Microkernel Goes General: Performance and Compatibility in the HongMeng Production Microkernel

Haibo Chen, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, et al. and Fengwei Xu. **USENIX OSDI'24**
Huawei Central Software Institute and Shanghai Jiao Tong University

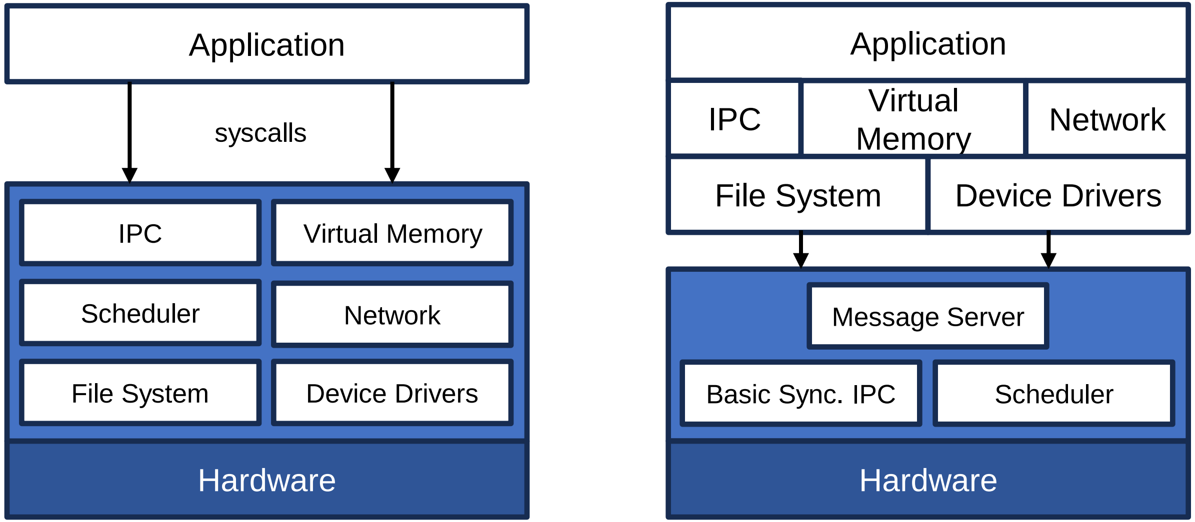
2024. 09. 05

Presentation by Nakyeong Kim
nkkim@dankook.ac.kr

Contents

1. Introduction

Monolithic vs Micro



1. Introduction

Monolithic vs Micro

	Monolithic Kernel	Microkernel
Functionality	Rich	Minimized
Isolation (Security)	Low	High
Performance	High	Low
Extensibility	Low	High
Communication	Direct Function Call	IPC
Usage	General-purpose (server)	Embedded System (safety-critical)

1. Introduction

SOTA Microkernels’ Challenges

- Compatibility
  - Minimal POSIX subset compliance via custom libraries

- Performance
  - Increased IPC frequency
  - Double bookkeeping
  - Capability-based access control

1. Introduction

HUAWEI 鸿蒙

HongMeng's Key Design

- Minimal Functionality
  - Least-privileged and well-isolated OS services
  - Thread scheduler, serial/timer drivers and lightweight access control

- Maximal Compatibility
  - Linux API/ABI compliant and driver reuse

- Performance-first Structure
  - IPC overhead/frequency mitigation
  - Address token-based access control

1. Introduction

HUAWEI 鸿蒙: primordial world

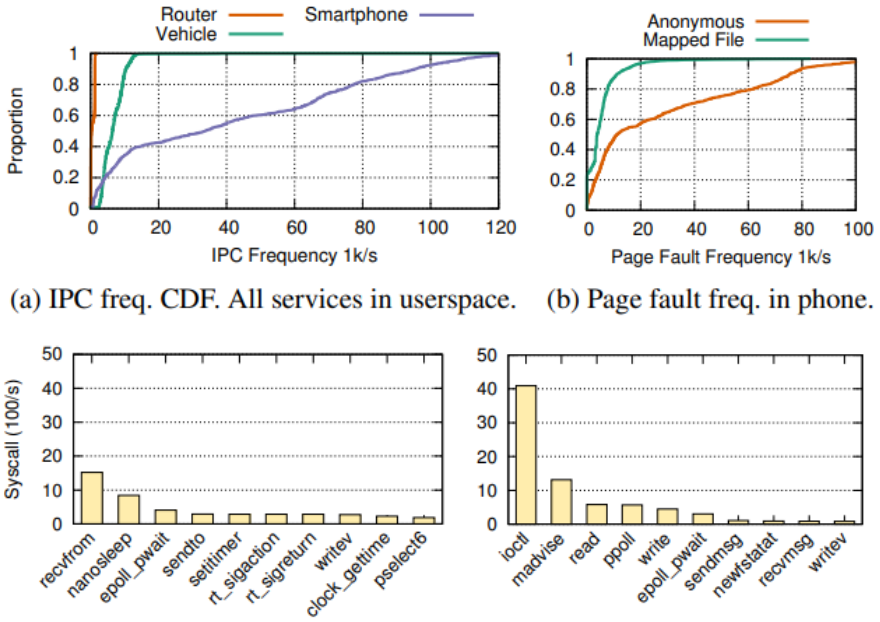
HarmonyOS

HongMeng's Key Design

- Minimal Functionality
  - Least-privileged and well-isolated OS services
  - Thread scheduler, serial/timer drivers and access control
- Maximal Compatibility
  - Linux API/ABI compliant and driver reuse
- Performance-first Structure
  - IPC overhead/frequency mitigation
  - Address token-based access control

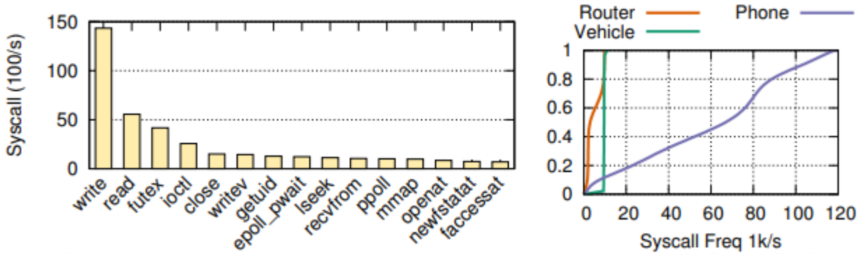
2. Observations

1) Increasing IPC Frequency



(a) IPC freq. CDF. All services in userspace. (b) Page fault freq. in phone.

(c) Syscall dist. and freq. in routers. (d) Syscall dist. and freq. in vehicles.

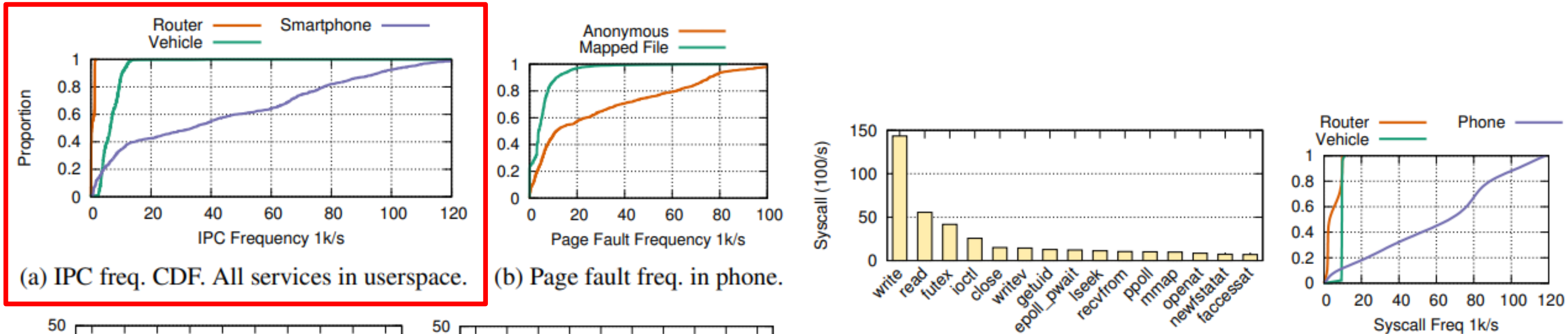


(e) Syscall dist. and freq. in smartphones. (f) Syscall freq. CDF.

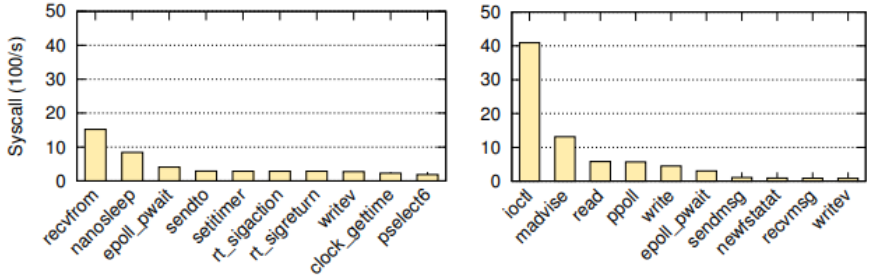
Figure 1: Characteristics of emerging scenarios obtained from the deployment of *HM* in tens of millions of devices. All OS services in *HM* are configured to be well-isolated in userspace.

2. Observations

1) Increasing IPC Frequency



(a) IPC freq. CDF. All services in userspace. (b) Page fault freq. in phone. (e) Syscall dist. and freq. in smartphones. (f) Syscall freq. CDF.

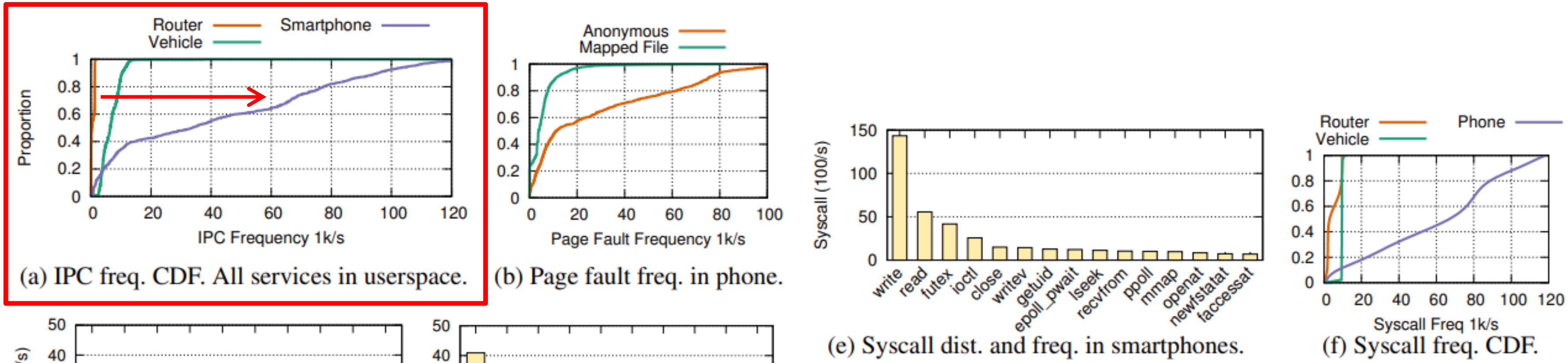


(c) Syscall dist. and freq. in routers. (d) Syscall dist. and freq. in vehicles.

Figure 1: Characteristics of emerging scenarios obtained from the deployment of *HM* in tens of millions of devices. All OS services in *HM* are configured to be well-isolated in userspace.

2. Observations

1) Increasing IPC Frequency

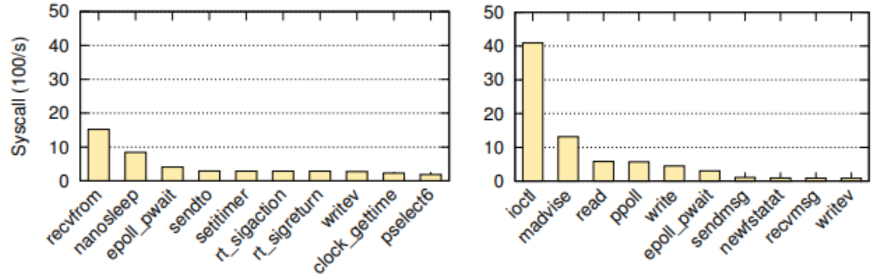


(a) IPC freq. CDF. All services in userspace.

(b) Page fault freq. in phone.

(e) Syscall dist. and freq. in smartphones.

(f) Syscall freq. CDF.



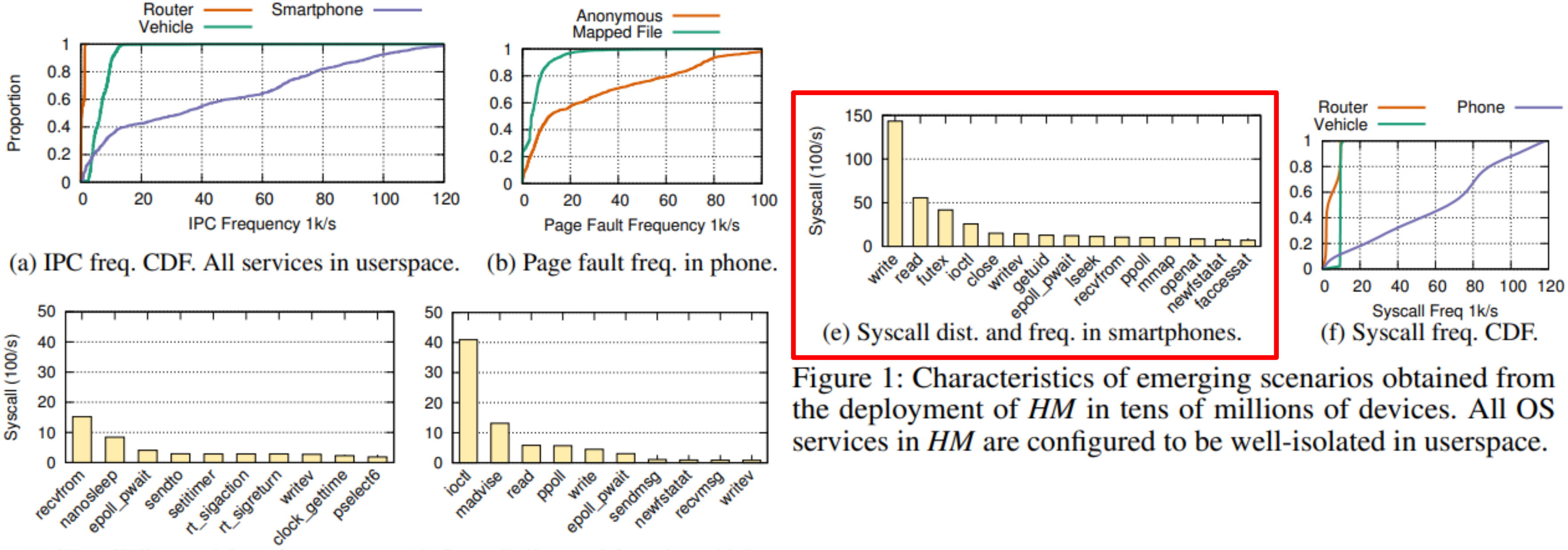
(c) Syscall dist. and freq. in routers.

(d) Syscall dist. and freq. in vehicles.

Figure 1: Characteristics of emerging scenarios obtained from the deployment of *HM* in tens of millions of devices. All OS services in *HM* are configured to be well-isolated in userspace.

2. Observations

1) Increasing IPC Frequency



(a) IPC freq. CDF. All services in userspace.

(b) Page fault freq. in phone.

(e) Syscall dist. and freq. in smartphones.

(f) Syscall freq. CDF.

(c) Syscall dist. and freq. in routers.

(d) Syscall dist. and freq. in vehicles.

Figure 1: Characteristics of emerging scenarios obtained from the deployment of *HM* in tens of millions of devices. All OS services in *HM* are configured to be well-isolated in userspace.

2. Observations

2) Double Bookkeeping

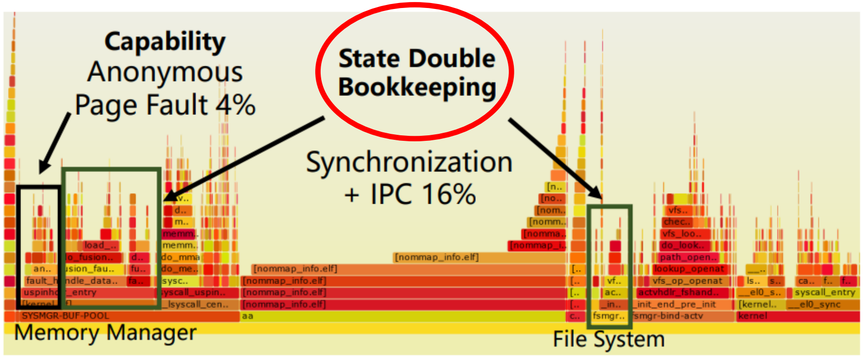


Figure 2: CPU flame graph of smartphone app startup in *HM*. Services coalescing and kernel paging are disabled.

- Minimal Principle
  - No centralized repository for shared objects (e.g., fd, page cache), but not

2. Observations

3) Capability-based Access Control

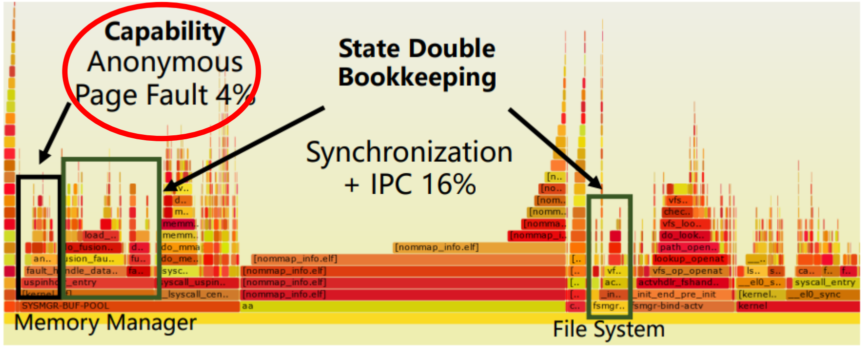


Figure 2: CPU flame graph of smartphone app startup in *HM*. Services coalescing and kernel paging are disabled.

- Hiding kernel objects(e.g., page table) introduce performance overhead because of frequent updating

2. Observations

4) Ecosystem Compatibility

- Achieving minimal subset of POSIX compliance via custom runtime libraries face deployment issues: not being binary compatible

- Challenging to implement efficient fd multiplexing(e.g., poll) and vector syscalls (e.g., ioctl)

2. Observations

5) Driver Reuse

- Deploying HM on smartphones require more than 700 drivers, while routers require fewer than 20 drivers

- It would take more than 5,000 person-years to rewrite those drivers

3. HM Overview

Principles

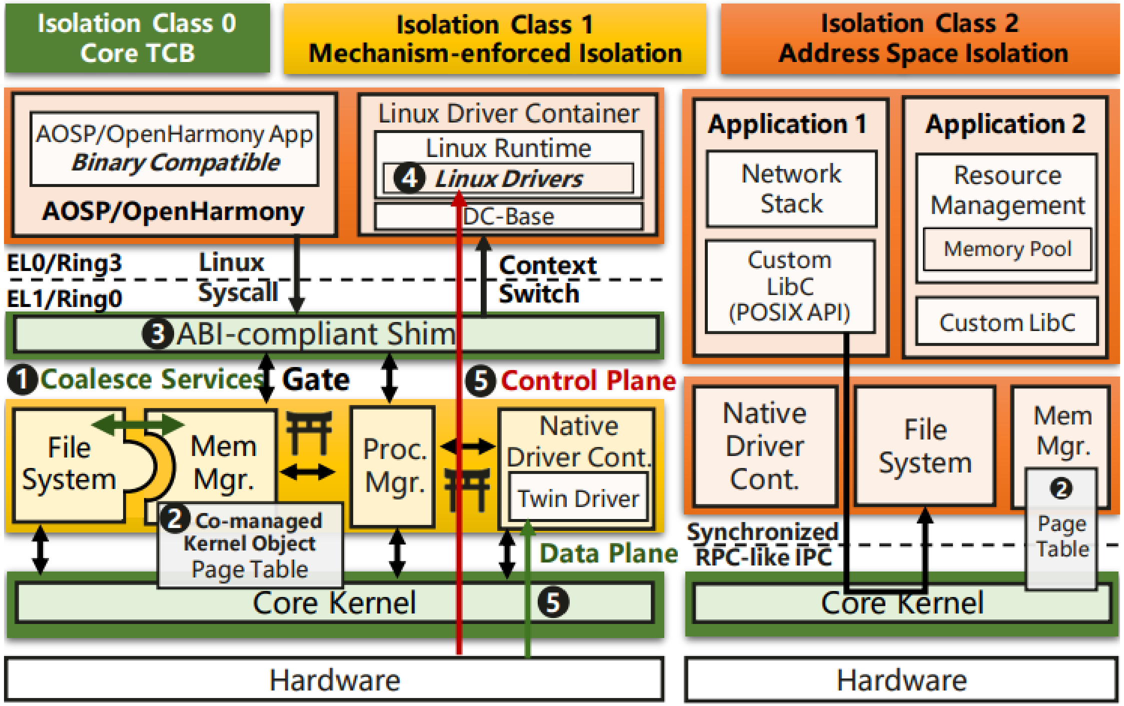
1. Retain minimality
2. Prioritize performance
3. Maximize eco-compatibility

Table 1: Design decisions of HongMeng.

	SOTA Microkernels	Hybrid Kernels	HongMeng's Design
Minimality	Minimal Kernel	Code Decoupling	**Retained:** Minimal microkernel with isolated, least-privileged OS services.
IPC	IPC w/ Fastpath	Function Call	**Enhanced:** Synchronous RPC addresses resource alloc./exhaustion/acct. issues.
Isolation	Userspace Services	Coalesce w/ Kernel	**Flexibilized:** Differentiated isolation classes for tailored isolation and performance.
Composition	Static Multi-server	Static Single Server	**Flexibilized:** Flexible composition to accommodate diverse scenarios.
Access Control	Capability-based	Object Manager	**Extended:** Address tokens enable efficient kernel objects co-management.
Memory	Paging in Userspace	Paging in Kernel	**Enhanced:** Centralized management in a service with policy-free paging in kernel.
App Interfaces	POSIX-compliant	POSIX+BSD/Win	**Extended:** Linux API/ABI compatible via an ABI-compliant shim.
Device Driver	Transplanting/VM	Native Driver	**Enhanced:** Reusing Linux drivers efficiently via driver container with twin drivers.

3. HM Overview

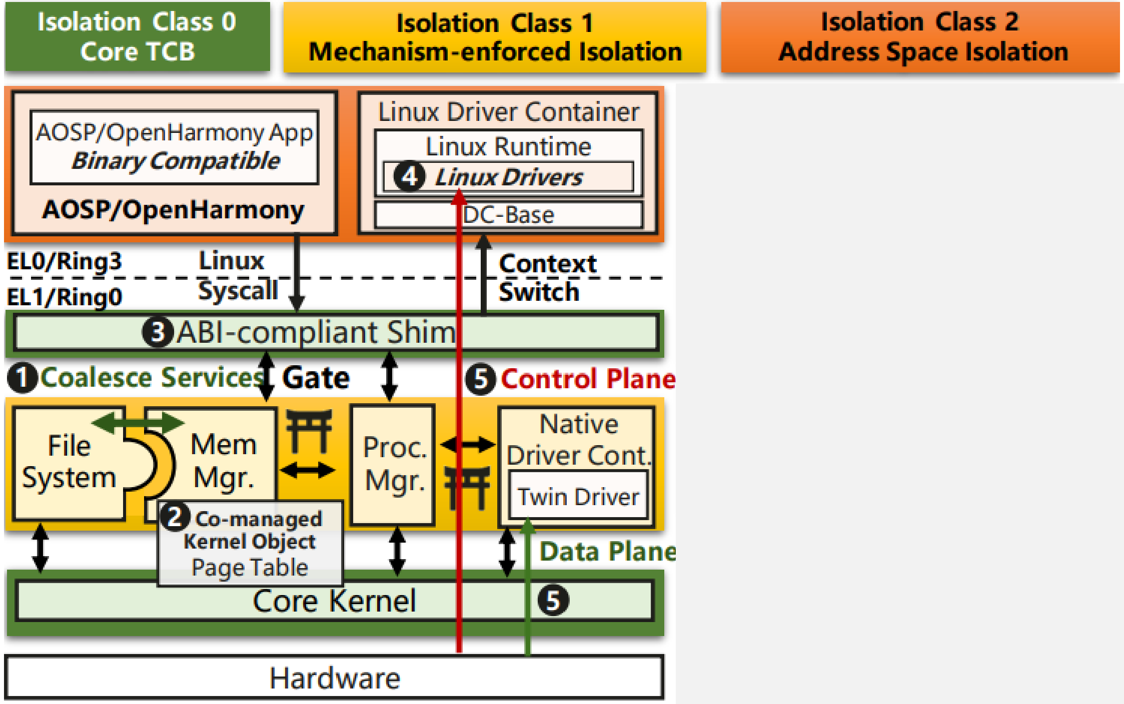
Architecture



(a) Smartphone (b) Smart Router

3. HM Overview

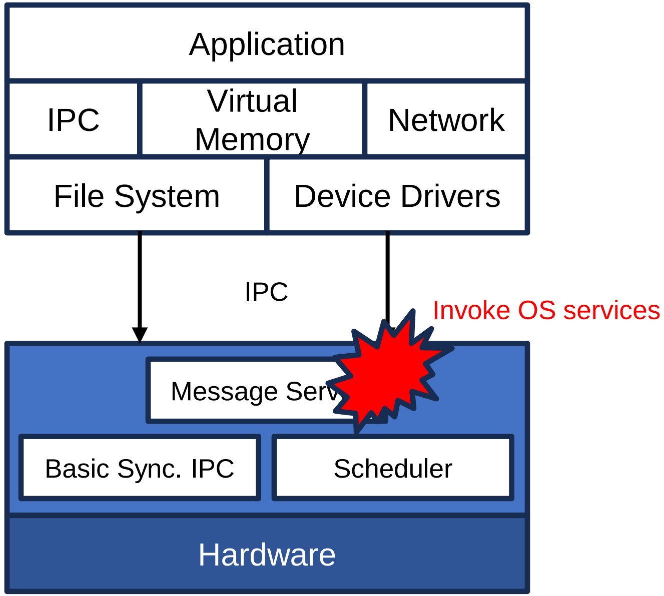
Architecture



(a) Smartphone

4. Performance Design

Synchronous IPC Fastpath

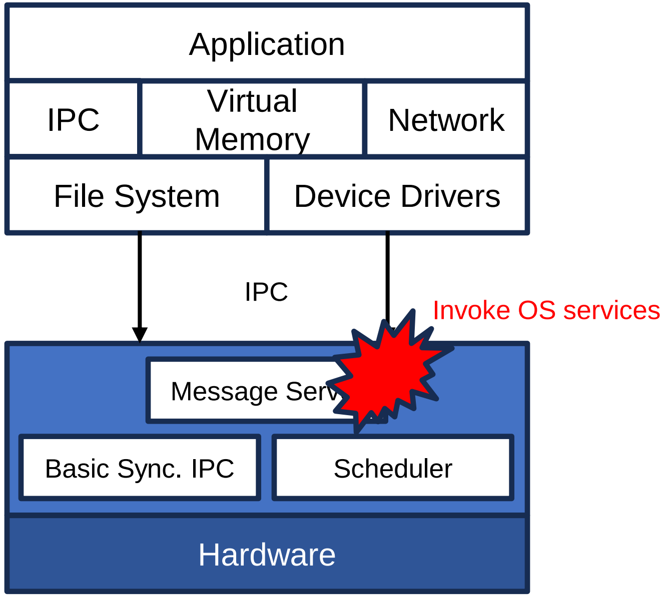


- Previous work suggest that asynchronous IPC can avoid serialization on multicore without blocking

- We observe that most IPCs are procedure calls

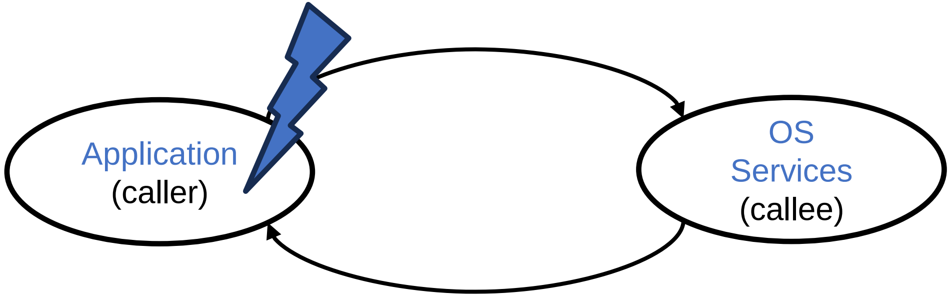
4. Performance Design

Synchronous IPC Fastpath



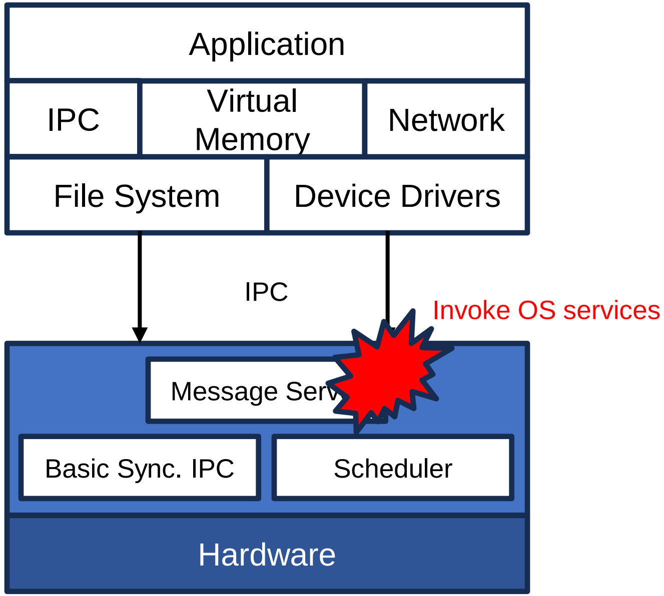
- Previous work suggest that asynchronous IPC can avoid serialization on multicore without blocking

- We observe that most IPCs are procedure calls



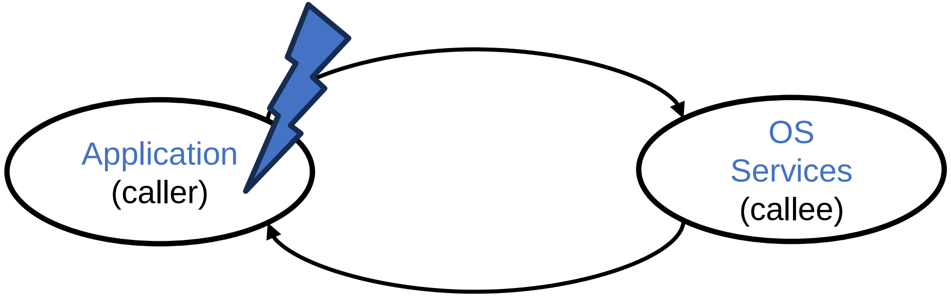
4. Performance Design

Synchronous IPC Fastpath



- Previous work suggest that asynchronous IPC can avoid serialization on multicore without blocking

- We observe that most IPCs are procedure calls



Synchronous RPC is appropriate

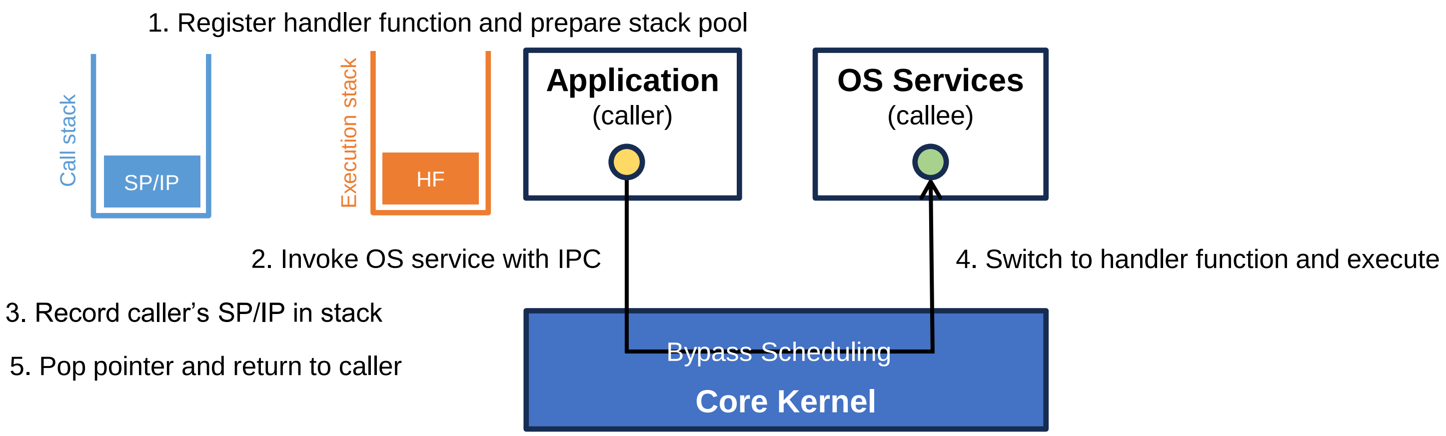
4. Performance Design

RPC Client

RPC Server

Synchronous IPC Fastpath

- RPC-like Thread Migration
  - Bypass scheduling and avoid switching registers



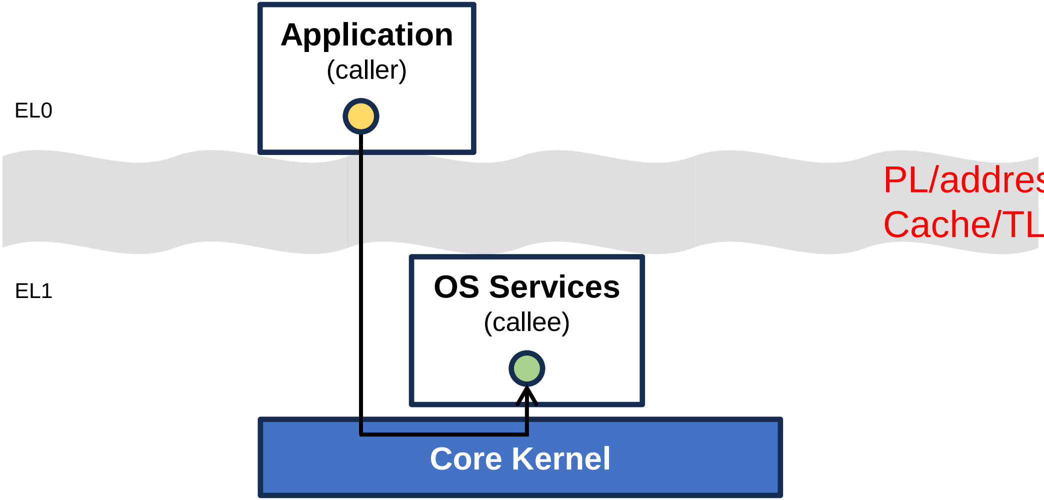
4. Performance Design

RPC Client

RPC Server

Synchronous IPC Fastpath

- RPC-like Thread Migration
  - Bypass scheduling and avoid switching registers
  - Still face performance degradation



PL/address space switching & Cache/TLB pollution

4. Performance Design

TCB: Trusted Computing Base

Differentiated Isolation Classes

- Not all services require same class of isolation

Isolation Class	Feature	Usage	Isolation Level	Example
IC0	Core TCB*	Verified, performance-critical, trusted OS services	No isolation	ABI-compliant shim
IC1	Mechanism-enforced isolation	In kernel space services (distinct domains)	Enforced isolation	ARM watchpoint, Intel PKS
IC2	Address space isolation	Non-performance-critical user space services	Enforced isolation with address space	Third-party code

4. Performance Design

TCB: Trusted Computing Base

Differentiated Isolation Classes

- Not all services require same class of isolation

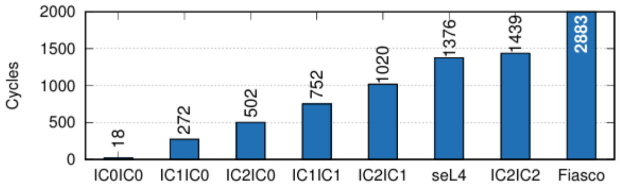


Figure 4: Round-trip IPC latency between ICx & ICy (ICxICy) in Raspberry Pi 4b. IC0 includes the core kernel. IC2 includes user apps. Zircon cannot run on Pi4b and is several times slower [49].

Isolation Class	Feature	Usage	Isolation Level	Example
IC0	Core TCB*	Verified, performance-critical, trusted OS services	No isolation	ABI-compliant shim
IC1	Mechanism-enforced isolation	In kernel space services (distinct domains)	Enforced isolation	ARM watchpoint, Intel PKS
IC2	Address space isolation	Non-performance-critical user space services	Enforced isolation with address space	Third-party code

4. Performance Design

Flexible Composition

- FS(file mapping) and memory manager(page cache) coupled tightly
- Double bookkeeping of shared states introduce memory footprint and synchronization overhead
- HM adopt configurable separating/shared cache approach between performance-demanding and safety-critical scenarios

4. Performance Design

Flexible Composition

- FS(file mapping) and memory manager(page cache) coupled tightly
- Double bookkeeping of shared states introduce memory footprint and synchronization overhead
- HM adopt configurable separating/shared cache approach between performance-demanding and safety-critical scenarios

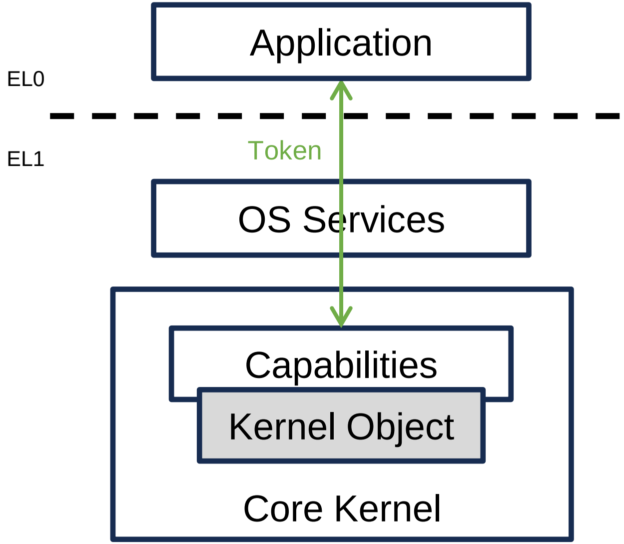
Table 3: Performance improved by coalescing the FS service and the memory manager in the big core of Kirin9000 [57].

	Separated	Coalesced	Linux
Page Fault (Cycles)	7092	5290 (Sep. Cache) 3785 (Shr. Cache)	3432
Tmpfs Write (MB/s)	1492	2067	2133
Memory Footprint (MB)	190	120	N/A

4. Performance Design

Address Token-based Access Control

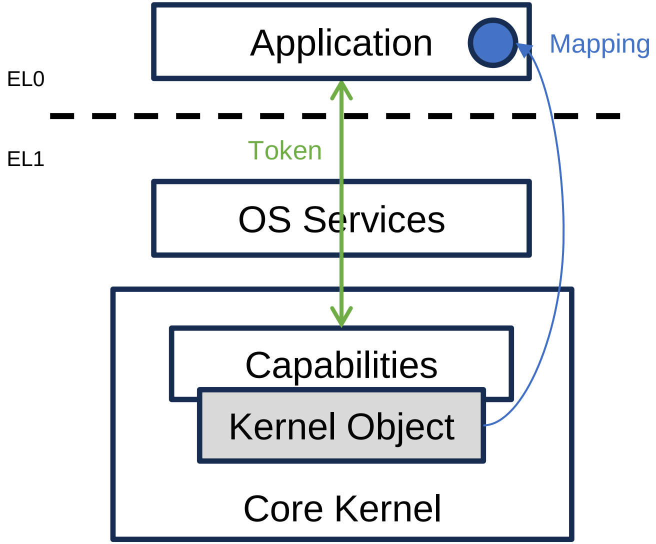
- Previous capability-based access control describe clear relationship, but slow



4. Performance Design

Address Token-based Access Control

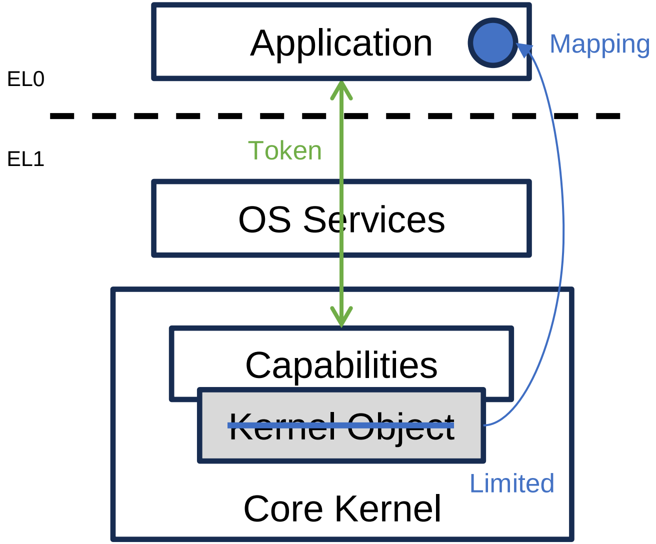
- Previous capability-based access control describe clear relationship, but slow



4. Performance Design

Address Token-based Access Control

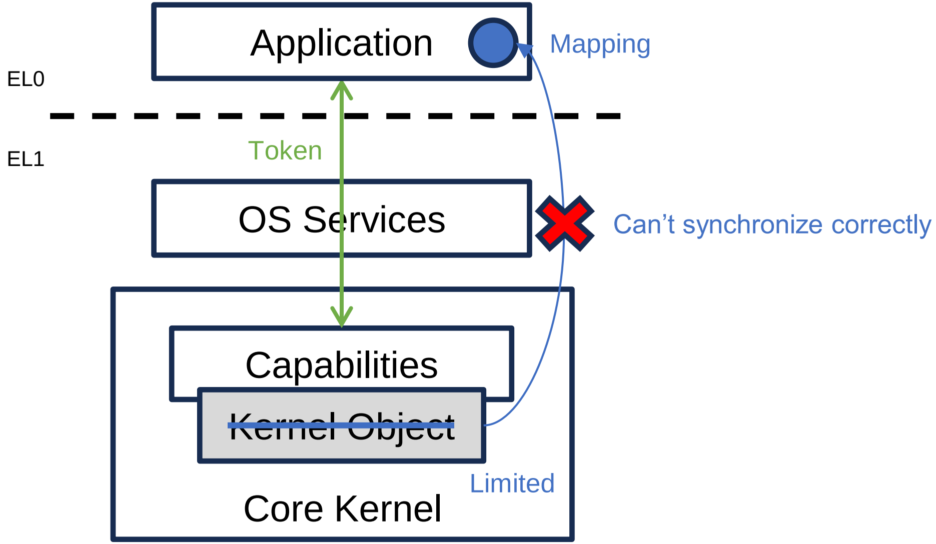
- Previous capability-based access control describe clear relationship, but slow



4. Performance Design

Address Token-based Access Control

- Previous capability-based access control describe clear relationship, but slow



4. Performance Design

Address Token-based Access Control

- Previous capability-based access control describe clear relationship, but slow
- ATAC include broader range of objects and enable efficient co-management
- To update read-only kernel object, new syscall(writev) should be used

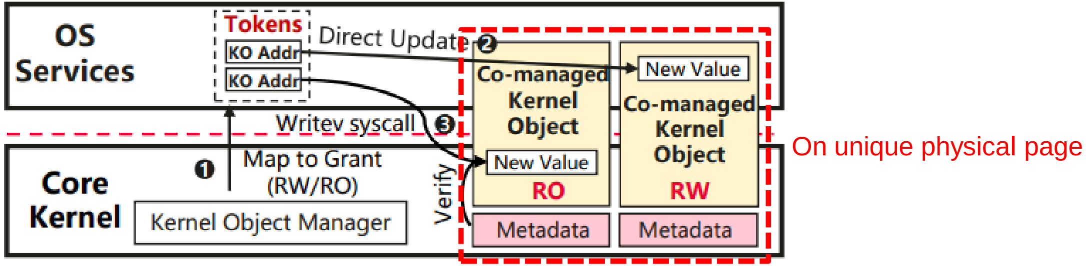


Figure 5: Address token-based access control in *HM*. ❶ Map kernel object's page to grant. ❷ Direct access to RW objects. ❸ Use `writev` to update RO objects, verified by the kernel.

4. Performance Design

Address Token-based Access Control

- Previous capability-based access control describe clear relationship, but slow
- ATAC include broader range of objects and enable efficient co-management
- To update read-only kernel object, new syscall(writev) should be used

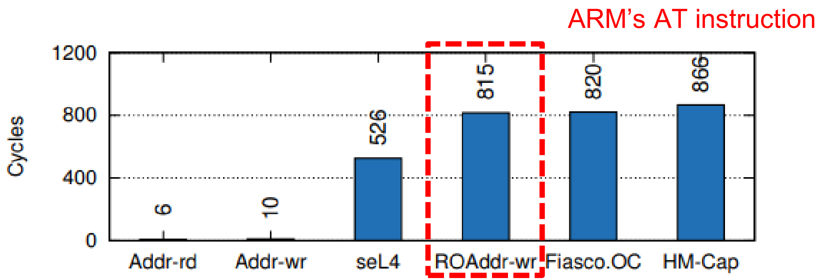


Figure 6: Latency of accessing kernel objects on Raspberry Pi 4b. *Addr-rd/wr* represent address tokens in *HM*. *ROAddr-wr* represents writing to read-only objects in *HM*.

4. Performance Design

Policy-free Kernel Paging

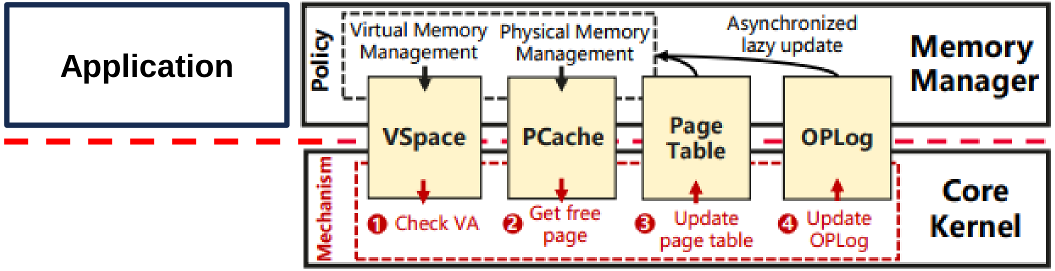


Figure 7: Policy-free kernel paging in *HM*. On page fault, the kernel checks the address ❶ and, if anonymous, ❷/❸ maps a pre-allocated page, and ❹ records an OPLog.

- Performance degradation occurred due to slow anonymous paging
- HM memory manager is outside core kernel to eliminate IPC round-trip

4. Performance Design

Policy-free Kernel Paging

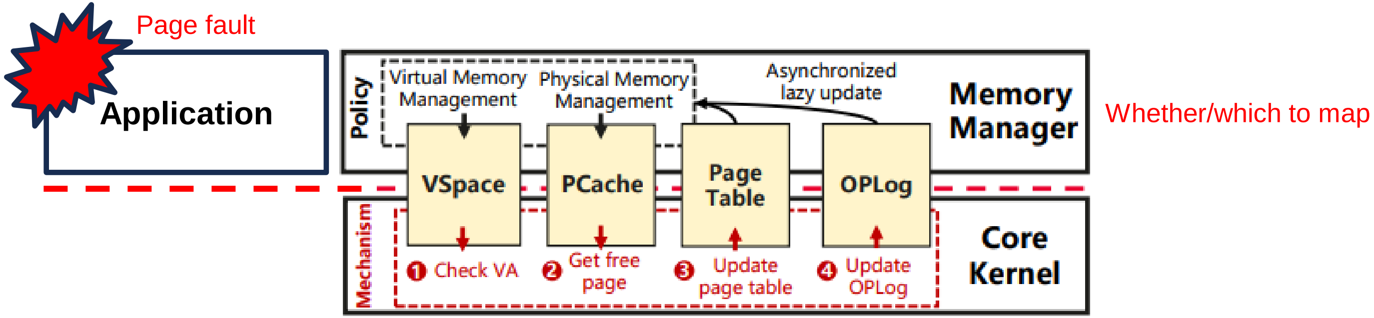


Figure 7: Policy-free kernel paging in *HM*. On page fault, the kernel checks the address ❶ and, if anonymous, ❷/❸ maps a pre-allocated page, and ❹ records an OPLog.

- Performance degradation occurred due to slow anonymous paging
- Pre-policy-decision and pre-page-allocation eliminate additional IPC roundtrip

4. Performance Design

Policy-free Kernel Paging

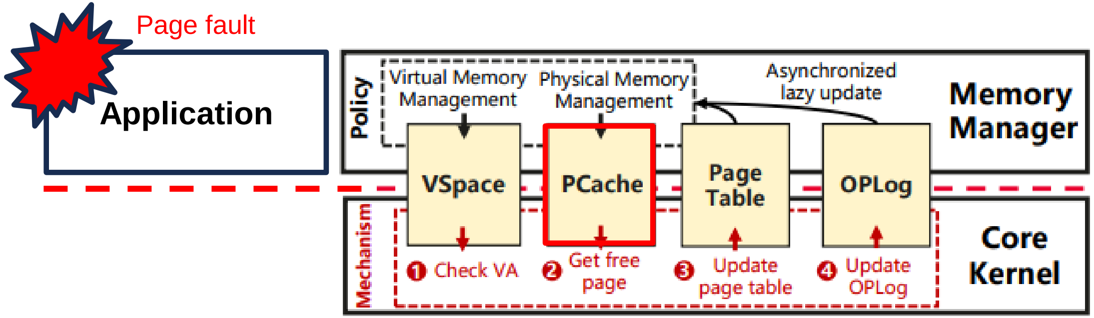


Figure 7: Policy-free kernel paging in *HM*. On page fault, the kernel checks the address ❶ and, if anonymous, ❷/❸ maps a pre-allocated page, and ❹ records an OPLog.

- Performance degradation occurred due to slow anonymous paging
- Pre-policy-decision and pre-page-allocation eliminate additional IPC roundtrip

4. Performance Design

Policy-free Kernel Paging

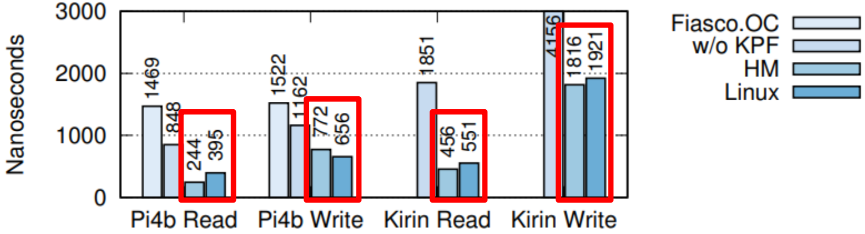
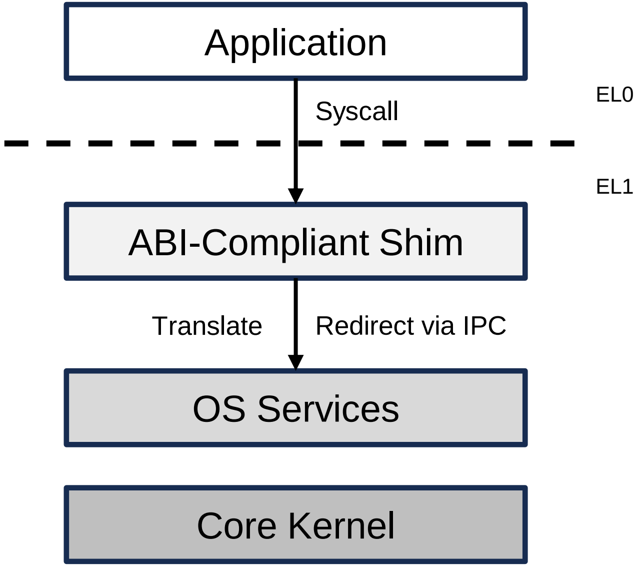


Figure 8: Page fault latency of private anonymous memory. Read is optimized with zero page. seL4 is not included since it does not support demand paging by default.

- Pre-allocated way reduce flexibility and introduce additional memory footprints

5. Compatibility Design

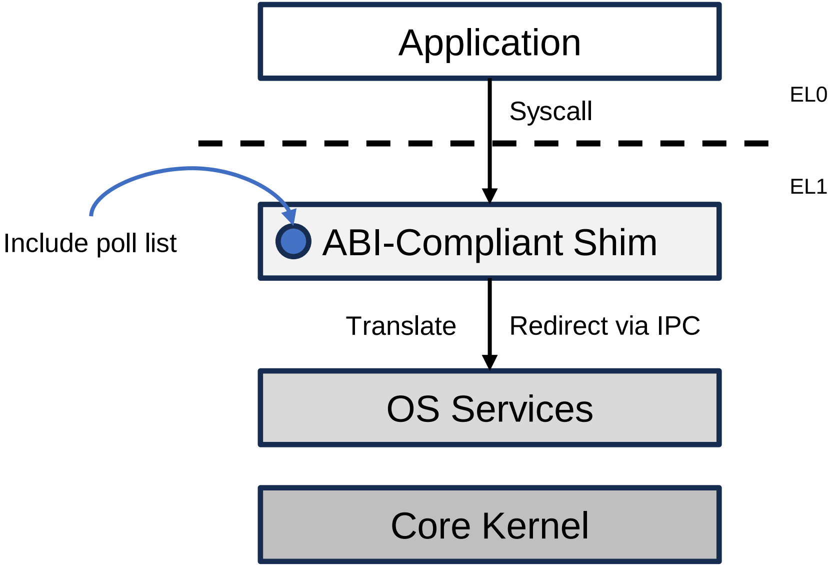
Linux ABI



- ABI-compliant shim serve central repository for global states(e.g., fd table)

5. Compatibility Design

Linux ABI



- ABI-compliant shim serve central repository for global states(e.g., fd table)

5. Compatibility Design

Driver Container

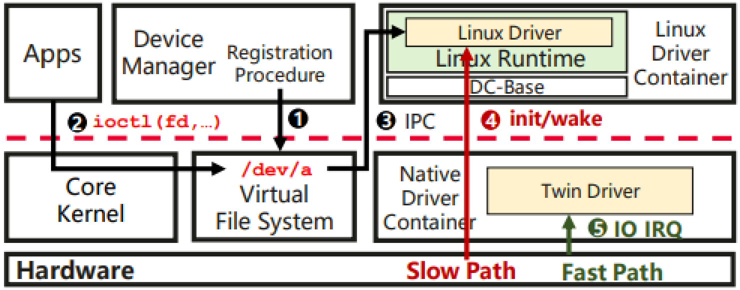
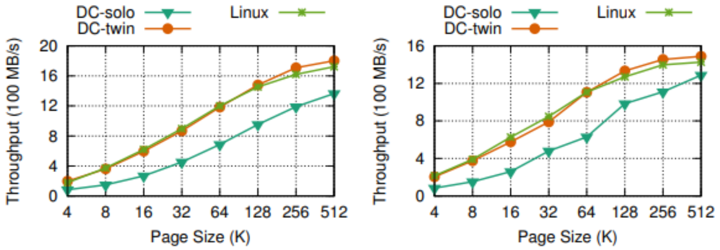


Figure 9: Drivers in *HM*. The device manager creates file nodes in the VFS ❶. VFS redirects invocations ❷ to drivers ❸. *HM* improves performance by separating the control ❹/data ❺ plane.

- LDC provide all Linux KAPIs by reusing Linux code base as userspace runtime
- Drivers can access hardware device directly
- All resource management functionalities are removed because runtime rely on HM
- Control plane and data plane are separated for performance

5. Compatibility Design

Driver Container



(a) BIO read. (b) BIO write.

Figure 10: Block I/O throughput on Kirin9000. DC-twin applies data and control plane separation, while DC-solo does not.

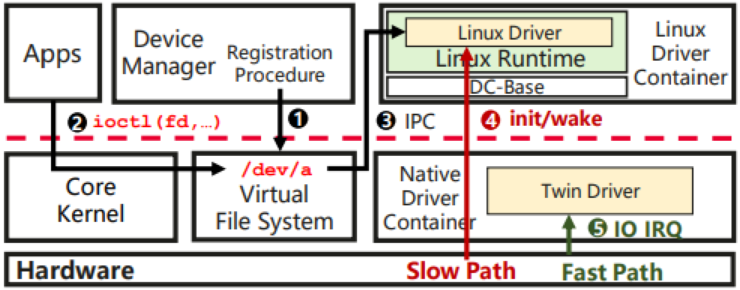


Figure 9: Drivers in *HM*. The device manager creates file nodes in the VFS ❶. VFS redirects invocations ❷ to drivers ❸. *HM* improves performance by separating the control ❹/data ❺ plane.

- Twin driver handle I/O IRQs on performance critical path (rewrite data handling procedure)
- When handling non-I/O IRQs and errors, LDC synchronize updated states back to twin driver

6. Evaluation

Environment

- Scenario
  - Smart phones
  - Smart vehicles
  - Smart routers
- Comparison
  - Linux 5.10 (optimized)
  - HM
- Benchmark
  - LMBench
  - Geekbench

Table 5: LMbench results.

Benchmark Commands[1]	Unit	Linux	*HM*	Norm.[2]
`lat_unix -P 1`	μs	10.23	10.39	0.98
`lat_tcp -m 16`	μs	21.22	17.19	1.23
`lat_tcp -m 16K`	μs	24.54	18.9	1.29
`lat_tcp -m 1K (Same Core)`	μs	21.21	17.19	1.23
`lat_tcp -m 1K (Cross core)`	μs	37.96	25.66	1.47
`lat_udp -m 16`	μs	17.83	19.48	0.92
`lat_udp -m 16K`	μs	23.63	22.02	1.07
`lat_udp -m 1K (Same Core)`	μs	18.04	19.55	0.92
`lat_udp -m 1K (Cross core)`	μs	34.17	26.84	1.27
`bw_tcp -m 10M`	MB/s	1812	3109	1.71
`bw_unix`	MB/s	7124	8478	1.19
`bw_mem 256m bcopy`	MB/s	17696	17202	1.02
`bw_mem 512m frd`	MB/s	14514	14593	0.99
`bw_mem 256m fcp`	MB/s	17492	15867	0.91
`bw_mem 512m fwr`	MB/s	34771	35318	1.01
`bw_file_rd 512M io_only`	MB/s	8976	9396	1.04
`bw_mmap_rd 512M mmap_only`	MB/s	26073	27520	1.05
`lat_mmap 512m`	μs	3315	3628	0.91
`lat_pagefault`	μs	0.83	0.78	1.06
`lat_ctx -s 16 8`	μs	4.53	3.41	1.32
`bw_pipe`	MB/s	3808	4127	1.08
`lat_pipe`	μs	9.00	7.88	1.14
`lat_proc exec`	μs	336	1305	0.26
`lat_proc fork`	μs	323	1280	0.25
`lat_proc shell`	μs	2269	4778	0.47
`lat_clone (create thread)`	μs	28.6	54.3	0.52

[1] Argument "`-P 1`" is omitted.
[2] *Norm.* shows the normalized performance. For throughput, use *HM*/Linux, for latency, use Linux/*HM*. The more the better.

6. Evaluation

Environment

- Scenario
  - Smart phones
  - Smart vehicles
  - Smart routers
- Comparison
  - Linux 5.10 (optimized)
  - HM
- Benchmark
  - LMBench
  - Geekbench

Table 5: LMbench results.

Benchmark Commands[1]	Unit	Linux	*HM*	Norm.[2]
`lat_unix -P 1`	μs	10.23	10.39	0.98
`lat_tcp -m 16`	μs	21.22	17.19	1.23
`lat_tcp -m 16K`	μs	24.54	18.9	1.29
`lat_tcp -m 1K (Same Core)`	μs	21.21	17.19	1.23
`lat_tcp -m 1K (Cross core)`	μs	37.96	25.66	1.47
`lat_udp -m 16`	μs	17.83	19.48	0.92
`lat_udp -m 16K`	μs	23.63	22.02	1.07
`lat_udp -m 1K (Same Core)`	μs	18.04	19.55	0.92
`lat_udp -m 1K (Cross core)`	μs	34.17	26.84	1.27
`bw_tcp -m 10M`	MB/s	1812	3109	1.71
`bw_unix`	MB/s	7124	8478	1.19
`bw_mem 256m bcopy`	MB/s	17696	17202	1.02
`bw_mem 512m frd`	MB/s	14514	14593	0.99
`bw_mem 256m fcp`	MB/s	17492	15867	0.91
`bw_mem 512m fwr`	MB/s	34771	35318	1.01
`bw_file_rd 512M io_only`	MB/s	8976	9396	1.04
`bw_mmap_rd 512M mmap_only`	MB/s	26073	27520	1.05
`lat_mmap 512m`	μs	3315	3628	0.91
`lat_pagefault`	μs	0.83	0.78	1.06
`lat_ctx -s 16 8`	μs	4.53	3.41	1.32
`bw_pipe`	MB/s	3808	4127	1.08
`lat_pipe`	μs	9.00	7.88	1.14
`lat_proc exec`	μs	336	1305	0.26
`lat_proc fork`	μs	323	1280	0.25
`lat_proc shell`	μs	2269	4778	0.47
`lat_clone (create thread)`	μs	28.6	54.3	0.52

[1] Argument "`-P 1`" is omitted.
[2] *Norm.* shows the normalized performance. For throughput, use *HM*/Linux, for latency, use Linux/*HM*. The more the better.

21% ↓

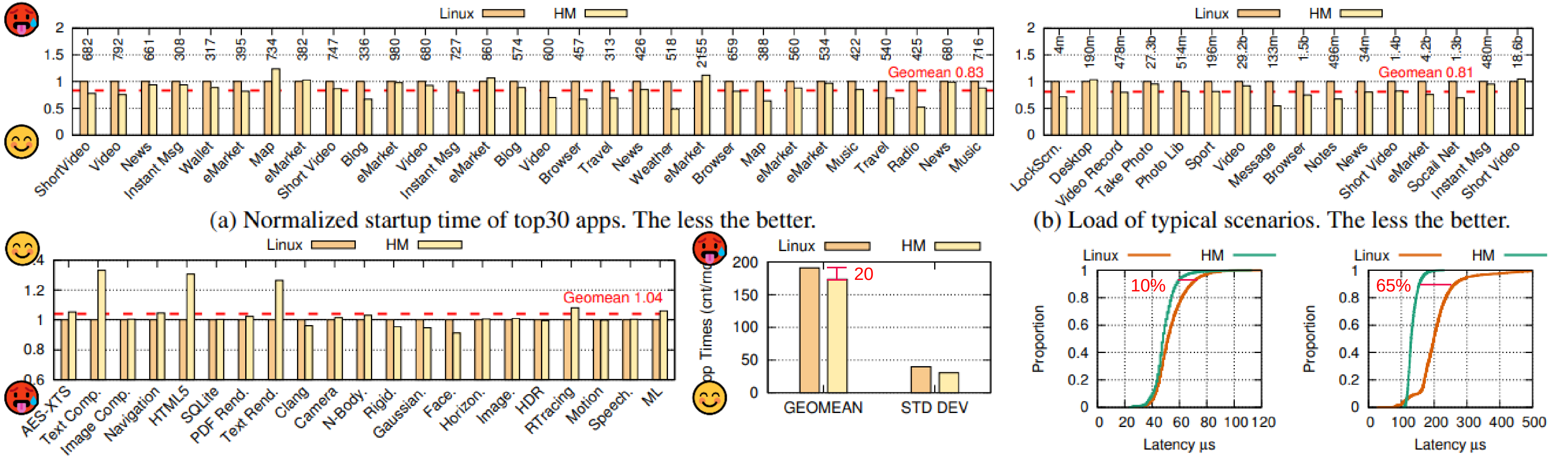
32% ↓

Copying VMAs

Additional IPCs between multiple OS services

6. Evaluation

Performance



(a) Normalized startup time of top30 apps. The less the better. (b) Load of typical scenarios. The less the better.

(c) Geekbench (single core). The more the better. (d) Frame drops. The less the better. (e) Video int. latency CDF. (f) Audio int. latency CDF.

Figure 11: Performance of *HM* compared with optimized Linux 5.10 on Kirin9000. (a), (b) and (c) normalized the result for comparison. Labels in (a) show the startup time in milliseconds on *HM*. Labels in (b) show the executed instructions on *HM*.

7. Conclusion

- For emerging applications, need for a general-purpose microkernel has increased as they require high security and isolation

- To maximize **performance** and **compatibility** while maintaining minimalism, HM applied IPC fastpaths, separation of class isolation levels, unified management of related OS services, address token-based access control, policy-less kernel paging, and driver management via device containers

- A comparison of HM and Linux 5.10 shows improvements in context switching time and memory usage, application execution time, load, frame drops, interrupt latency, and more

Q&A

Thank you!

2024. 09. 05

Presentation by Nakyeong Kim
nkkim@dankook.ac.kr