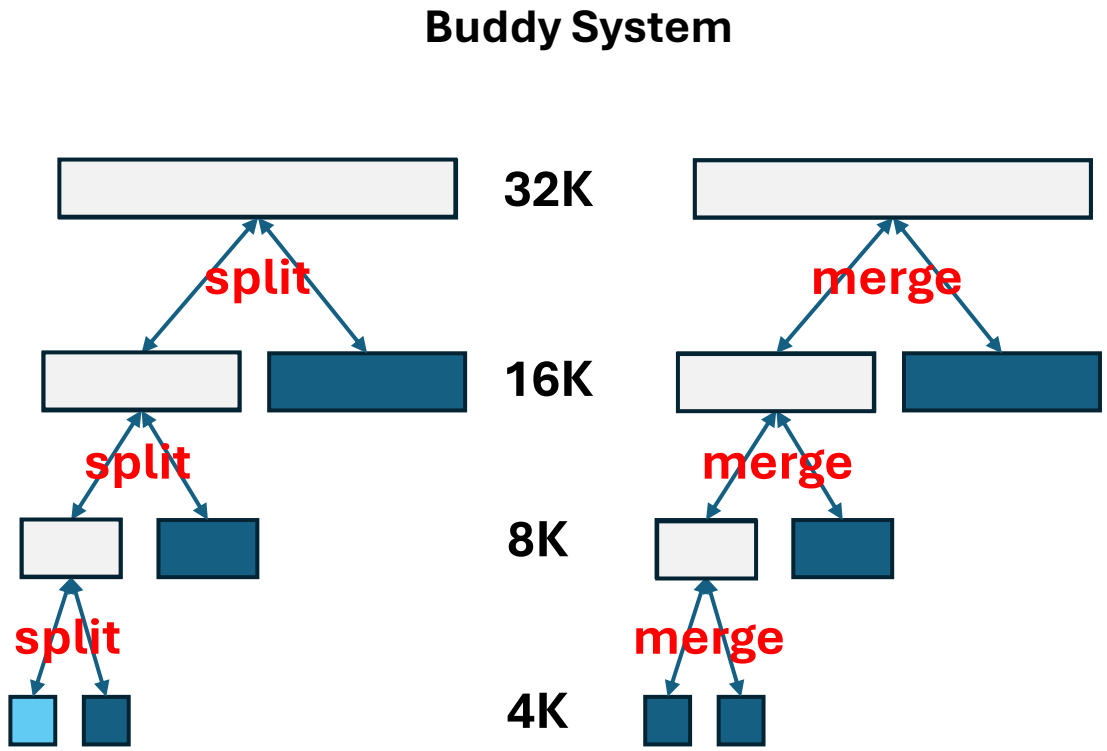
Guangtao Zhu[1,2], Zhuang Lu[1], Qianying Zhang[3],
*Shijun Zhao[1], and Rui Hou[1]

[1]Institute of Information Engineering, Chinese Academy of Sciences
[2]Trustworthy Systems Group, UNSW Sydney
[3]Capital Normal University

中国科学院信息工程研究所
INSTITUTE OF INFORMATION ENGINEERING,CAS

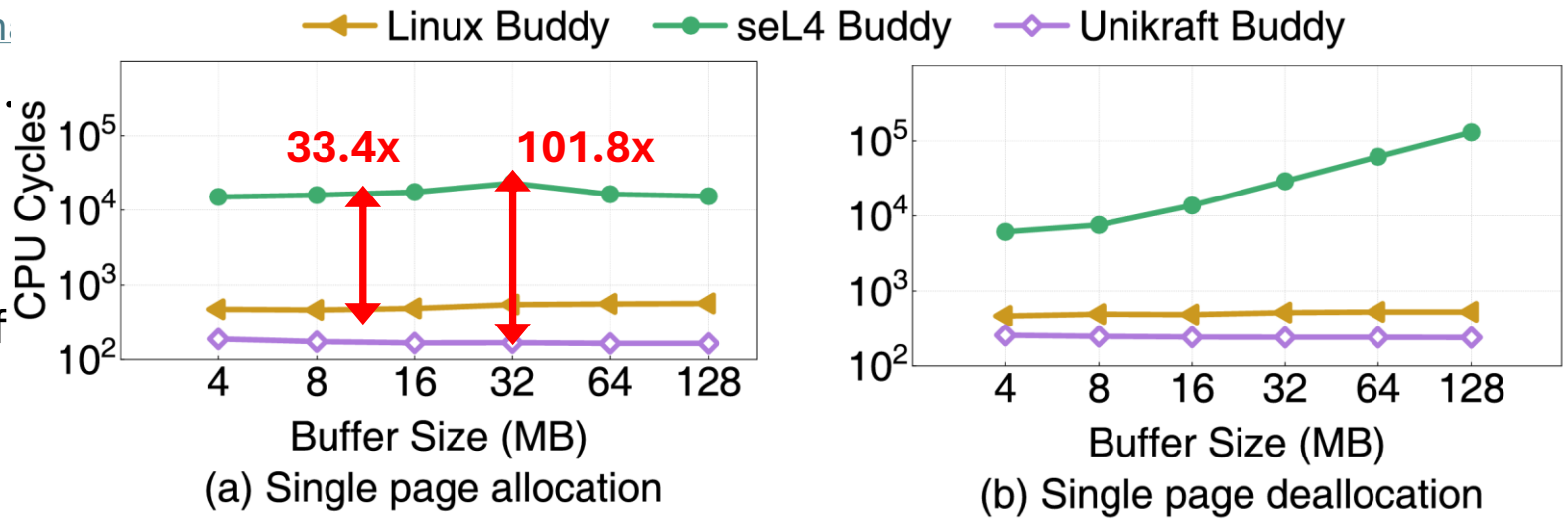
Capability-based resource management on seL4

- seL4 – securely multiplexing hardware + capability-protected kernel abstractions
- Physical memory
  - Untyped (memory)
  - Pagetable (mappable memory)
  - Frame (mappable memory)
- Policies for resource management are delegated to the user-level LibraryOS.



Testbed Setup

- Platform
  - Raspberry Pi 3B+ (64-bit) platform.
  - A quad core Cortex-A53 processor (fixed at 1.4 GHz) and 1 GB of DRAM.
- Baselines
  - seL4_Libs' buddy
    - LibOS architecture
    - seL4_libs/libsel4allocman at m
  - Linux buddy (kernel version 6.
  - Unikraft (LibOS) buddy
- Target
  - Average cycles of allocating/f



Breakdown Analysis

- Design
  - Both baselines have similar policies (split, merge, recursively), which are cheap
  - Both baselines build policies on the abstraction of memory (e.g., seL4 – *Untyped*, Linux – *struct page*[1])

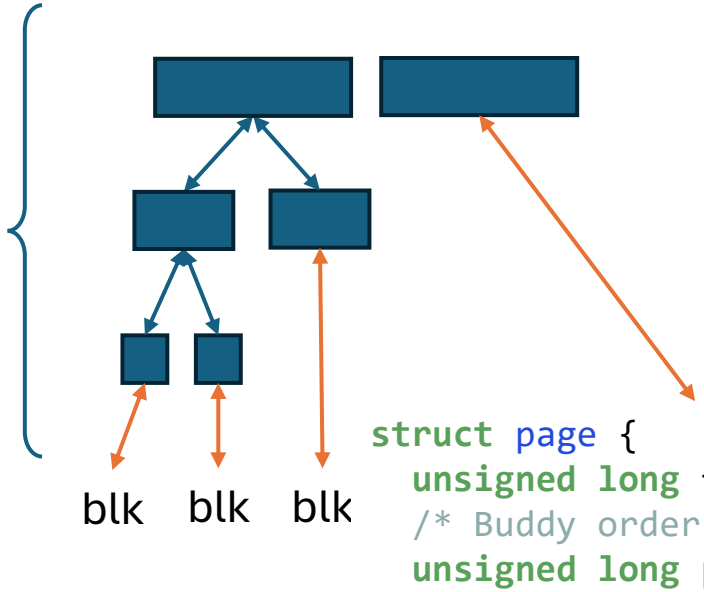
1. **Update block metadata according to split/merge operations.**

Each split takes 137 cycles on Linux

(4KB out of 4MB)

Enforcing buddy system's policy is not expensive...

Buddy system algorithm/ policy



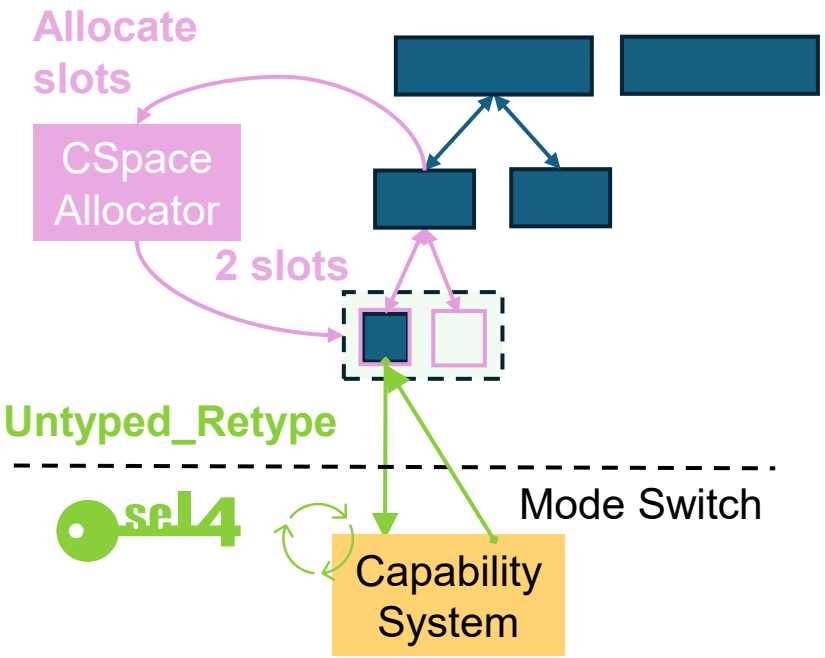

```
struct page {
  unsigned long flags;
  /* Buddy order (2^private) */
  unsigned long private;
  ...
  struct list_head buddy_list;
};
```

[1] https://github.com/torvalds/linux/blob/master/include/linux/mm_types.h

Breakdown Analysis

- Design
  - Both baselines have similar policies (split, merge, recursively), which are cheap
  - Both baselines build policies on the abstraction of memory (e.g., seL4 – *Untyped*, Linux – *struct page*[1])
  - **Difference – the call chain to create/reclaim an abstraction on seL4 Buddy is longer**

1. Update block metadata according to split/merge operations.
2. **User-level capability bookkeeping** **+ 568 cycles (user-level)**
3. **Kernel invocations for memory abstractions (Untyped) provisioning** **+ 1,396 cycles (kernel)**



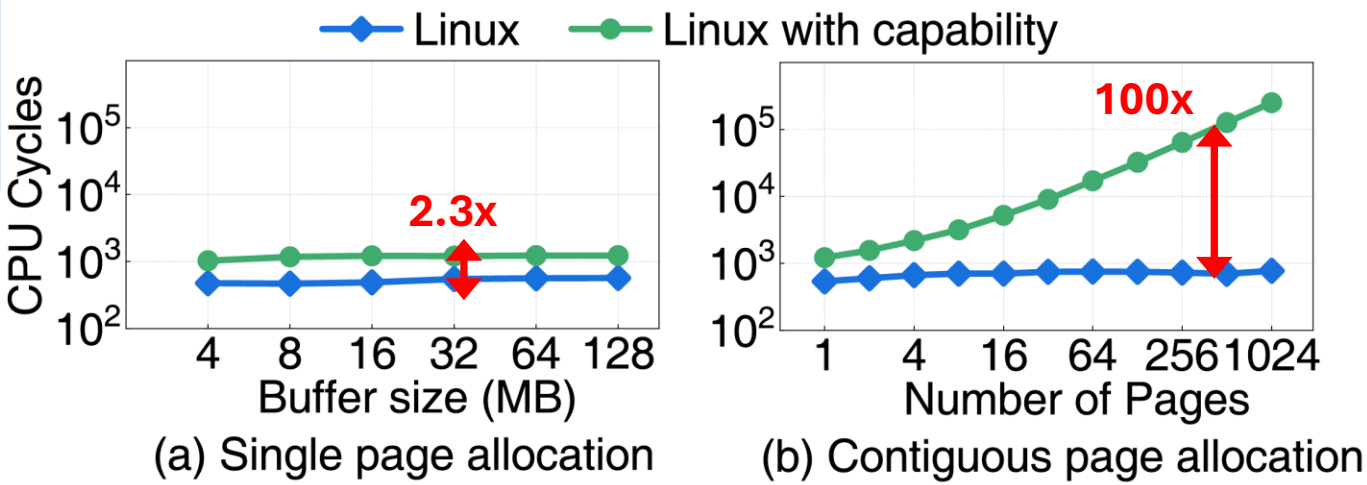
Breakdown analysis

- Design
  - Both baselines have similar policies (split, merge, recursively), which are cheap
  - Both baselines build policies on the abstraction of memory (e.g., seL4 – *Untyped*, Linux – *struct page*[1])
  - **Difference – the call chain to create/reclaim an abstraction on seL4 Buddy is longer**

1. Update block metadata according to split/merge operations.
2. **User-level capability bookkeeping**
3. **Kernel invocations for memory abstractions provisioning**

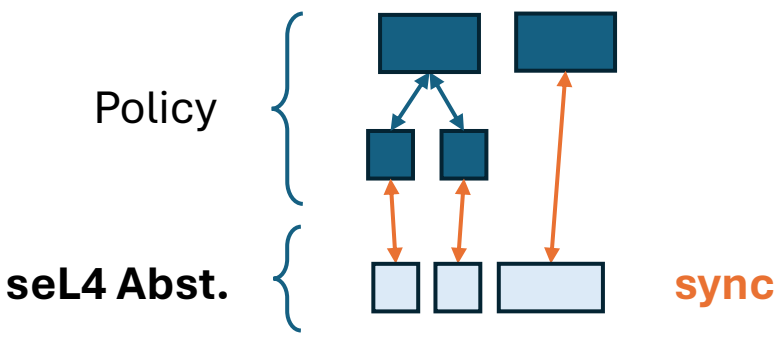
Don't build buddy system policies on (kernel) abstractions that are expensive to provision!

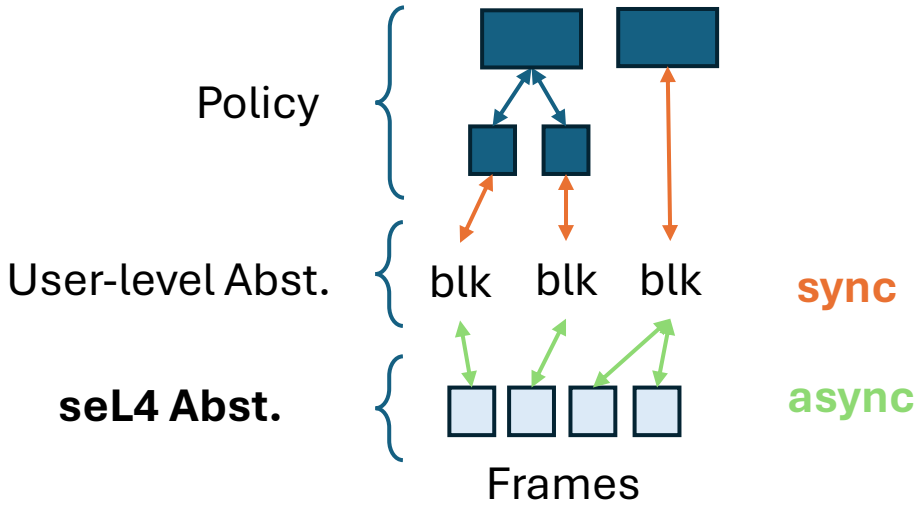
Reproduce seL4 LibOS' call chain on the Linux buddy



Create extra abstraction layer to speed up split/merge operations

- Buddy system builds on high-level abstractions and serves requests efficiently
- Split/merge operations should get “usable” block right away
- Construct high-level abstractions with *Frame* objects
- Frame-provisioning happens asynchronously (e.g., pre-provision)





Refactor the Allocator

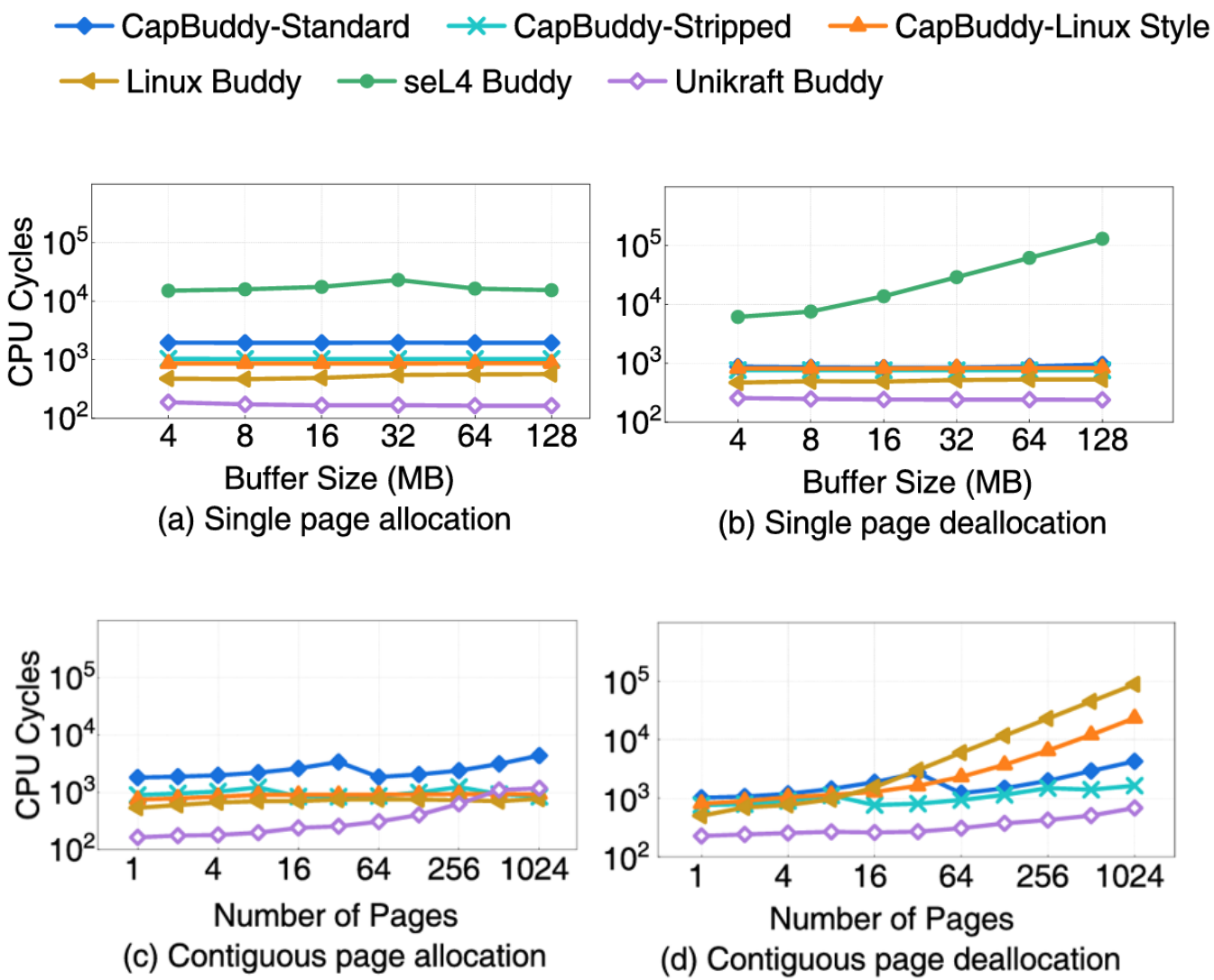
- Split-design
  - A **front-end**, which builds on the high-level abstractions, can provide fastpaths for handling memory requests
  - A **front-end** maintains a list of available Frame sequences where high-level abstractions build

Like a pool for available Frames

  - A **back-end** can work like a daemon and provision the low-level abstractions (i.e., Frames) using syscalls
  - A **back-end** can use watermark to monitor the available high-level abstractions of the front-end, and provide/reclaim them dynamically

Performance

- Our allocator – CapBuddy
- Two types of front-ends
  - Linux-style buddy allocator
  - Bitmap-based buddy allocator



End-to-end Latency of Alloc+Map & Unmap+Free

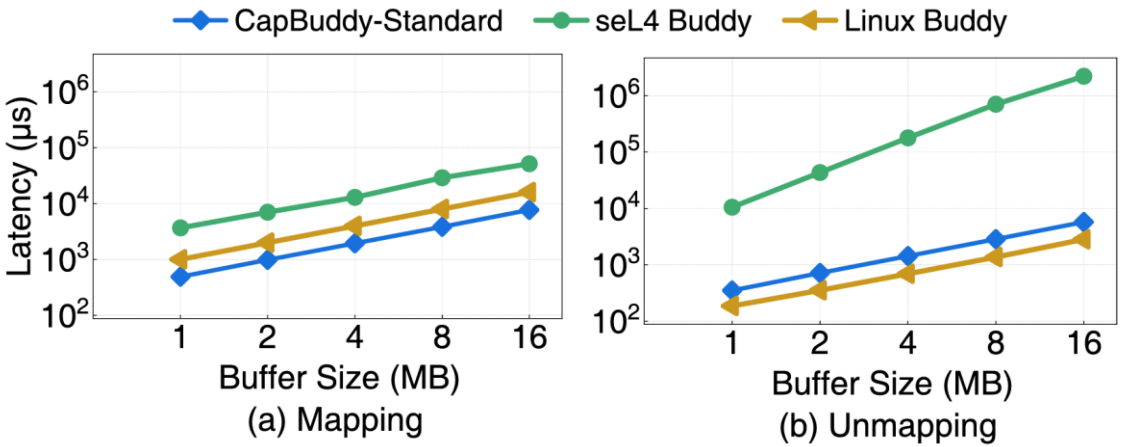
Linux

- *mmap & munmap* with *MAP_POPULATE*

seL4 LibOS

- *vspace_new_pages* & *vspace_unmap_pages*

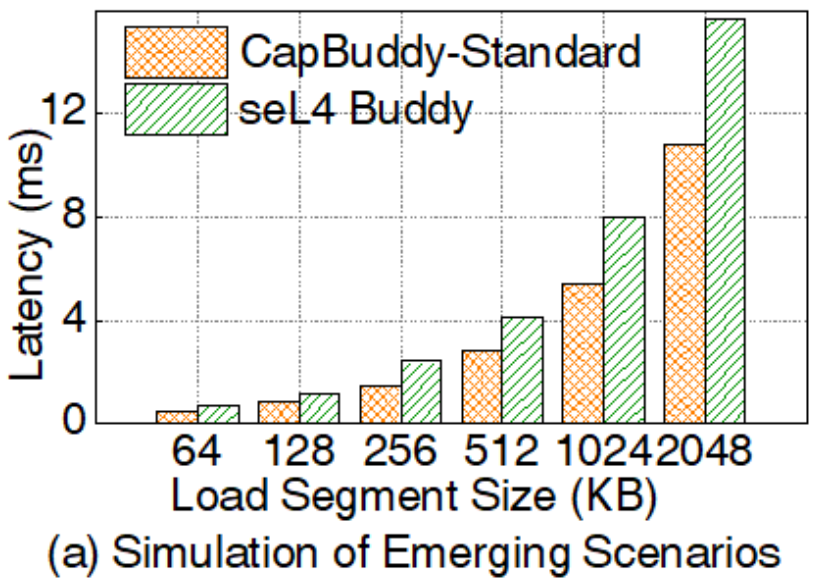
Evaluated System	Alloc + Mmap	Dealloc + Unmap
Linux Buddy	14 810	14 389
CapBuddy-Standard	5716	5731
seL4 Buddy	27 689	11 076



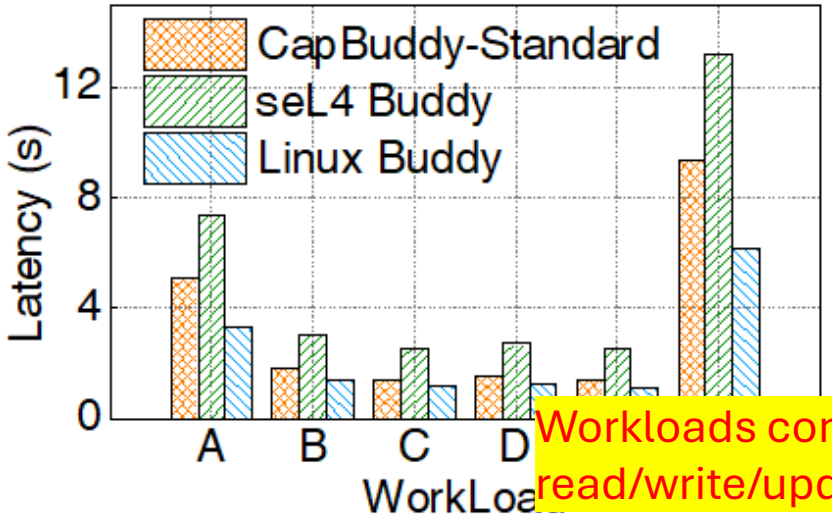
General Benchmark

- Loading ELF segments by creating mappings dynamically.
- Dynamic mapping management using SQLite3 as a benchmark

Overheads optimised by 40%



(a) Simulation of Emerging Scenarios



(b) SQLite3 YCSB evaluation

Workloads comprise of different read/write/update/scan operations

Takeaways

- The buddy system itself is not the bottleneck
- Synchronously provisioning capability-backed memory abstraction is expensive
- Moving provisioning off the critical path makes seL4-based buddy system comparable to that of Linux’s

Questions?

Thanks for your listening!