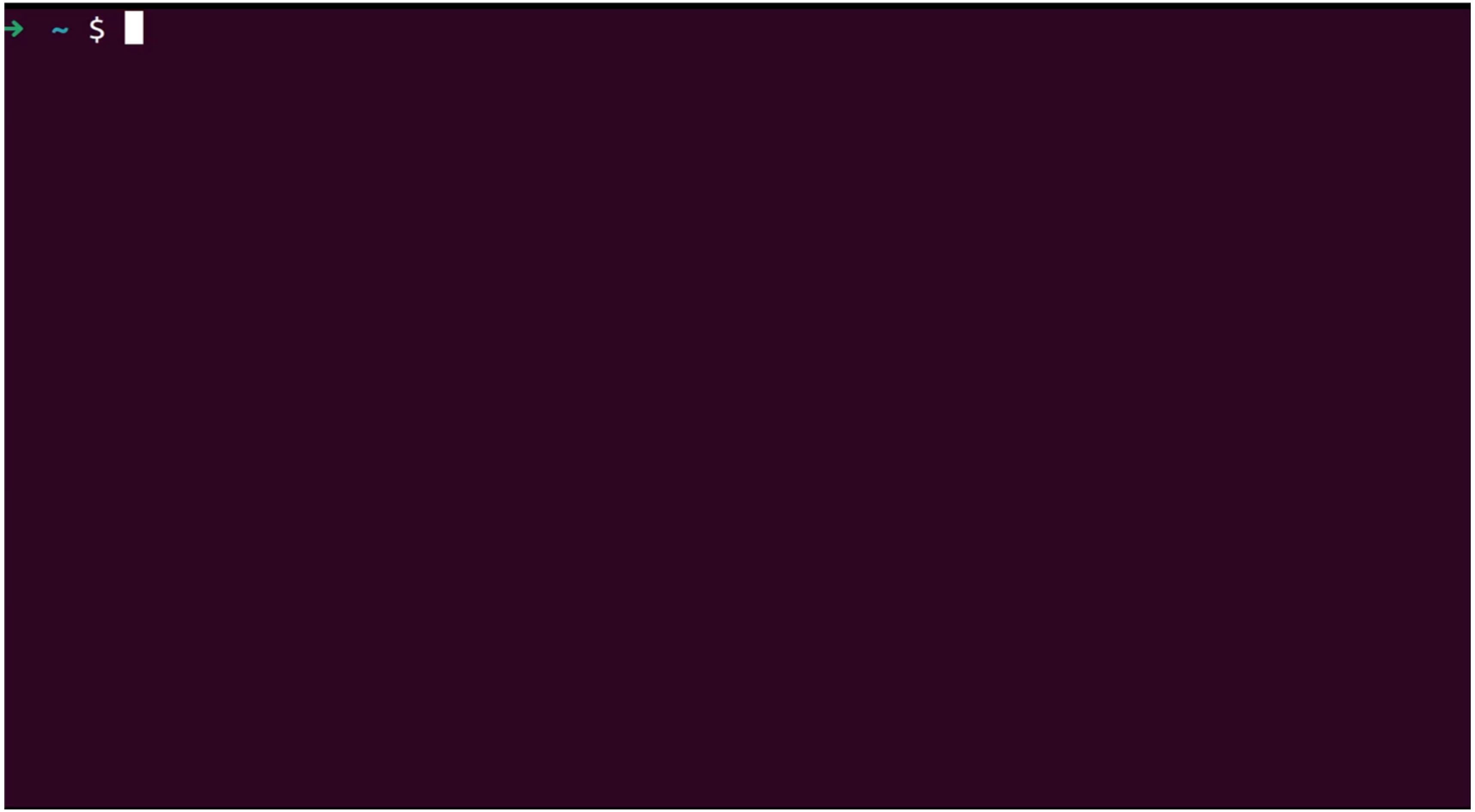
On-demand-fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications

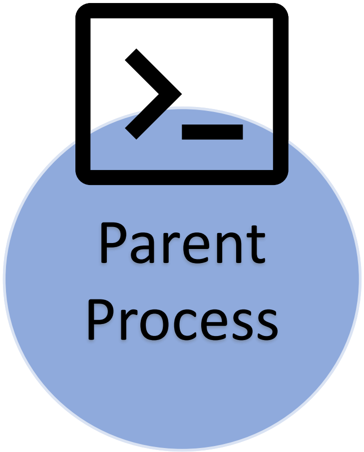
Kaiyang Zhao, Sishuai Gong, Pedro Fonseca

Purdue University



What is process fork?

Fork creates a child process by **duplicating** the calling process.



What is process fork?

Fork creates a child process by **duplicating** the calling process.

What is process fork?

Fork creates a child process by **duplicating** the calling process.

Modern uses of fork

Fuzzers

Serverless

Databases

<u>Modern Uses</u>

Modern uses of fork

Fuzzers
(Hundreds of **MBs**)

Serverless

Databases

<u>Modern Uses</u>

Modern uses of fork

Fuzzers
(Hundreds of **MBs**)

Serverless
(A few **GBs**)

Databases

Modern Uses

Modern uses of fork

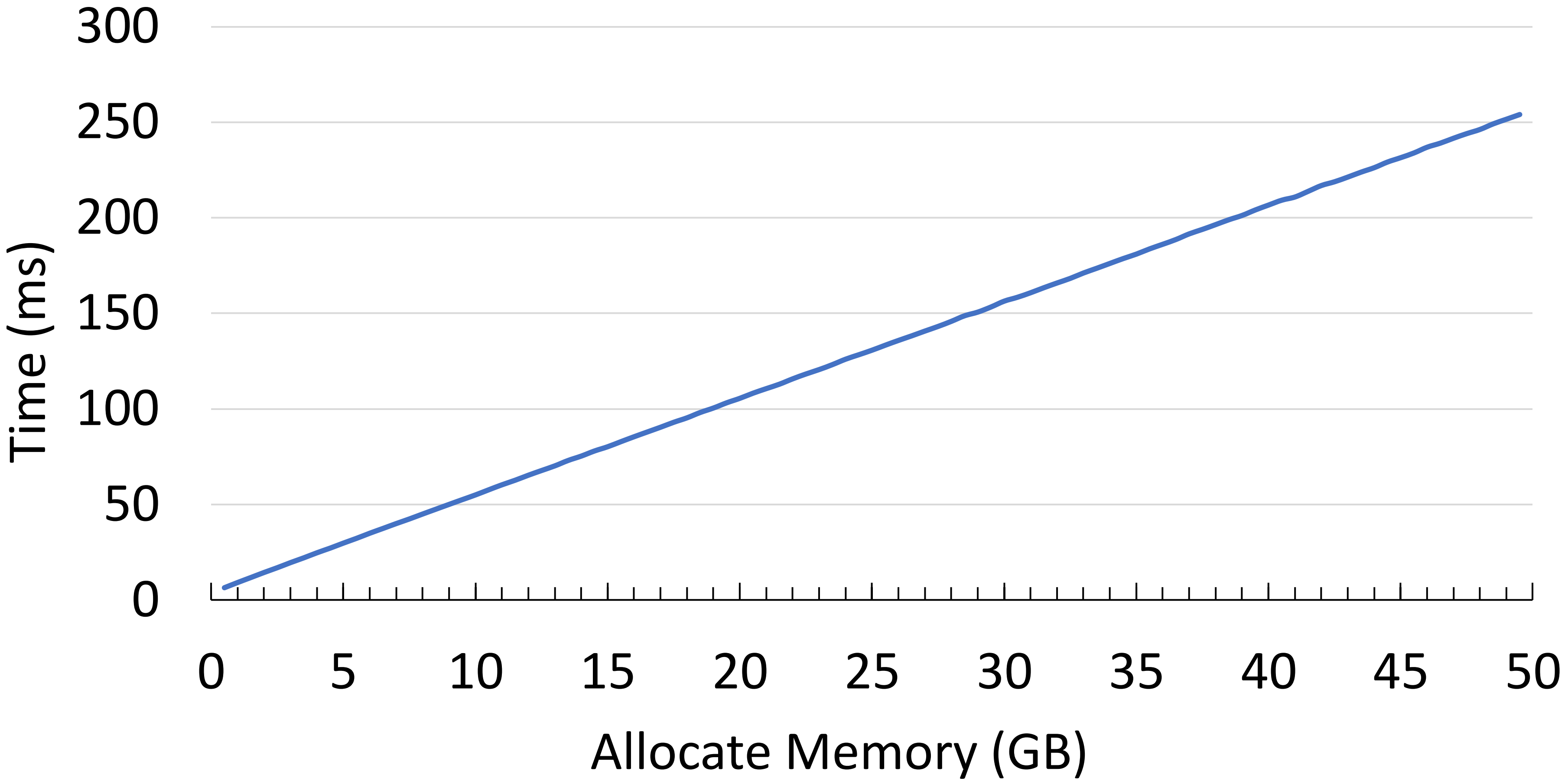
Fuzzers
(Hundreds of **MBs**)

Serverless
(A few **GBs**)

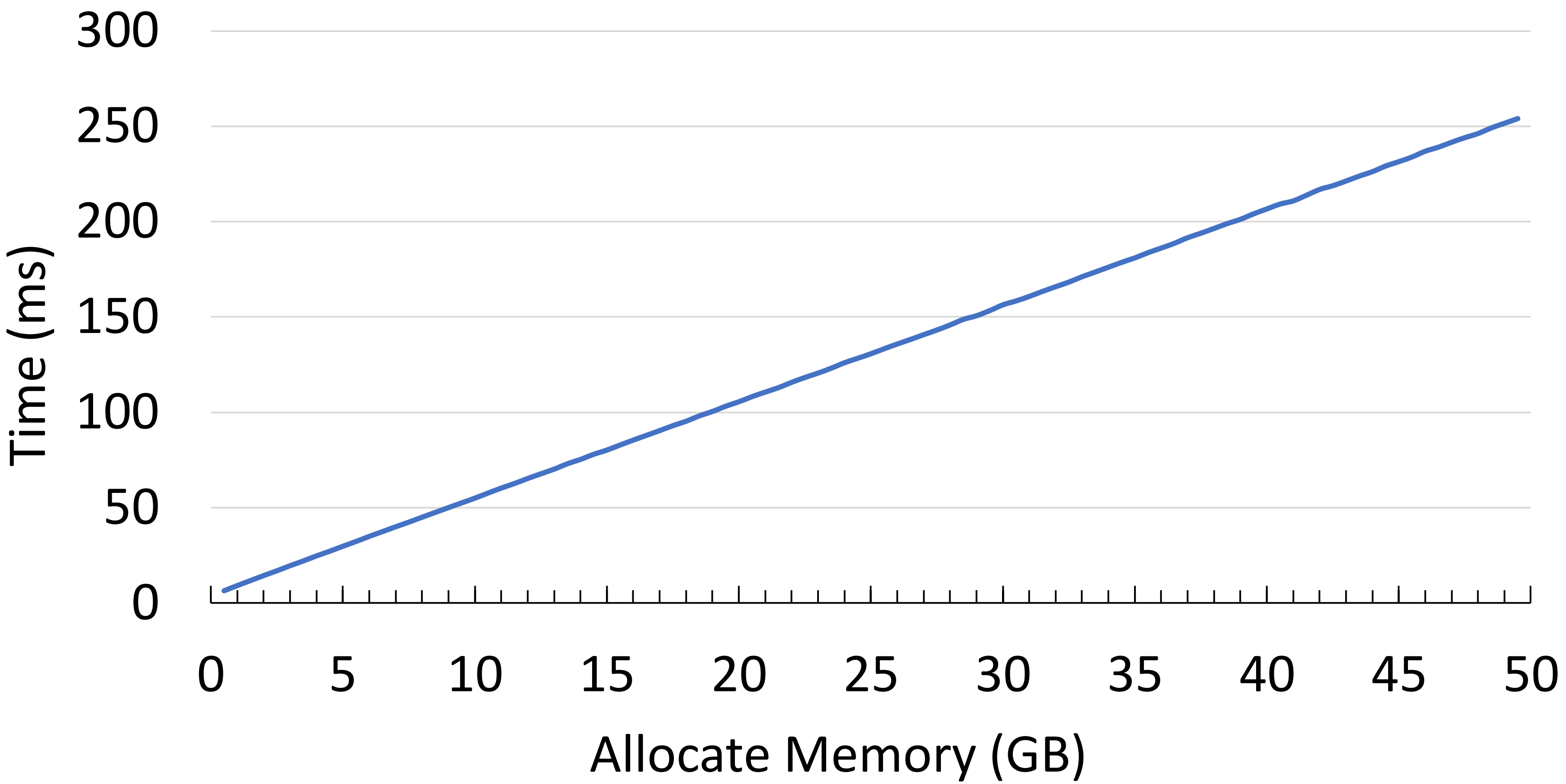
Databases
(A few **TBs**)

Modern Uses

Fork has a latency problem

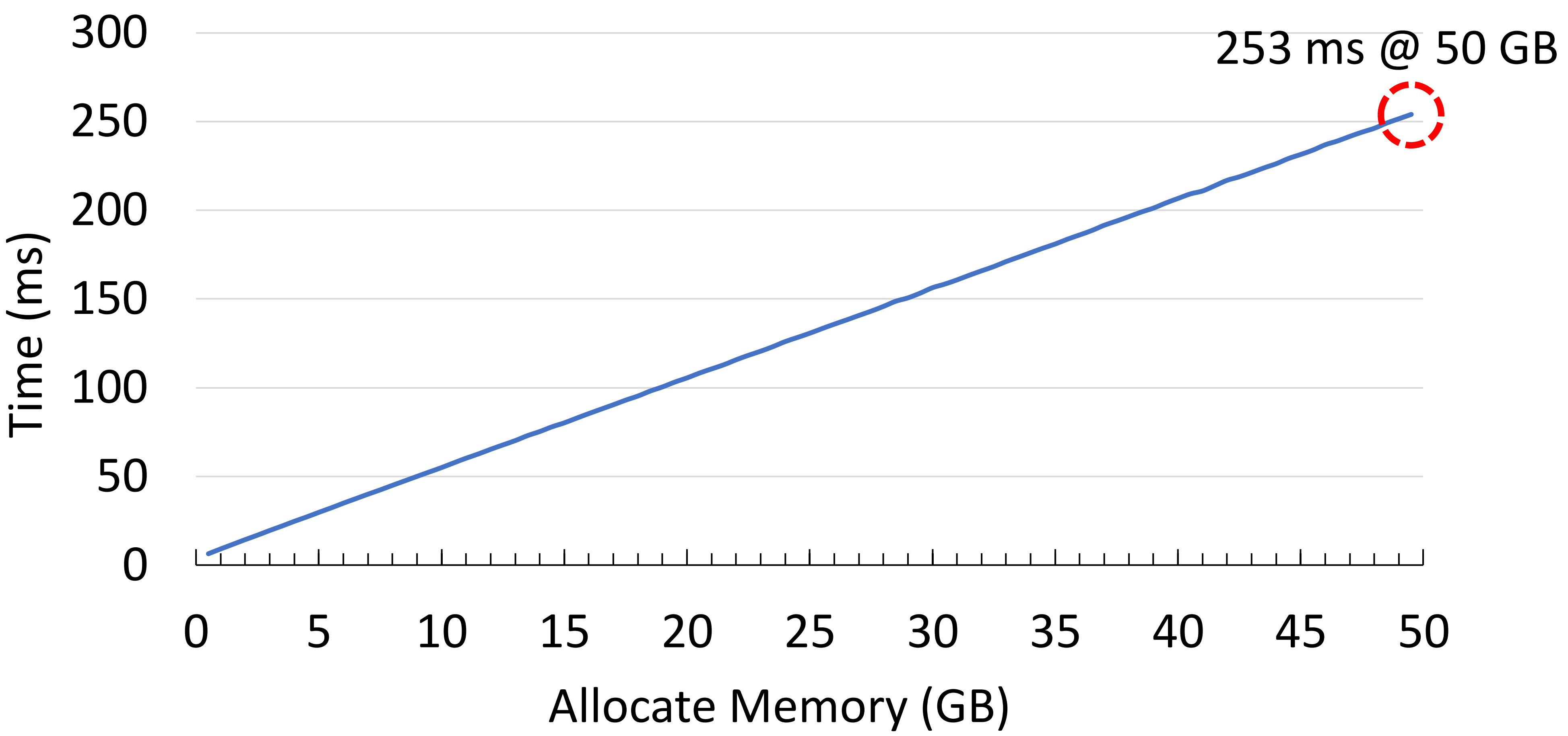


Fork has a latency problem



Fork gets slower as memory gets larger

Fork has a latency problem



Fork gets slower as memory gets larger

A slow fork is bad

Code for snapshotting

A slow fork is bad

Code for snapshotting

handle_request()

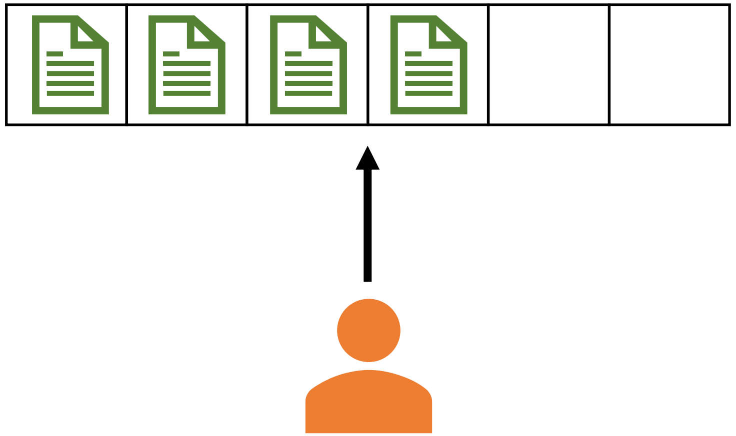
↓

fork()

↓

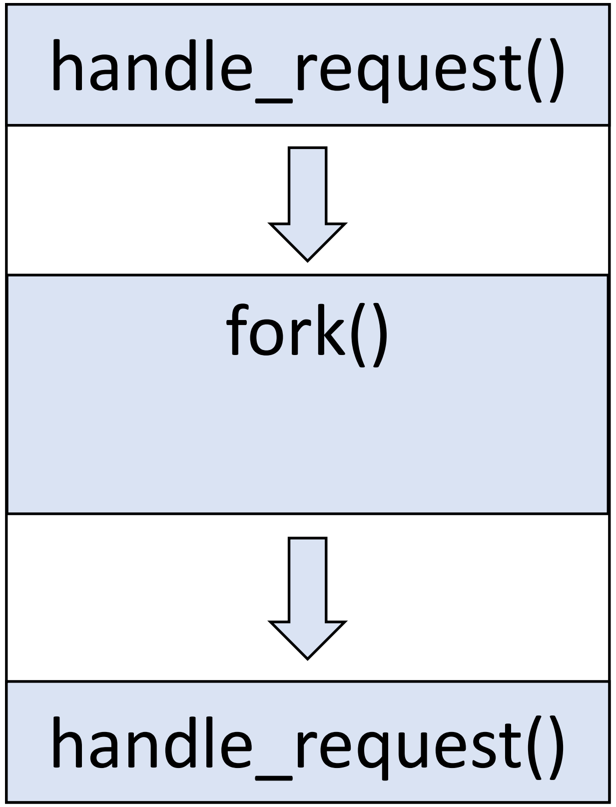
handle_request()

Requests keep queueing

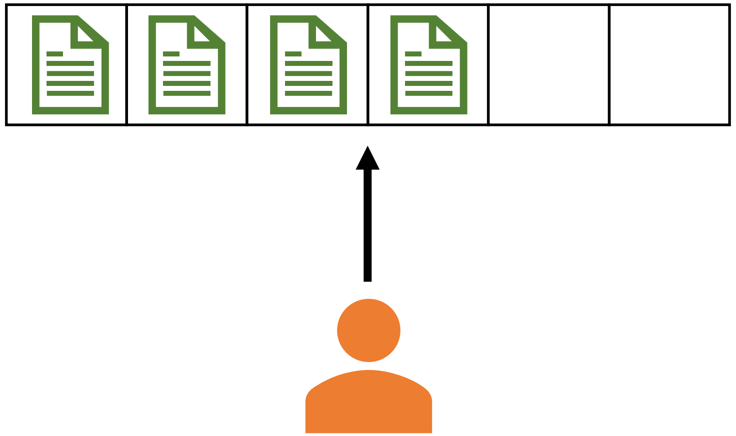


A slow fork is bad

Code for snapshotting



Requests keep queueing



Long latency of fork blocks applications on the critical path

Fork has an efficiency problem

Fork has an efficiency problem

- Fork sets up the entire address space of the child process

Fork has an efficiency problem

- Fork sets up the entire address space of the child process
- But some applications only access a small portion of the memory in the child process

Fork has an efficiency problem

- Fork sets up the entire address space of the child process
- But some applications only access a small portion of the memory in the child process
- E.g., when an application is being fuzzed

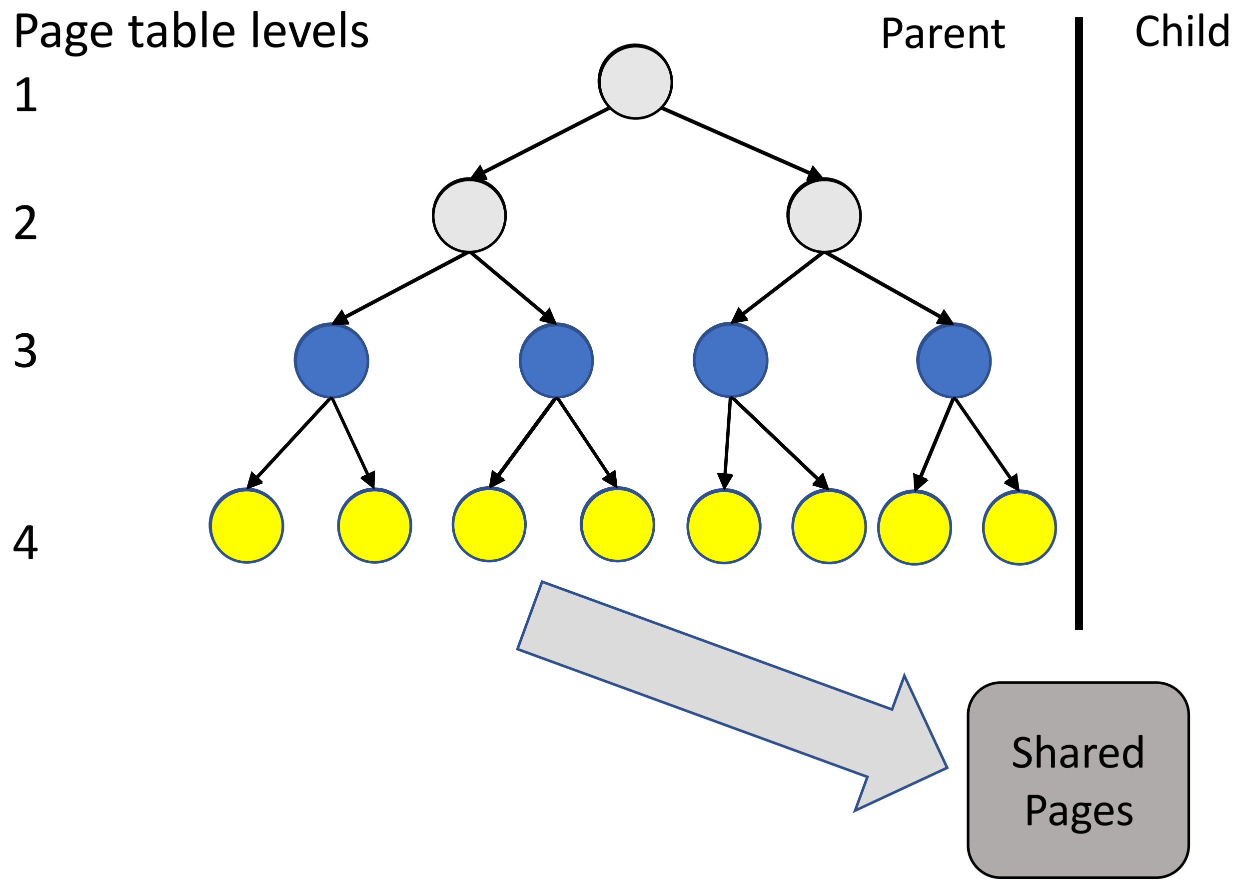
Fork has an efficiency problem

- Fork sets up the entire address space of the child process
- But some applications only access a small portion of the memory in the child process
- E.g., when an application is being fuzzed

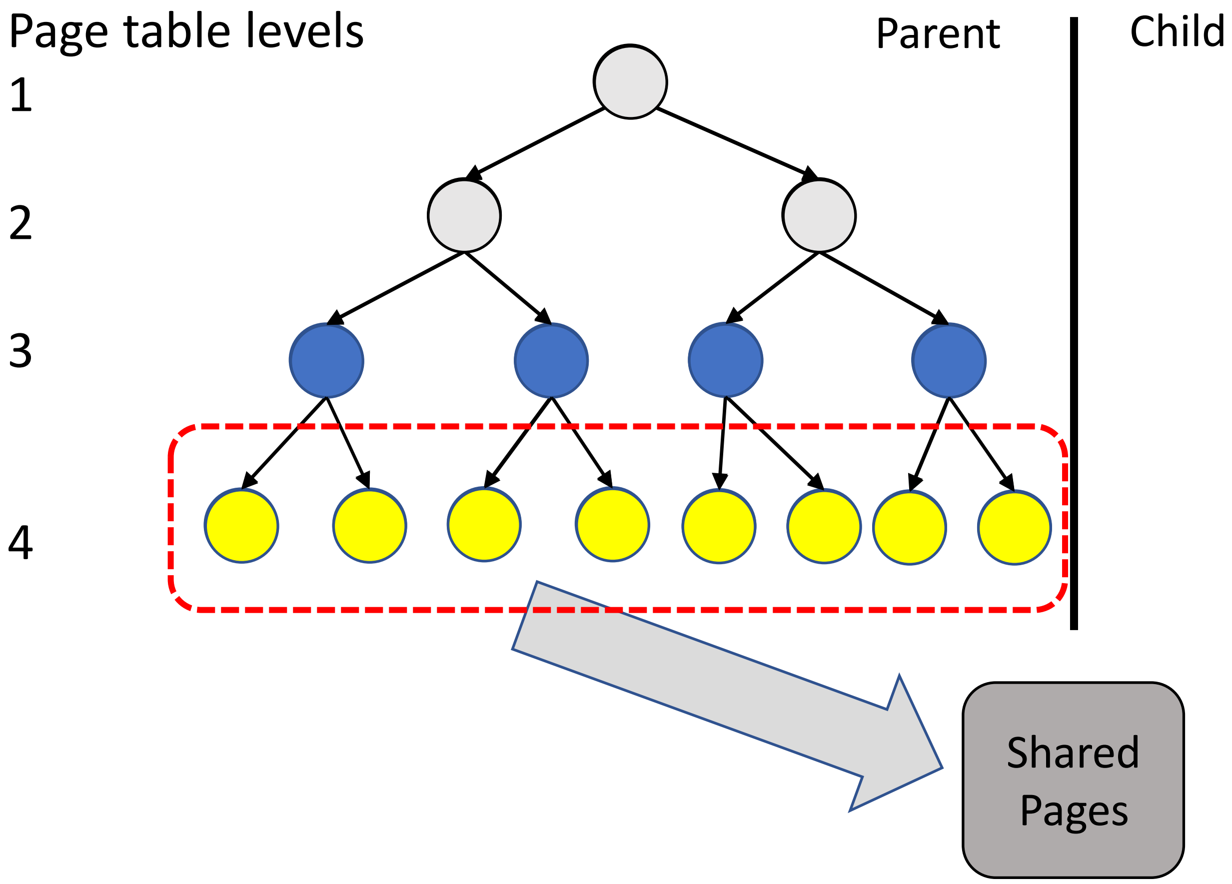
Setting up the whole address space is wasteful

Why is fork **slow** and **inefficient**?

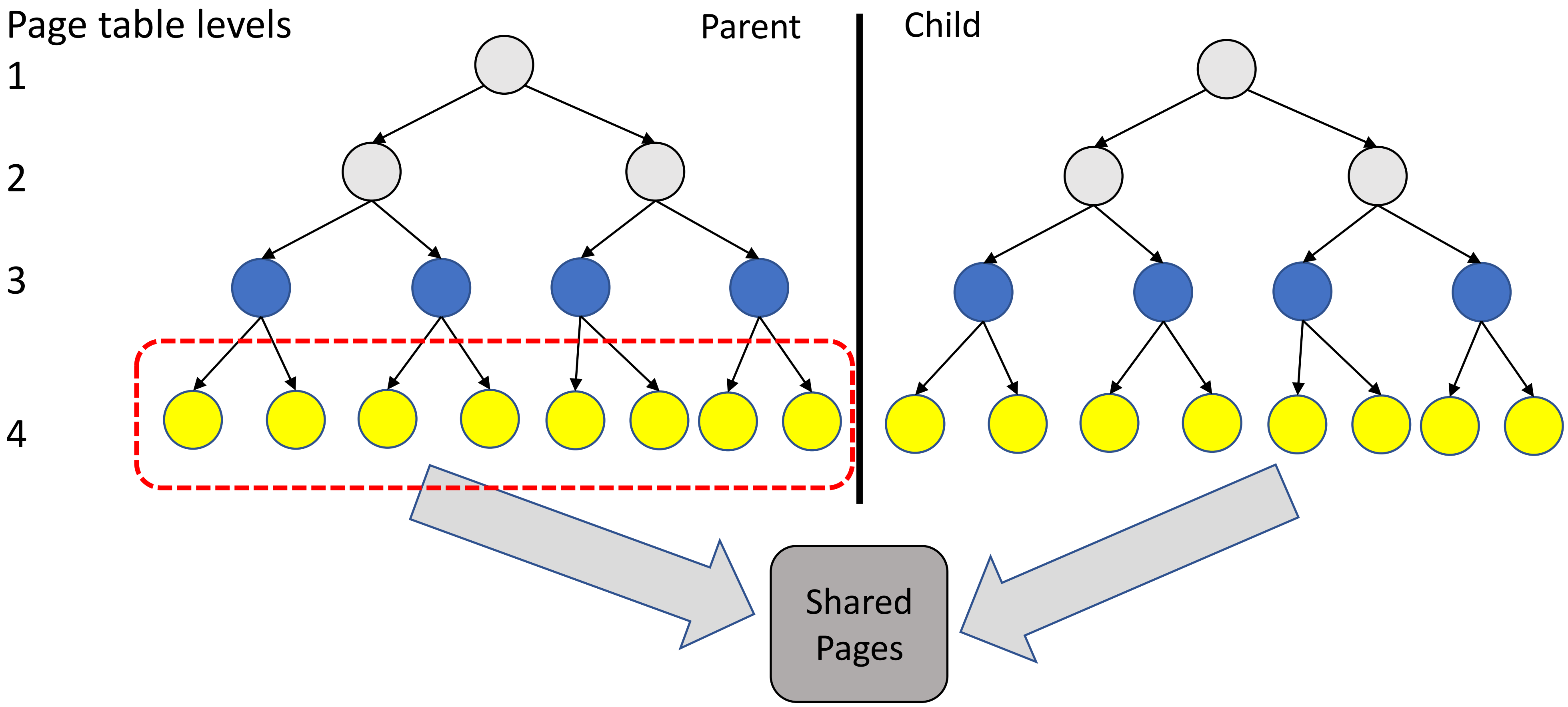
Fork copies page tables



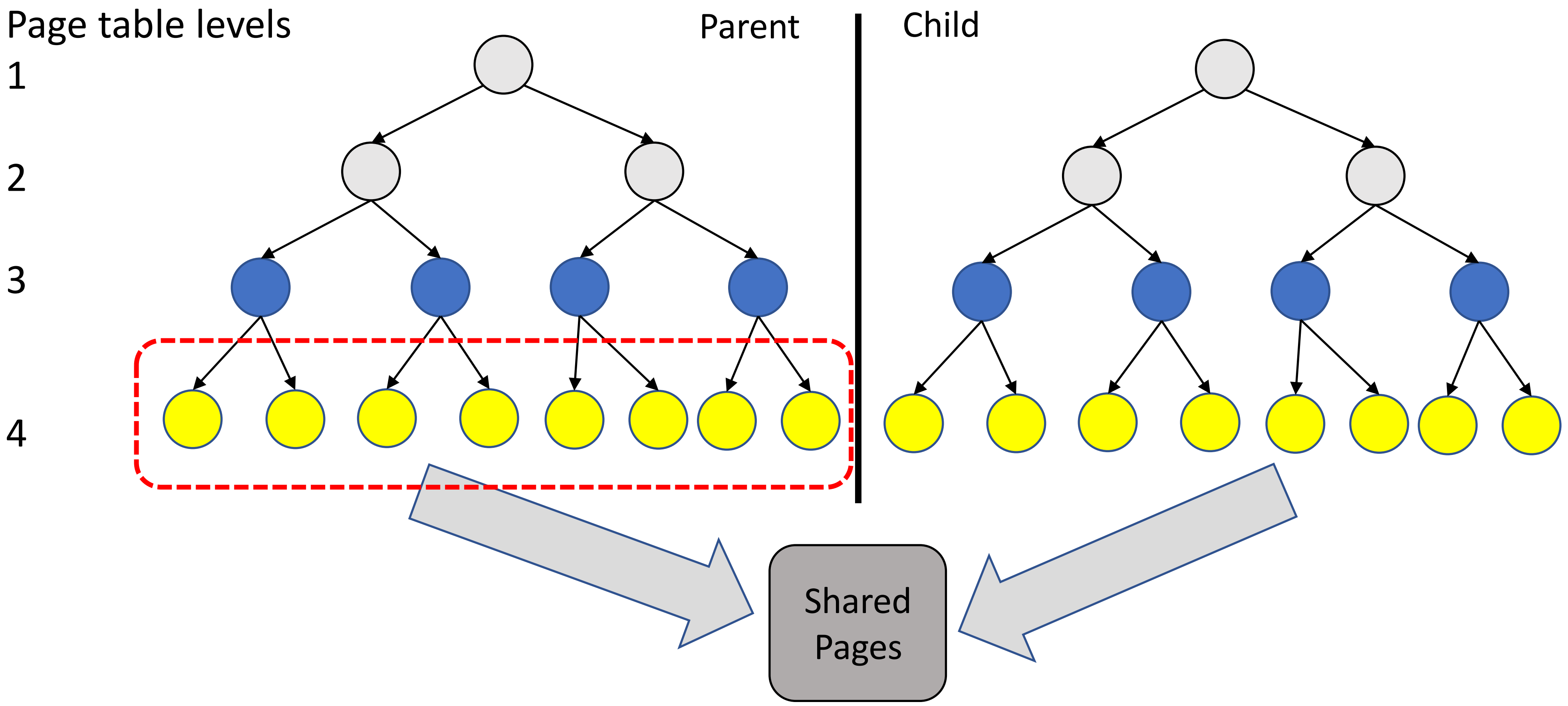
Fork copies page tables



Fork copies page tables

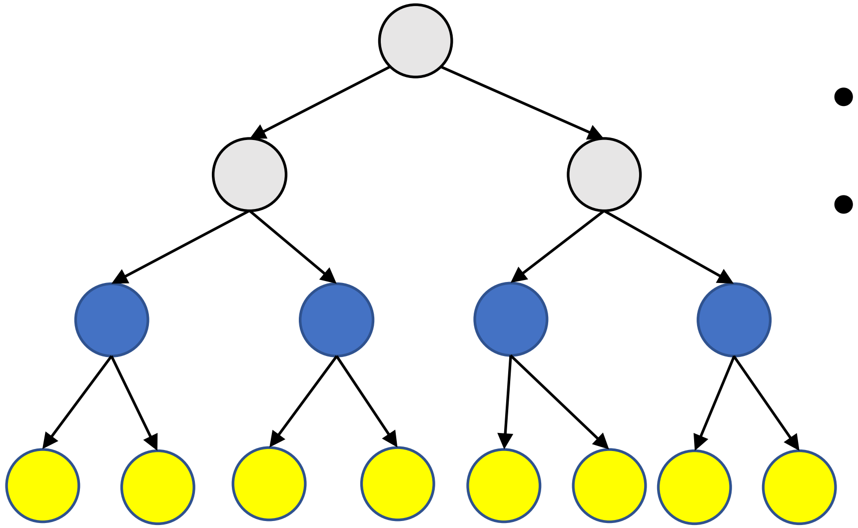


Fork copies page tables



Copying is prohibitively expensive for large applications

Huge pages are not a good solution



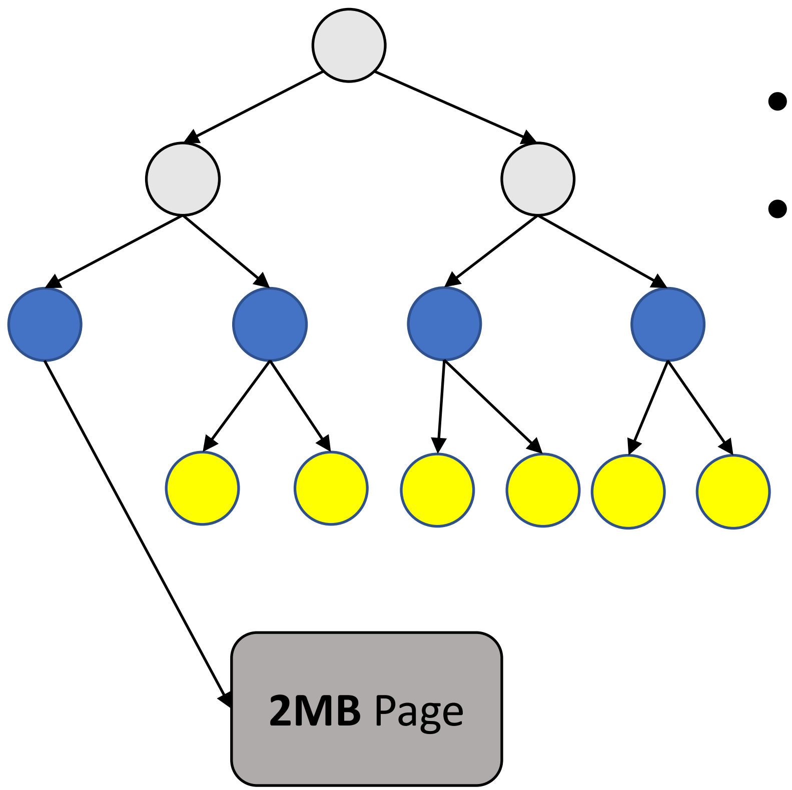
- Fewer pages mean fewer page tables to copy
- Lower the latency of fork, but suffer from:

Increased internal fragmentation

Expensive page faults

System-wide latency spike

Huge pages are not a good solution



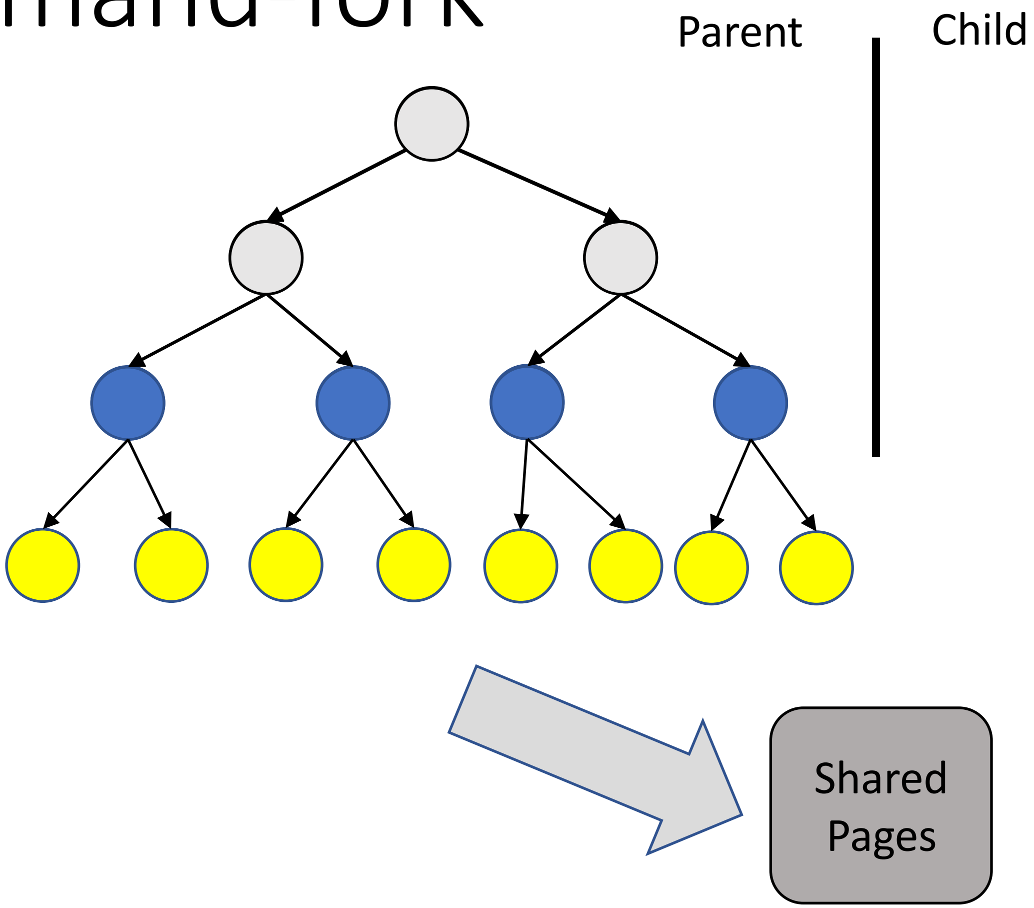
- Fewer pages mean fewer page tables to copy
- Lower the latency of fork, but suffer from:

Increased internal fragmentation

Expensive page faults

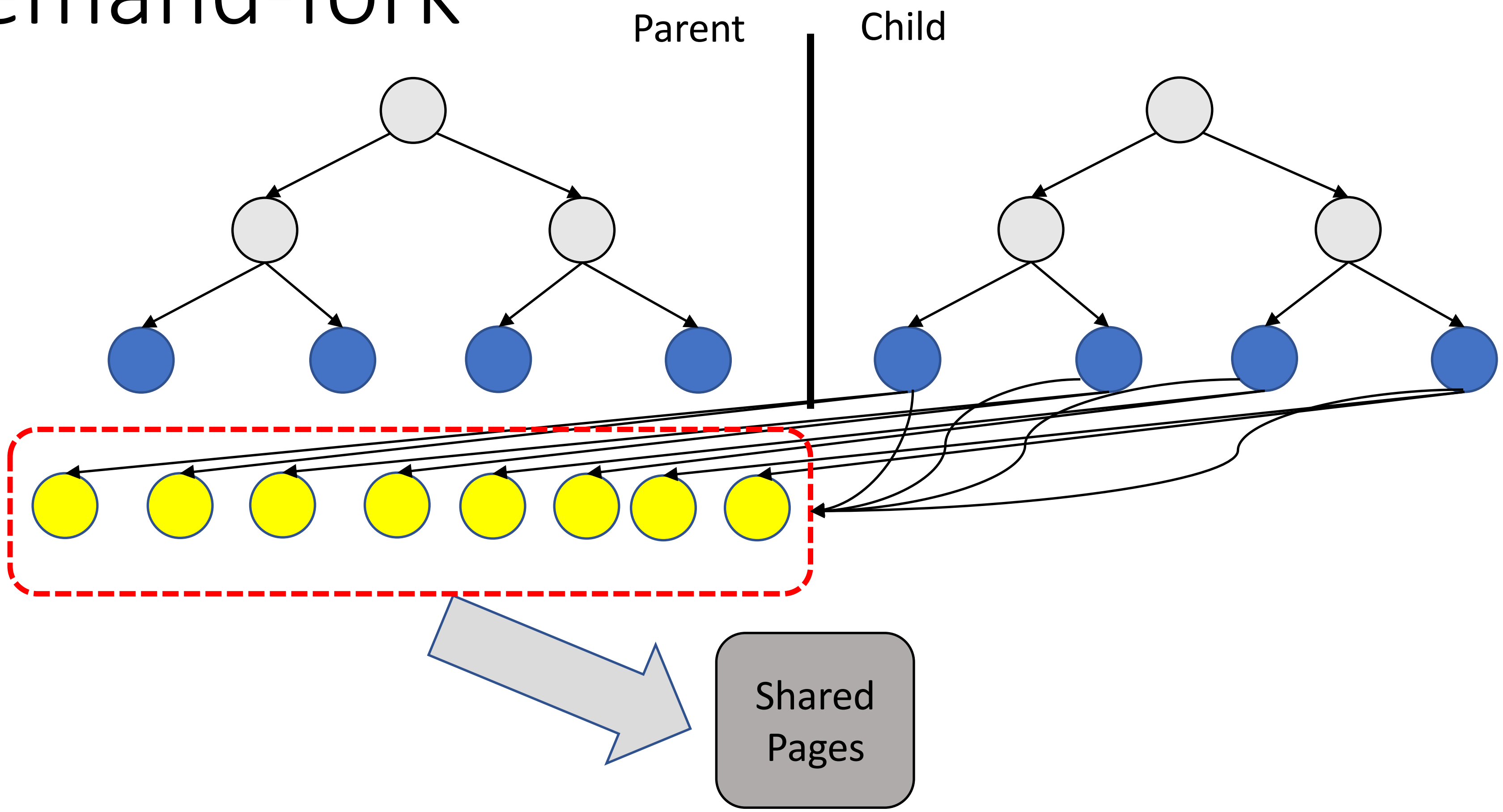
System-wide latency spike

On-demand-fork



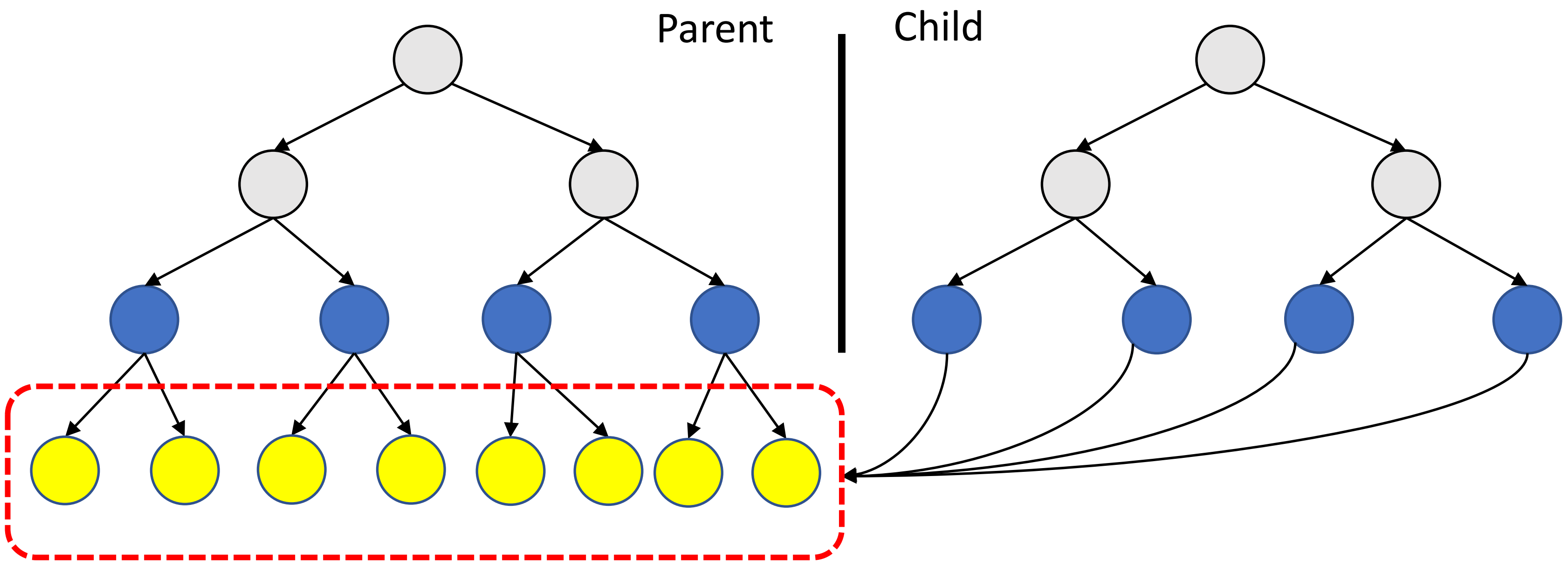
- Shares last-level page tables during fork
- Ensures microsecond-level latency for dozens of GBs of memory
- No issues of huge pages

On-demand-fork

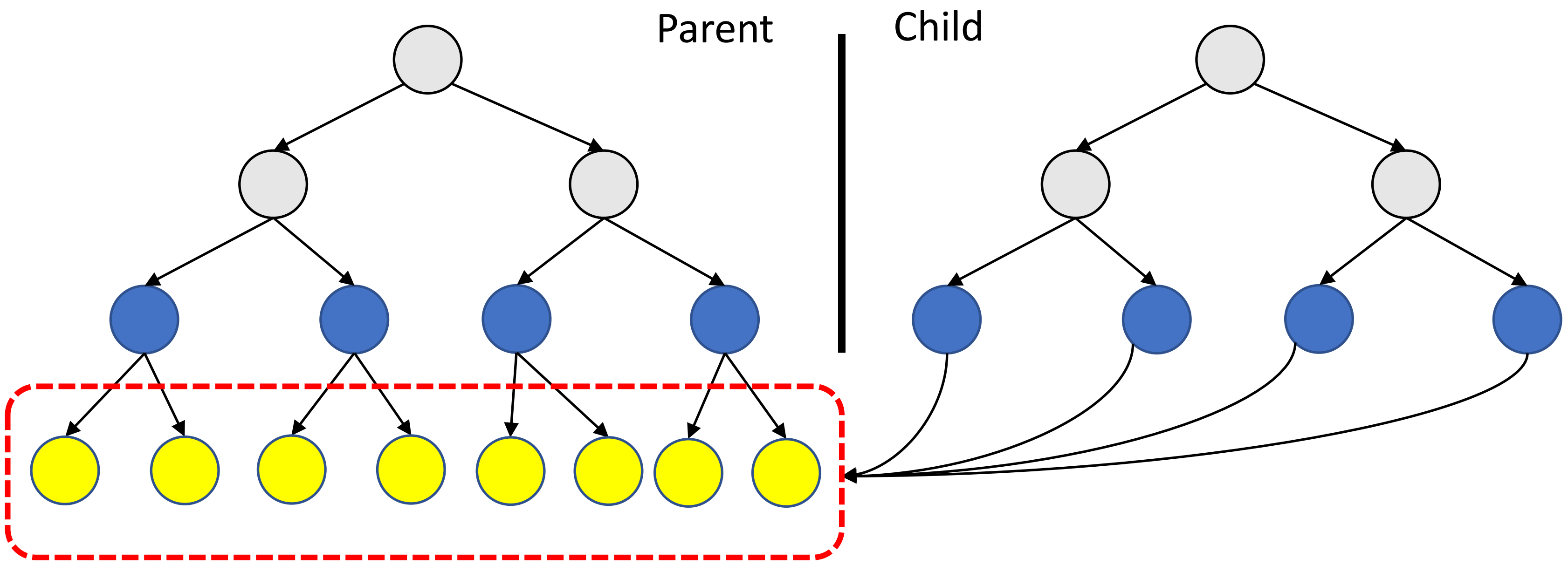


- Shares last-level page tables during fork
- Ensures microsecond-level latency for dozens of GBs of memory
- No issues of huge pages

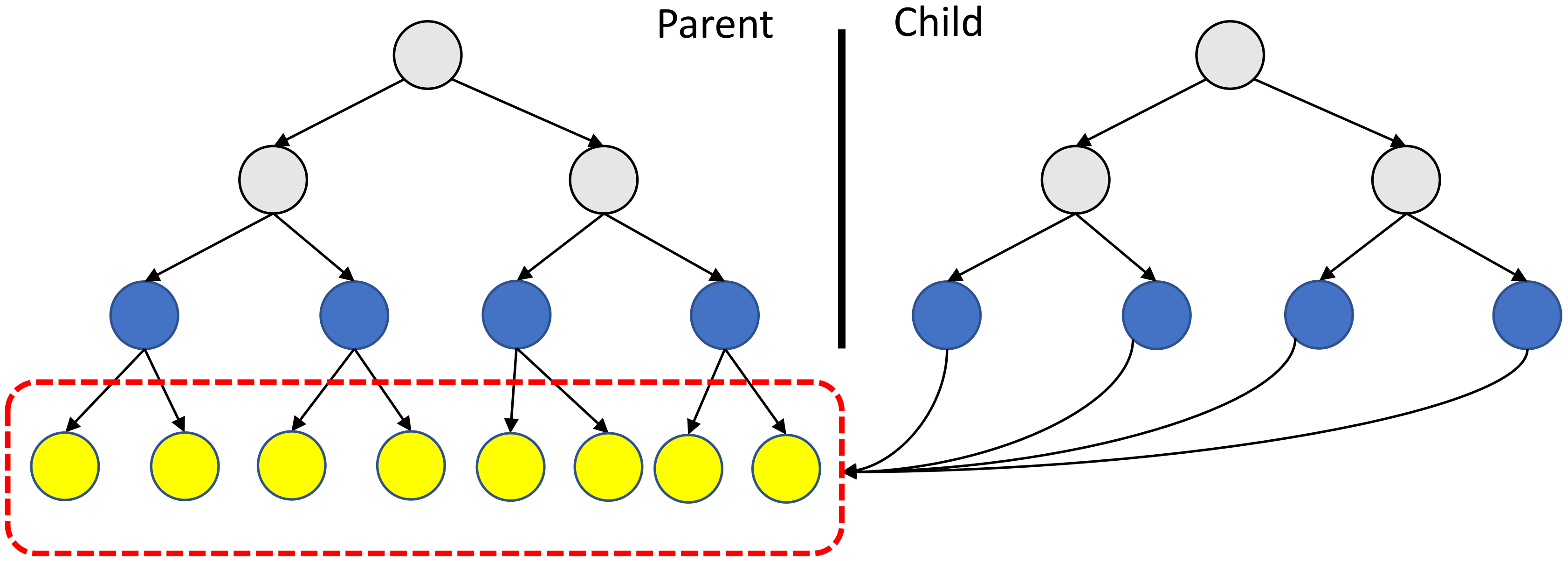
Fast read after fork



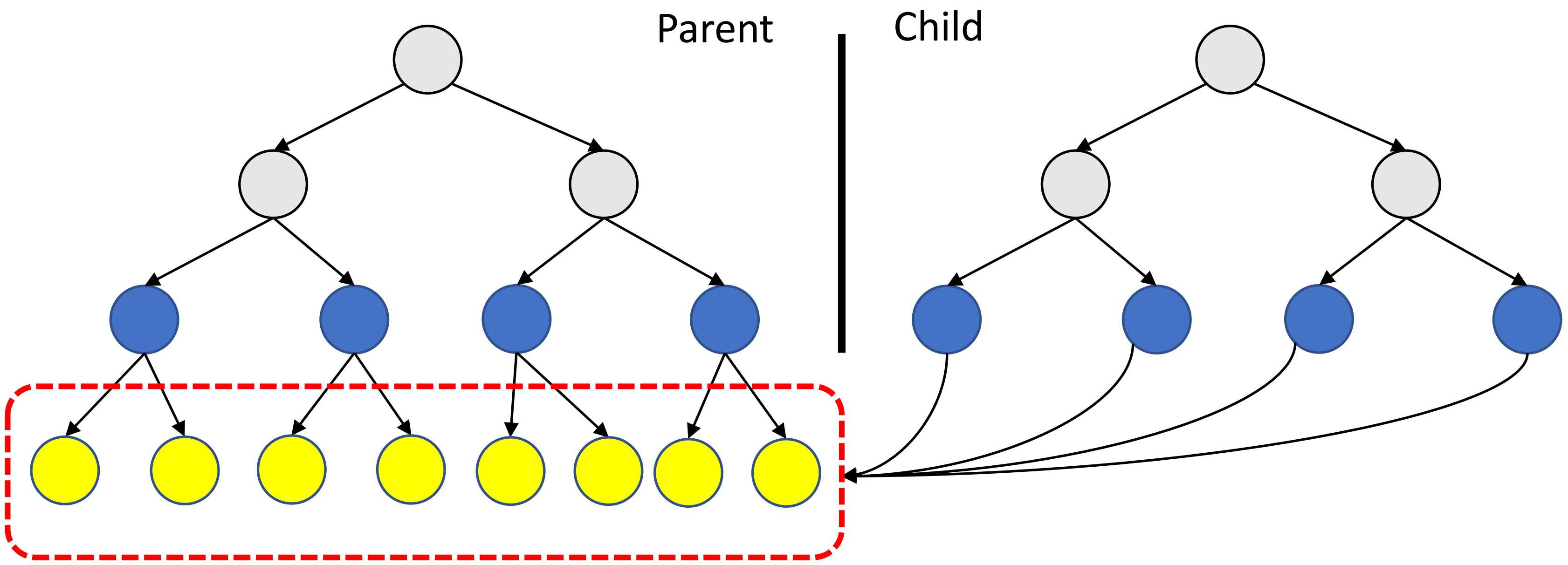
Fast read after fork



Fast read after fork

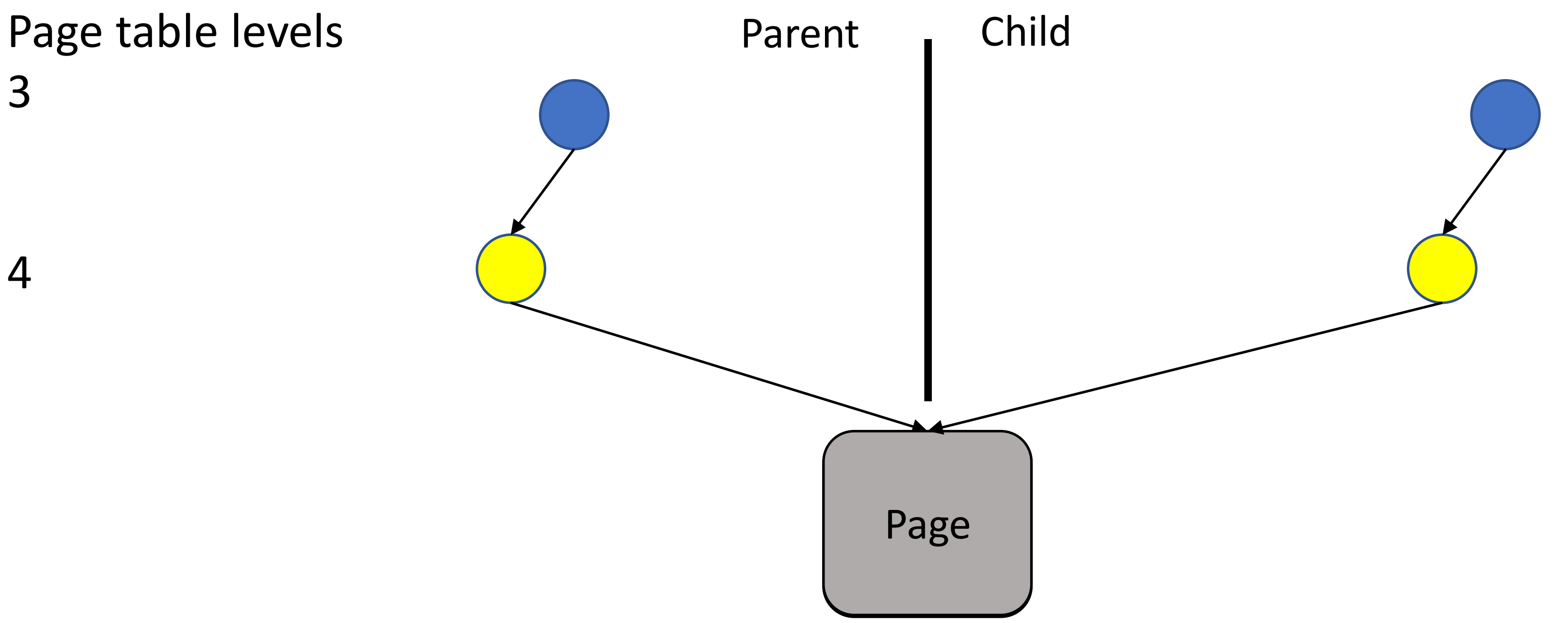


Fast read after fork



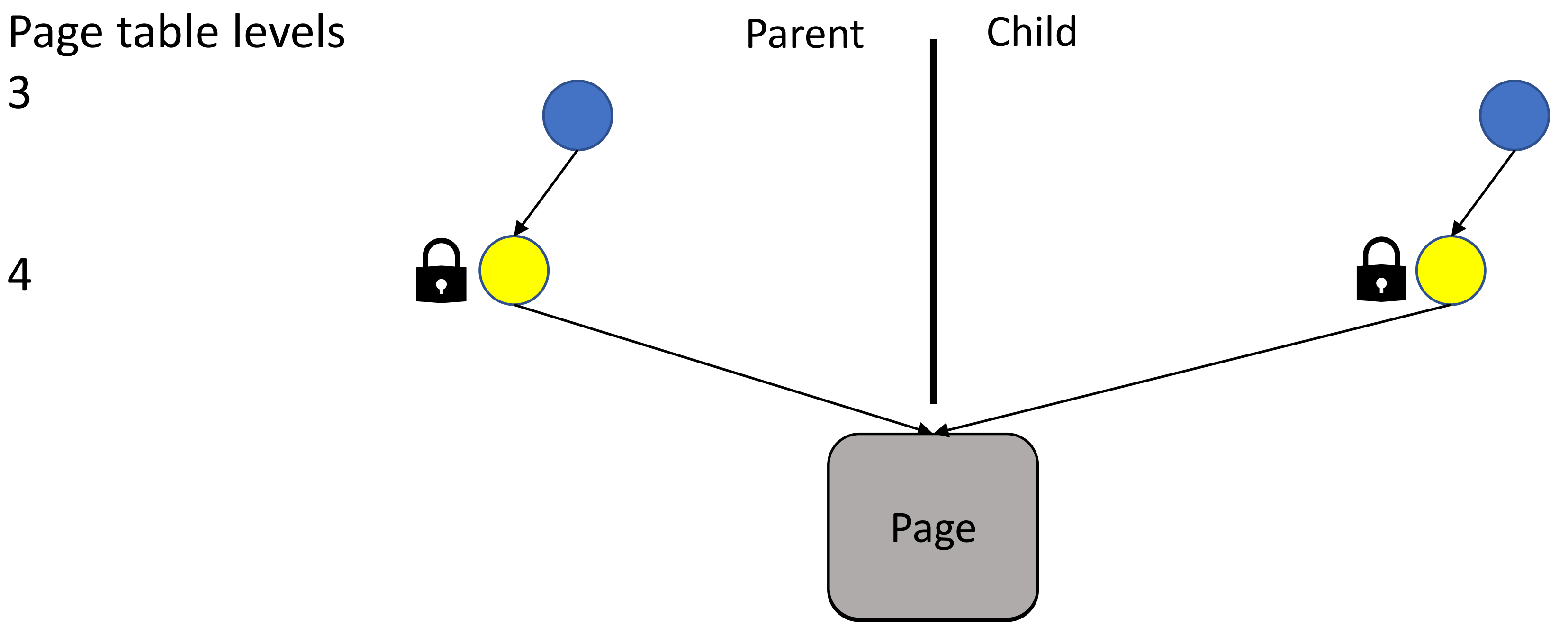
No cost of copying page tables for read access

Preserving copy-on-write semantics



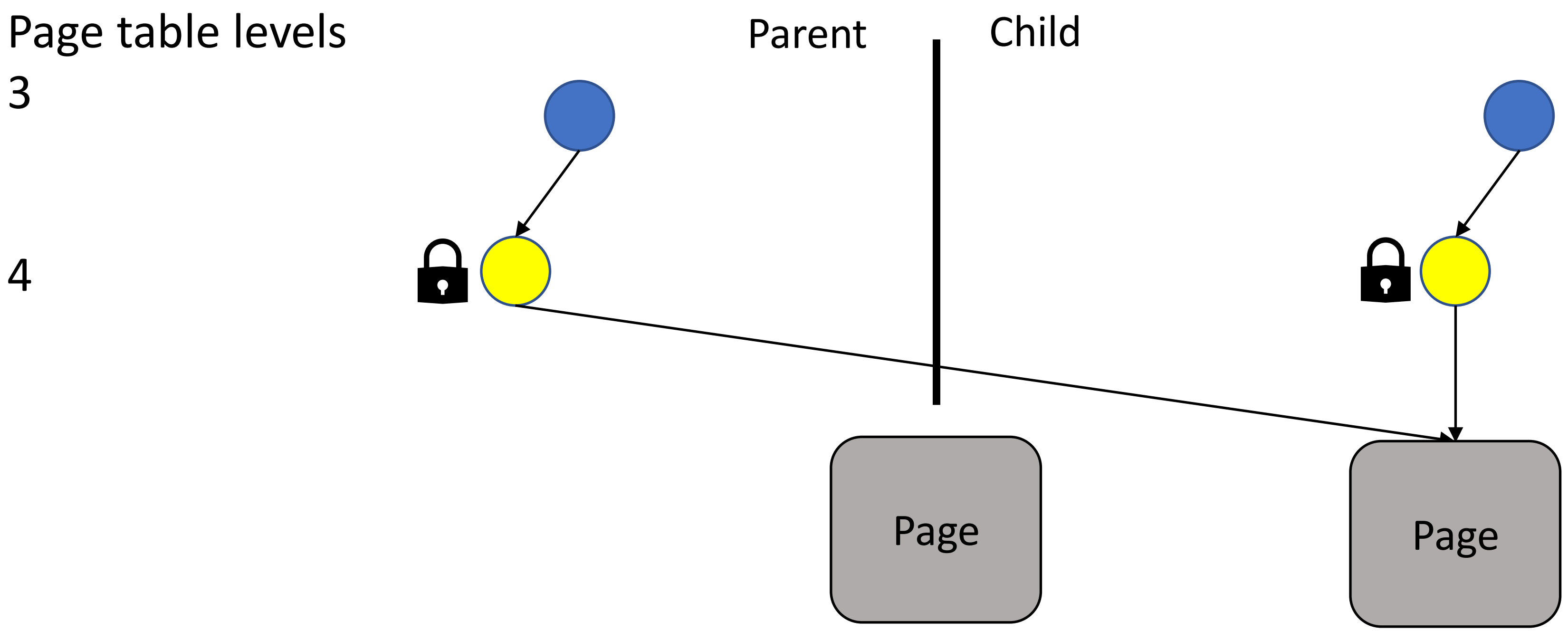
- The same view of the memory in the parent and child
- Traditional fork disables the write permission in **last-level** page tables
- Whoever writes gets a private copy of the physical page

Preserving copy-on-write semantics



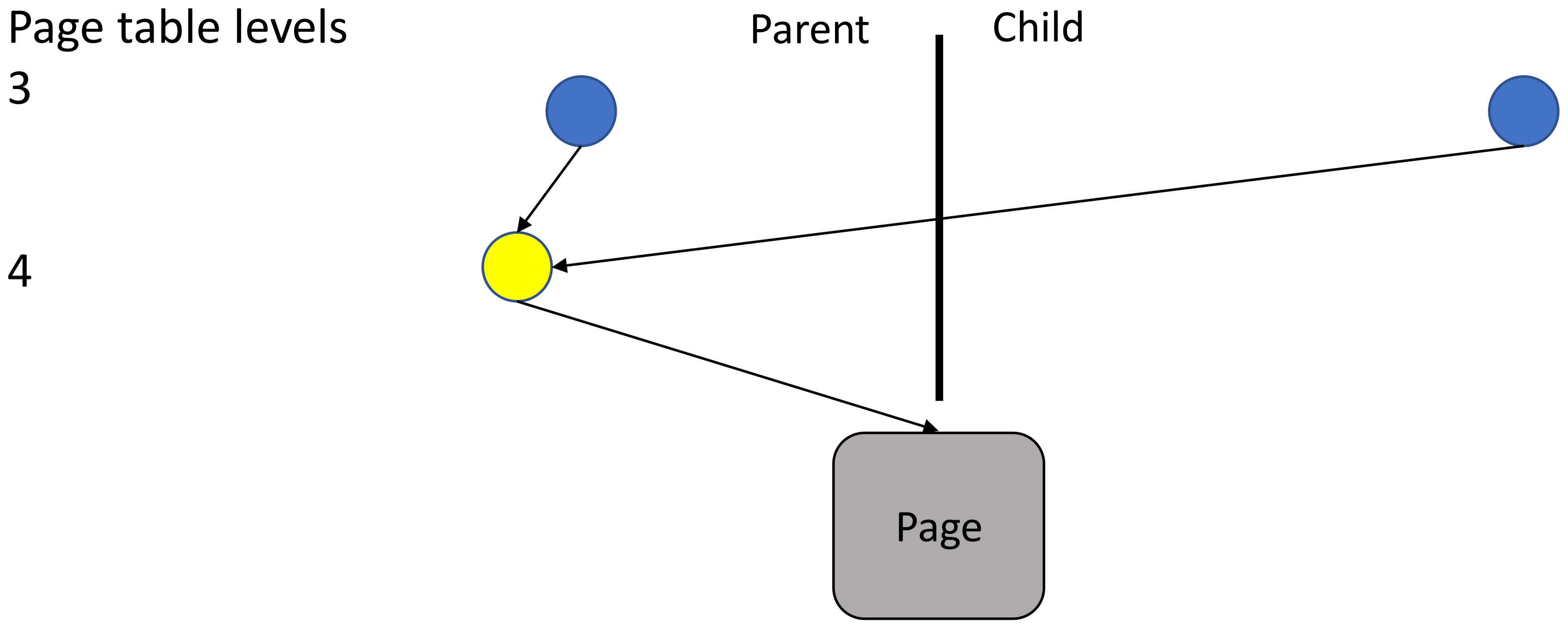
- The same view of the memory in the parent and child
- Traditional fork disables the write permission in **last-level** page tables
- Whoever writes gets a private copy of the physical page

Preserving copy-on-write semantics



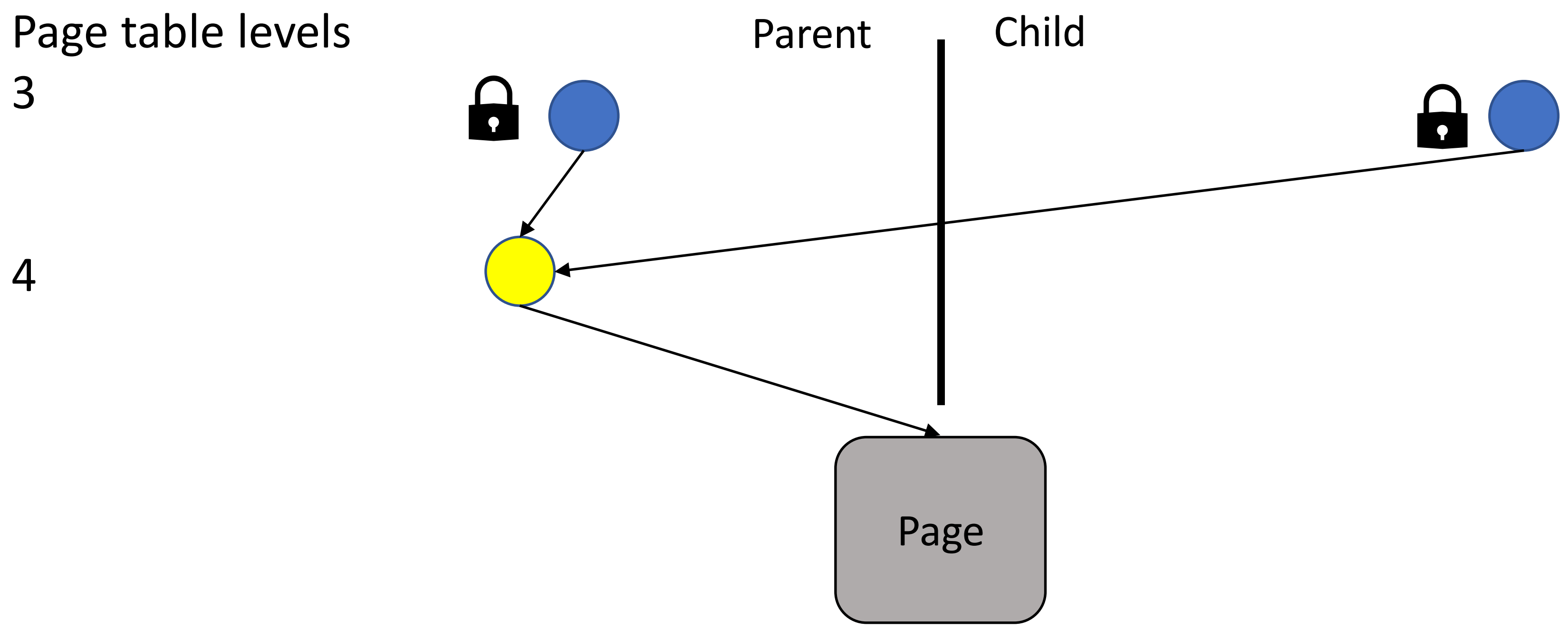
- The same view of the memory in the parent and child
- Traditional fork disables the write permission in **last-level** page tables
- Whoever writes gets a private copy of the physical page

Preserving copy-on-write semantics



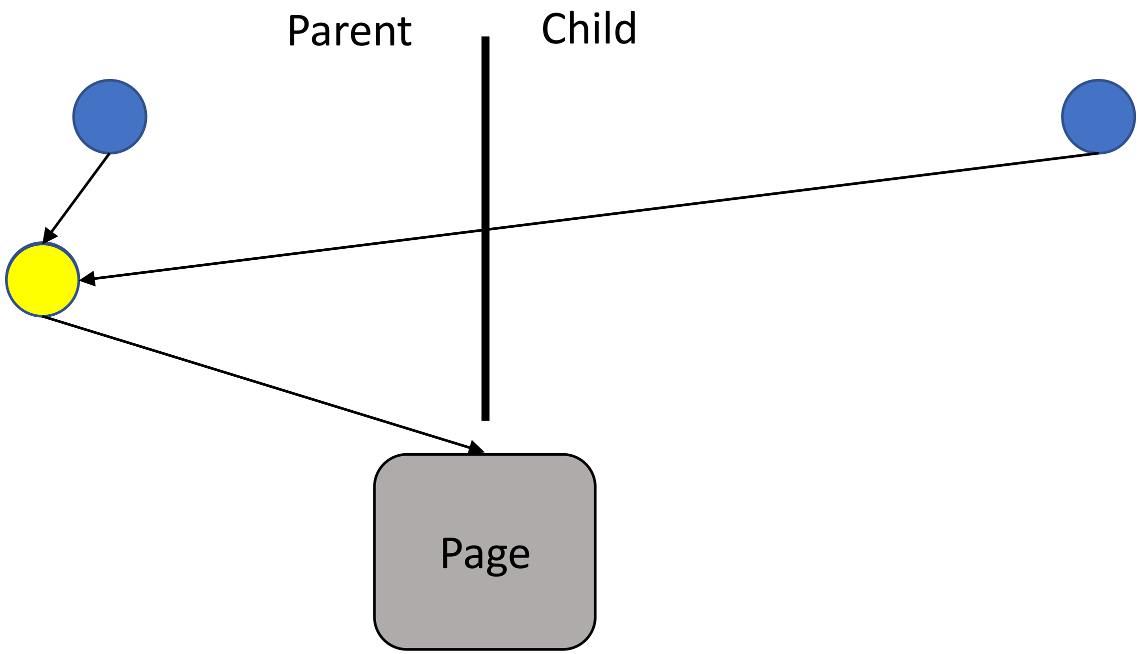
On-demand-fork disables the write permission in **3rd** level tables

Preserving copy-on-write semantics



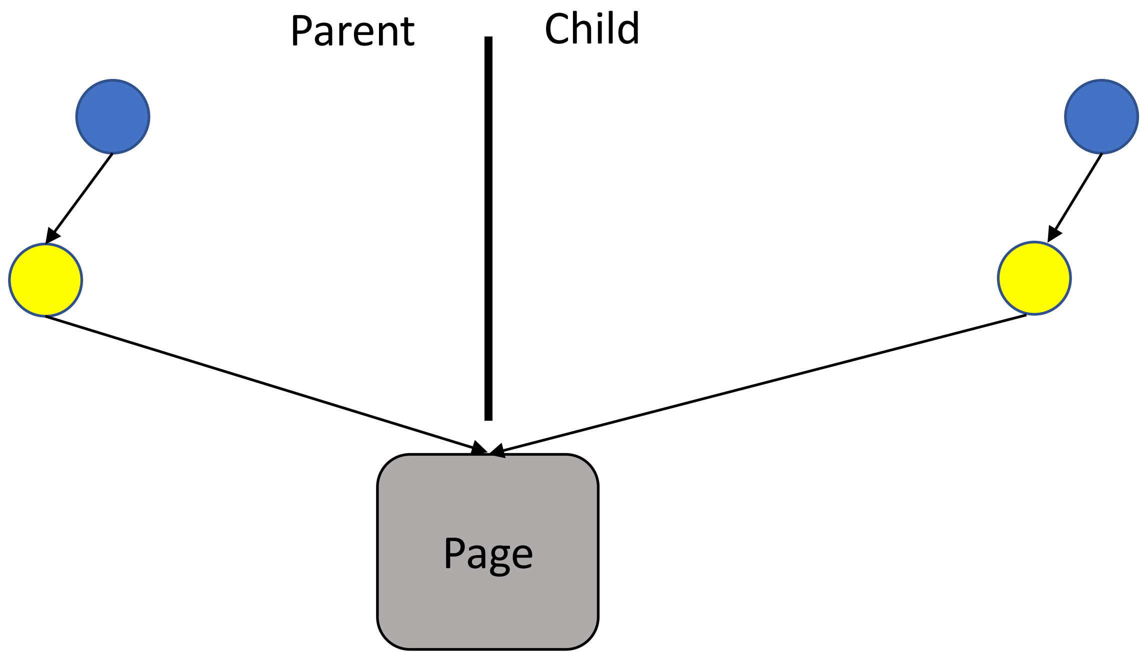
On-demand-fork disables the write permission in **3rd** level tables

On-demand page table copying



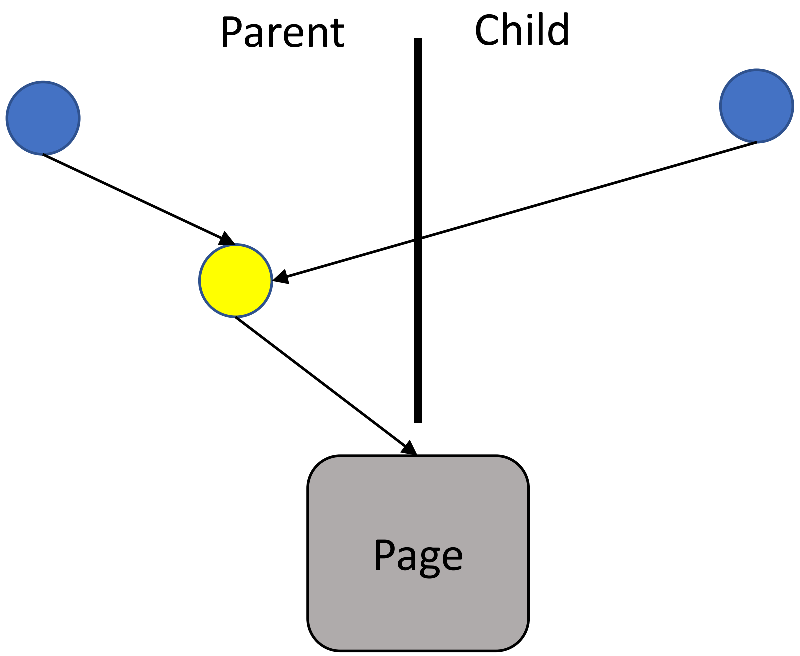
- Page faults for write access only
- Increased cost for only the first write access

On-demand page table copying



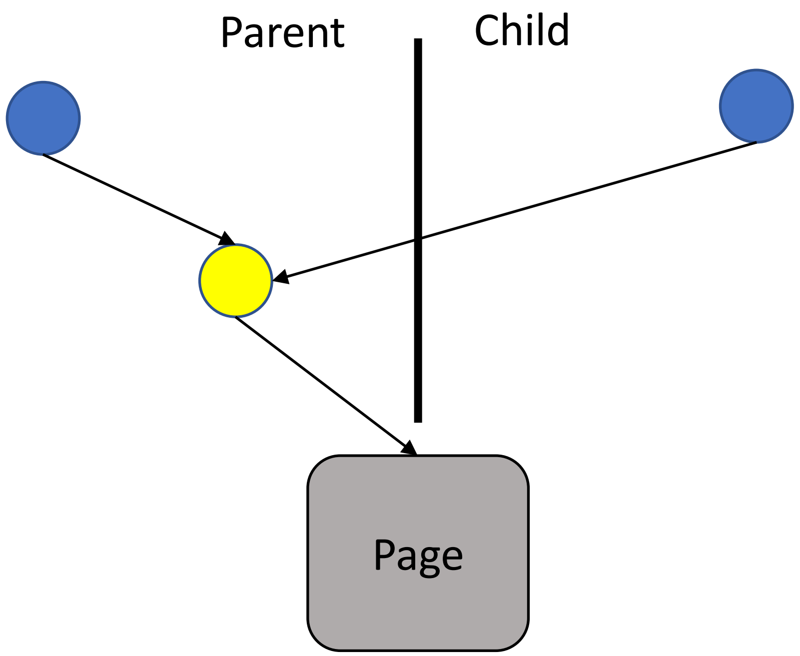
- Page faults for write access only
- Increased cost for only the first write access

Keeping track of shared tables



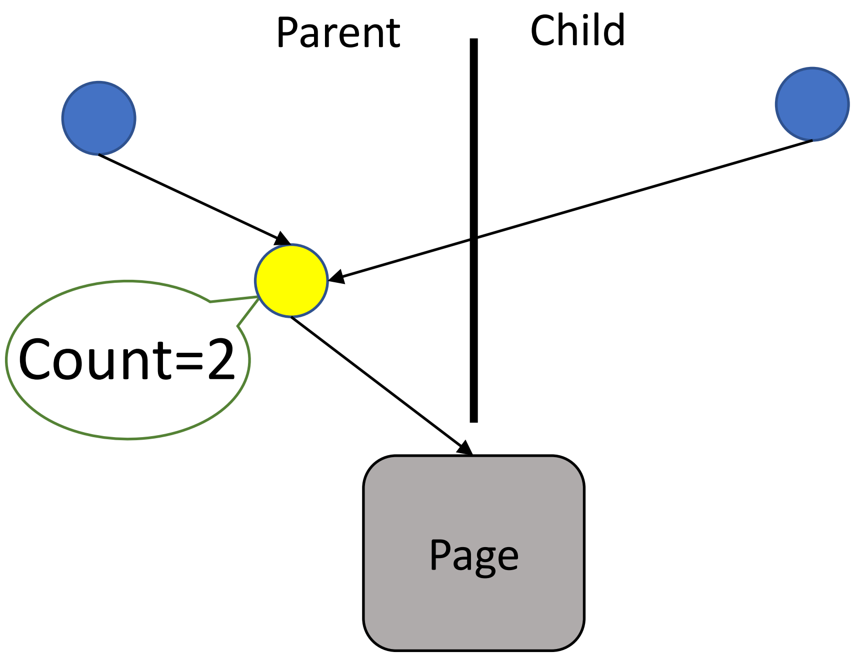
- **Challenge**: Need to know when to free last-level page tables

Keeping track of shared tables



- **Challenge**: Need to know when to free last-level page tables
- **Solution**: reference counts last-level page tables

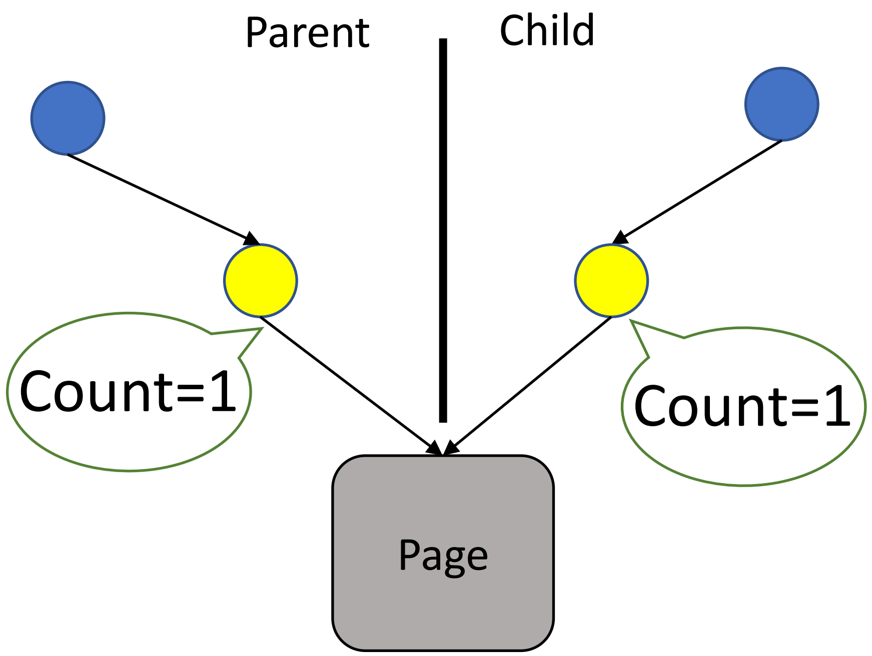
Keeping track of shared tables



During system call

- Reference counts last-level page tables
- Count equals the number of processes that share the page table

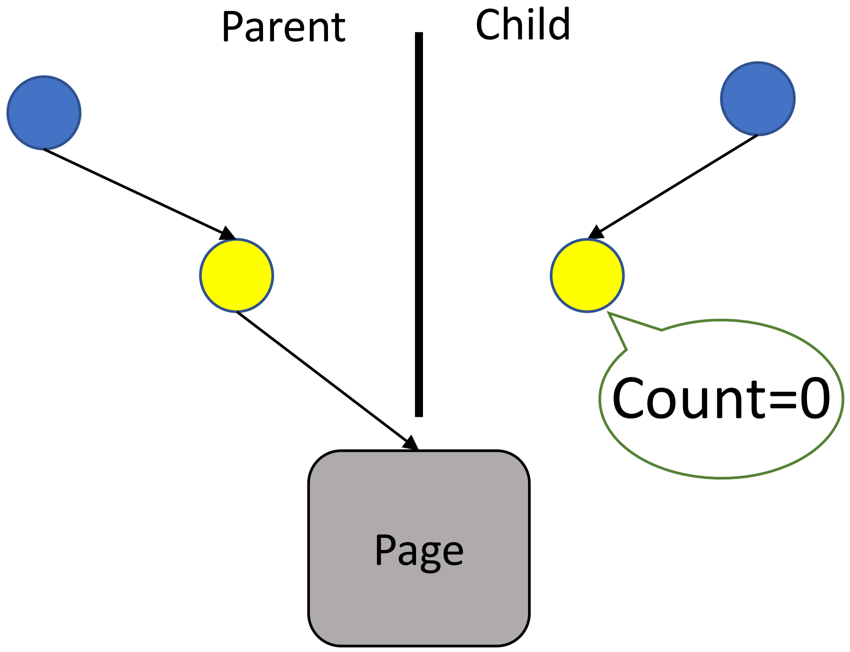
Keeping track of shared tables



On-demand copying

- Reference counts last-level page tables
- Count equals the number of processes that share the page table

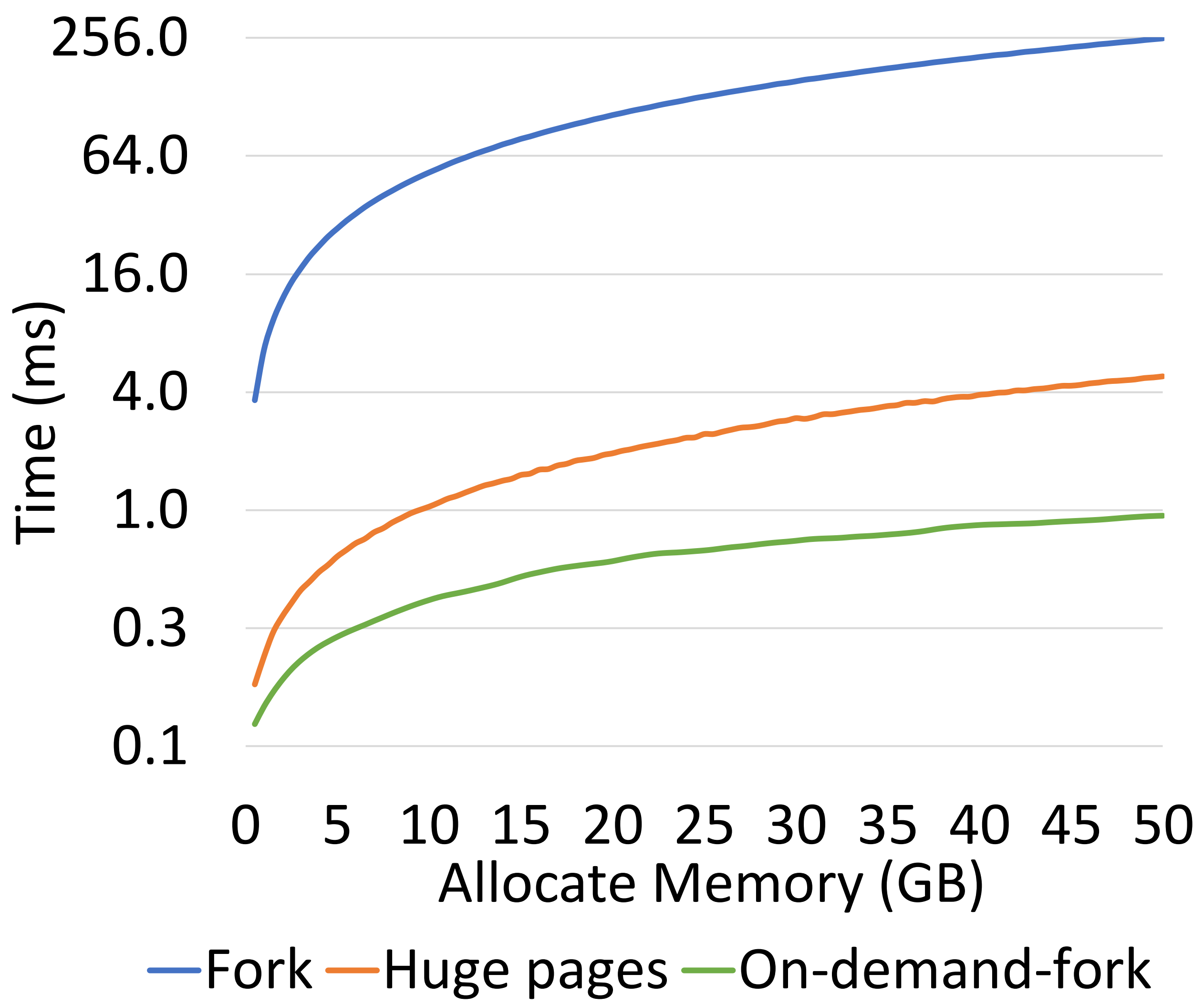
Keeping track of shared tables



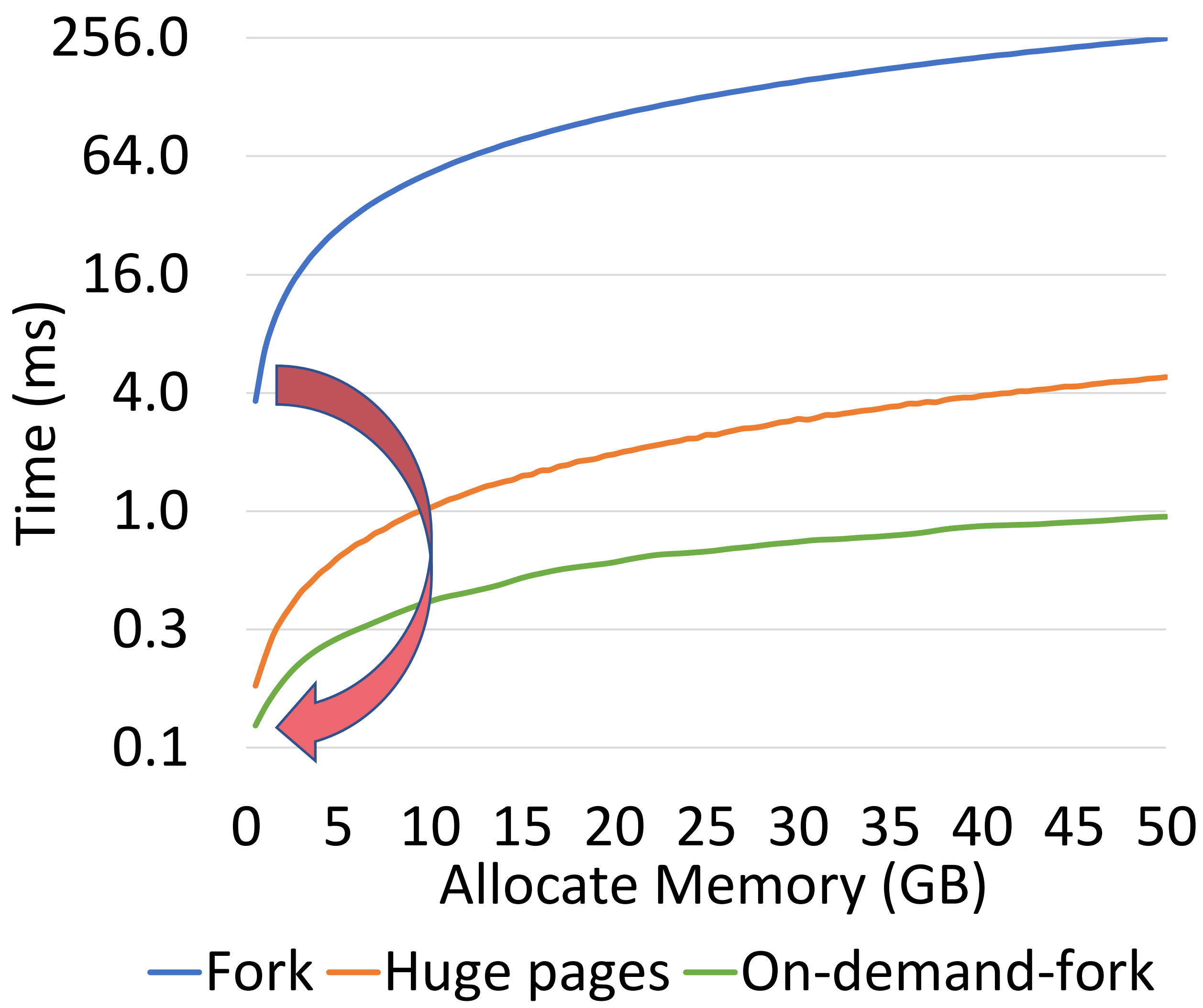
Unmapping

Last-level page tables are freed after count reaches zero

Microbenchmarks: system call latency

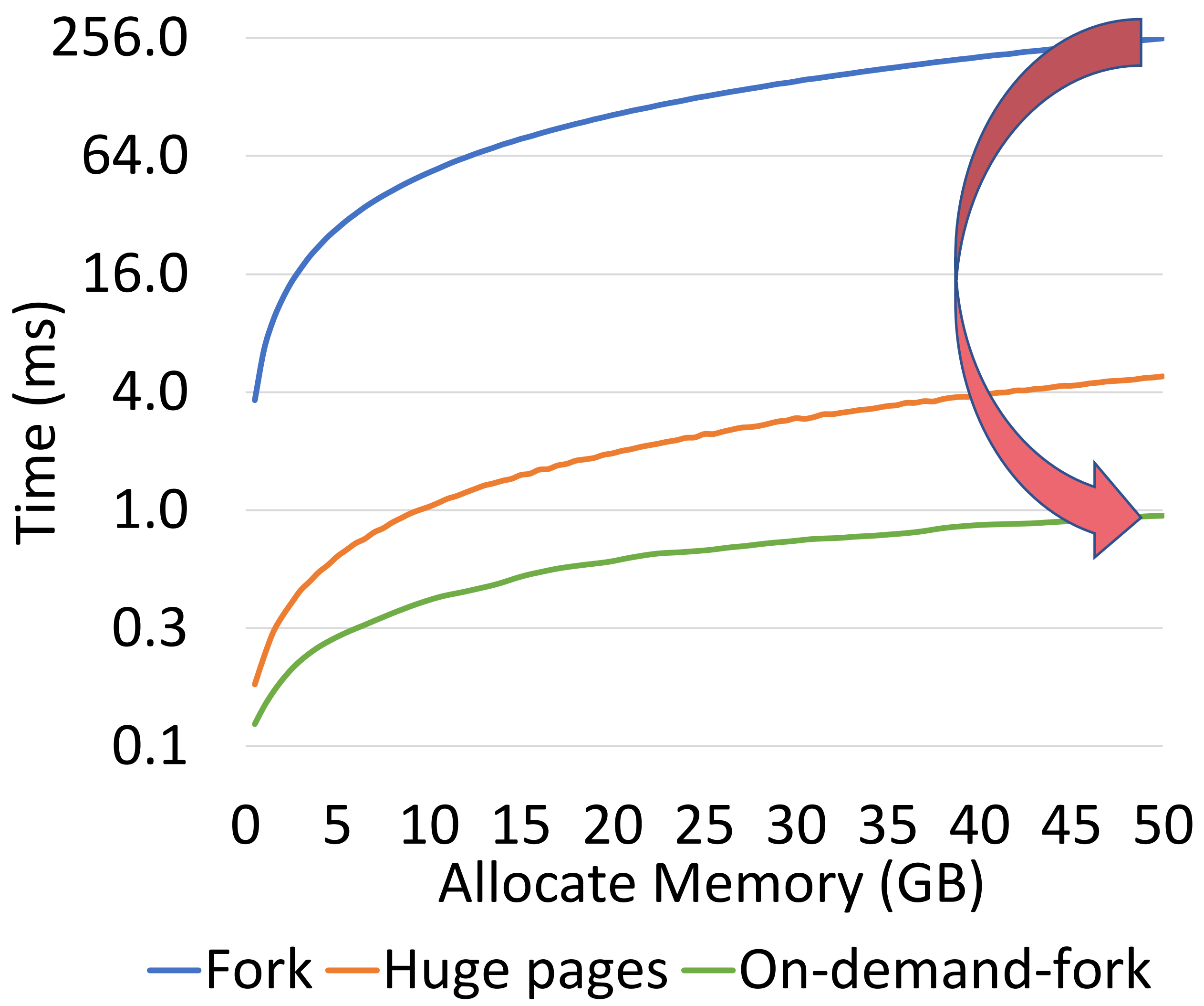


Microbenchmarks: system call latency



65 times faster at 1GB

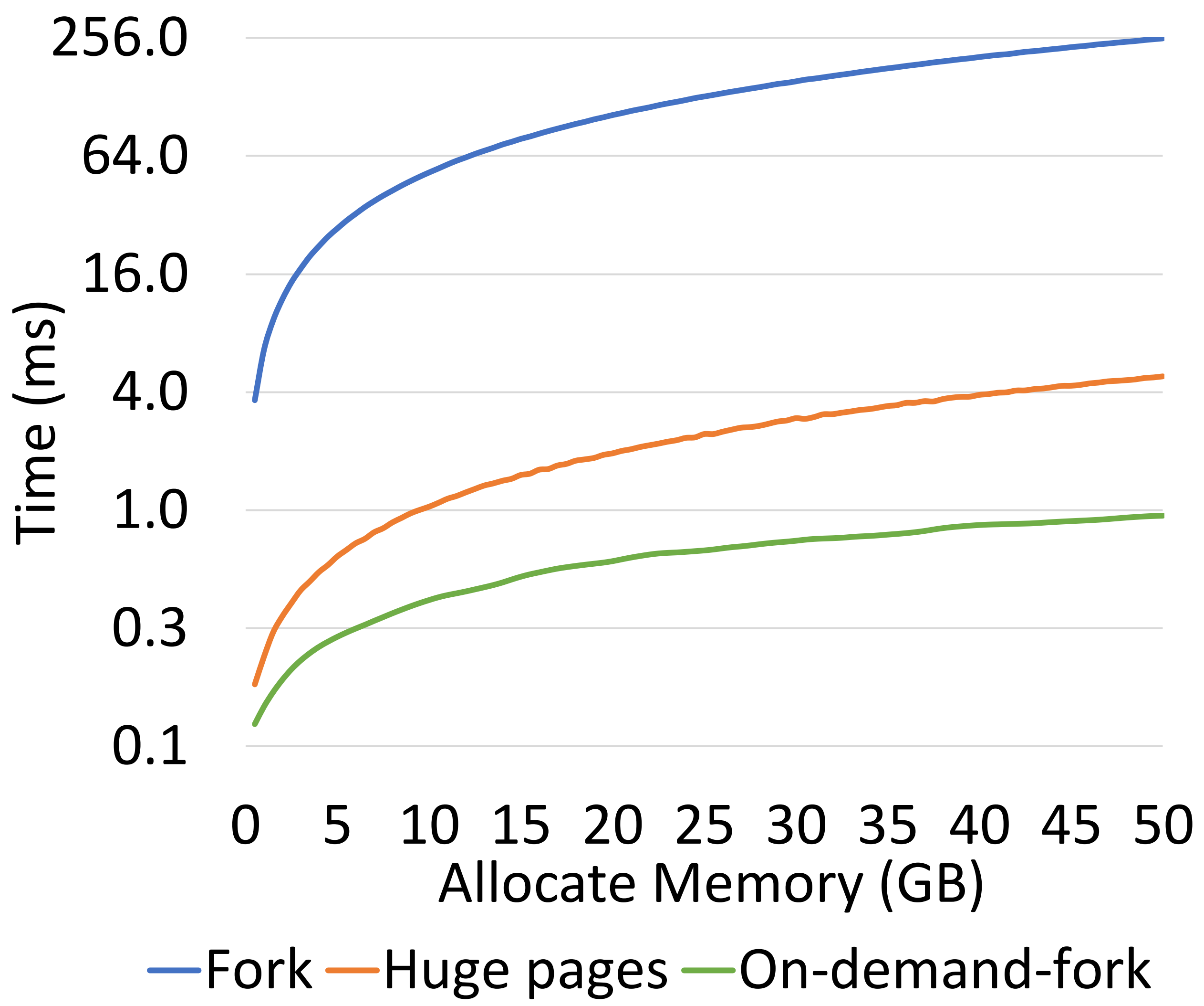
Microbenchmarks: system call latency



65 times faster at 1GB

270 times faster at 50GB

Microbenchmarks: system call latency

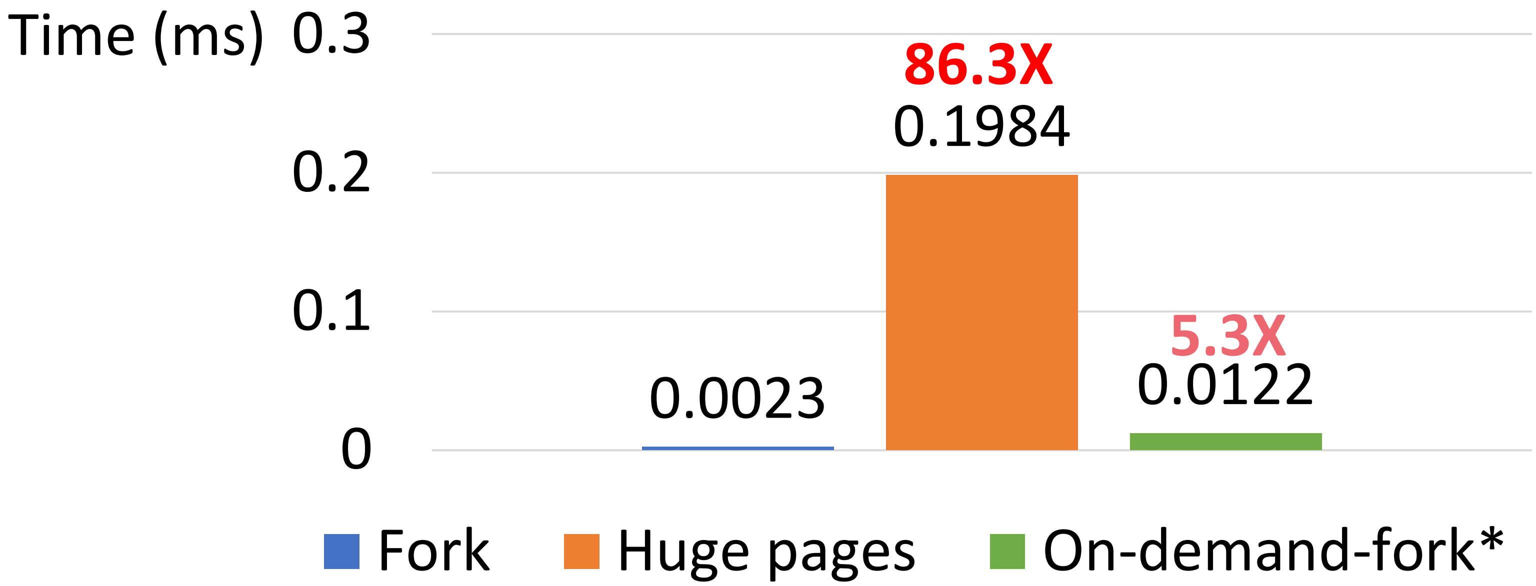


65 times faster at 1GB

270 times faster at 50GB

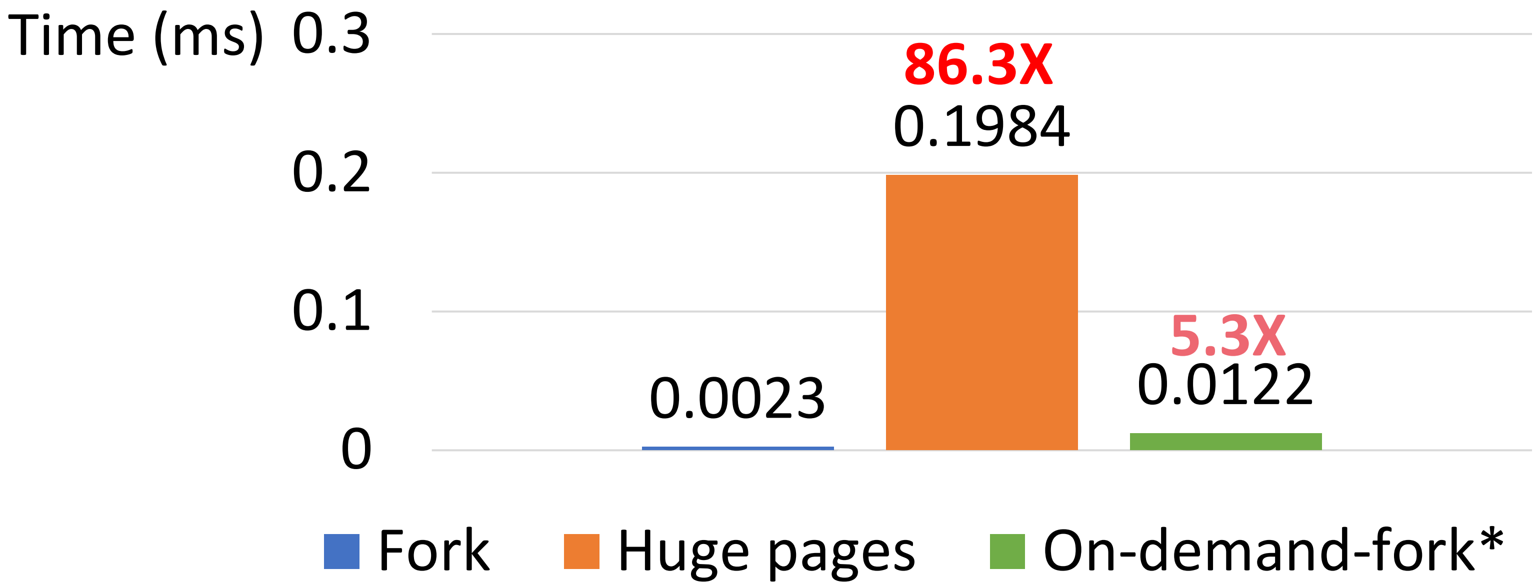
Faster than huge pages

Microbenchmarks: fault handling time



*: worst-case

Microbenchmarks: fault handling time



Worst case page fault handling time is reasonable

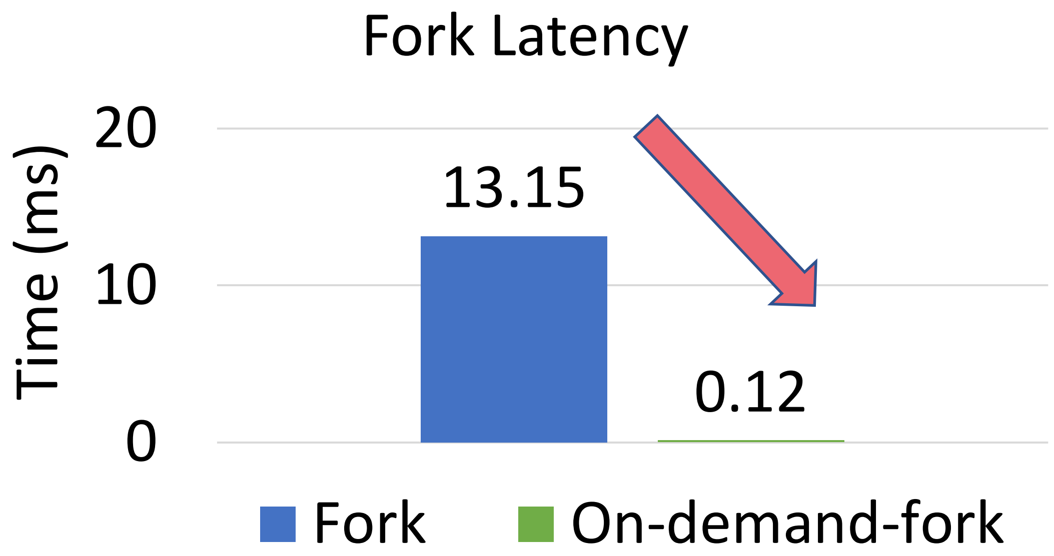
*: worst-case

Real-world applications: SQLite test suite

The test suite runs each test case in a child process

Real-world applications: SQLite test suite

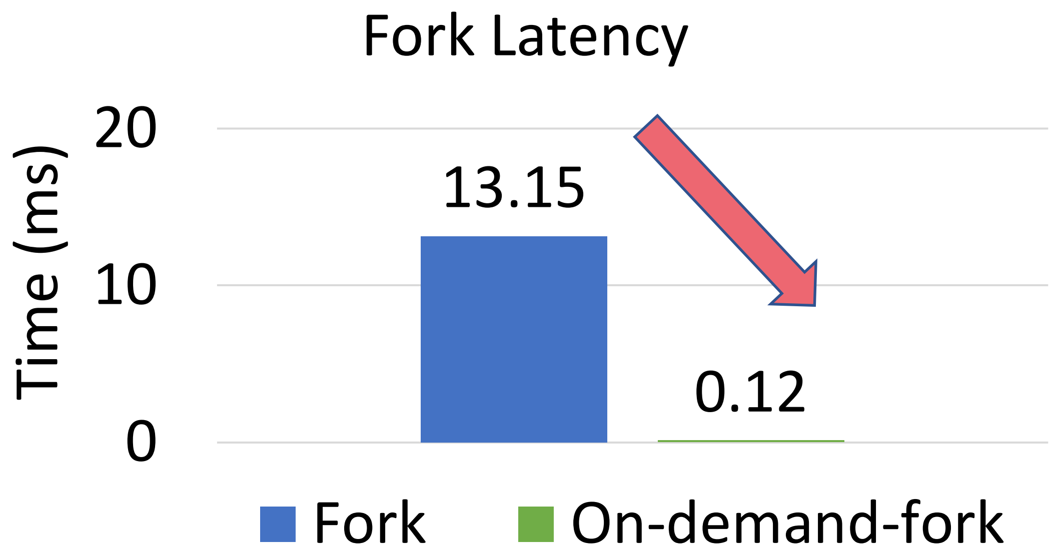
The test suite runs each test case in a child process

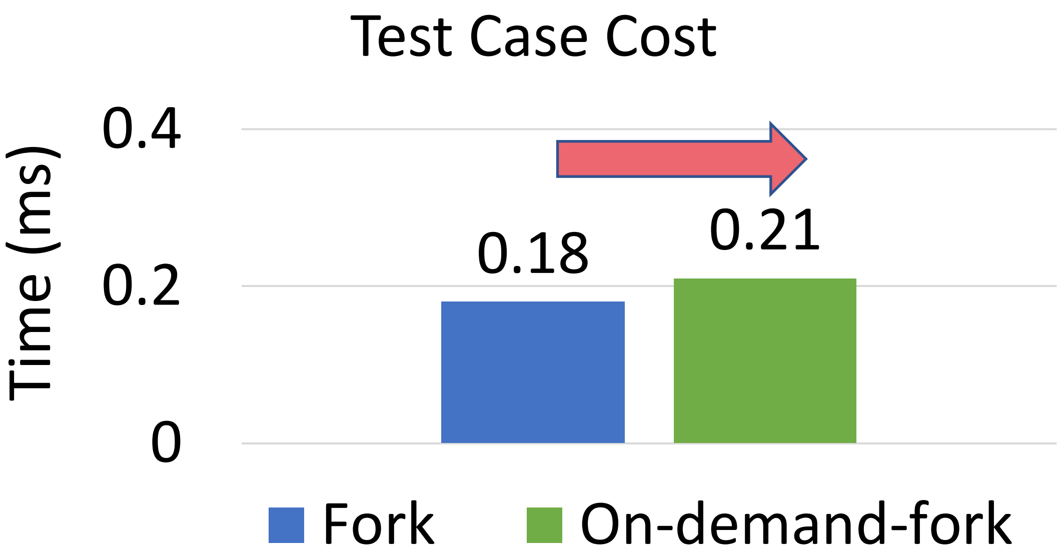


99% lower fork latency

Real-world applications: SQLite test suite

The test suite runs each test case in a child process





99% lower fork latency

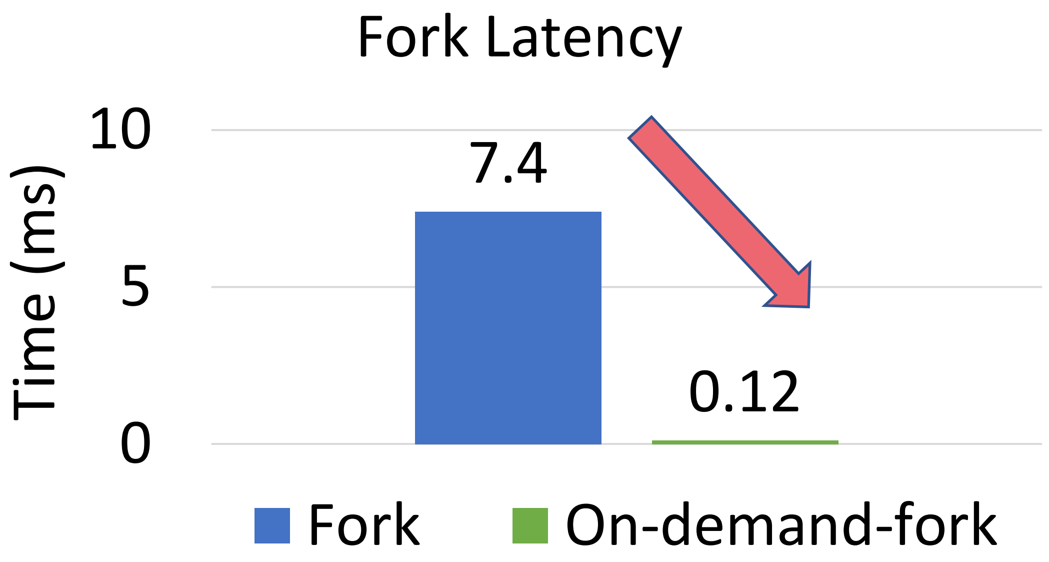
Similar cost of running a test

Real-world applications: Redis

Redis forks on the critical path to take snapshots

Real-world applications: Redis

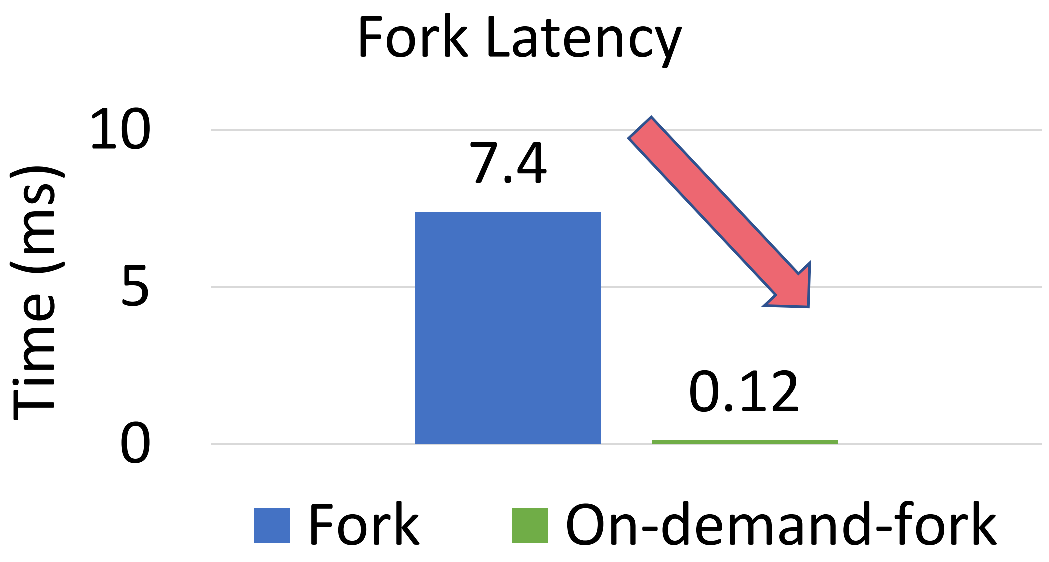
Redis forks on the critical path to take snapshots

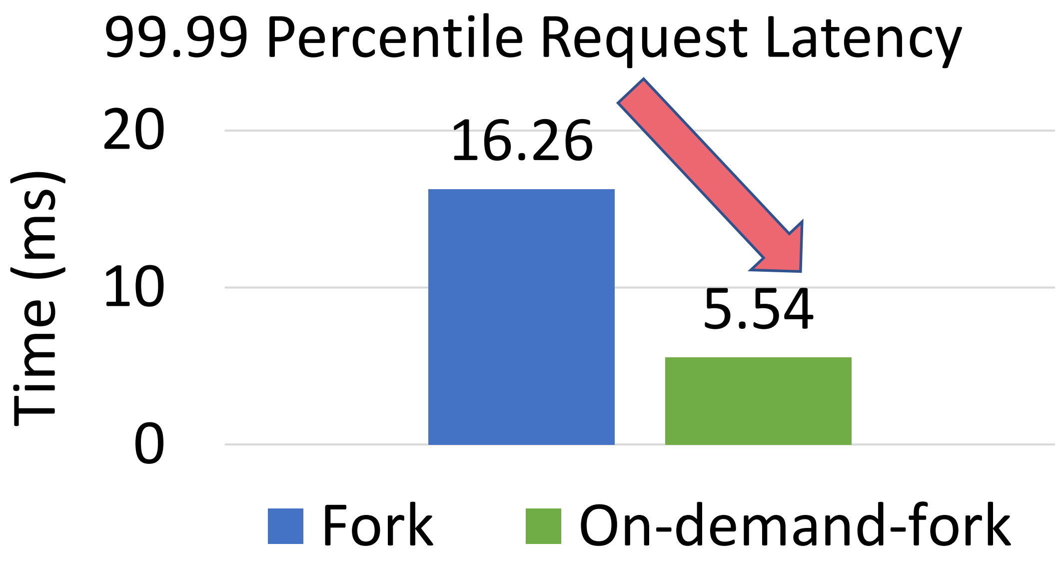


98% lower fork latency

Real-world applications: Redis

Redis forks on the critical path to take snapshots

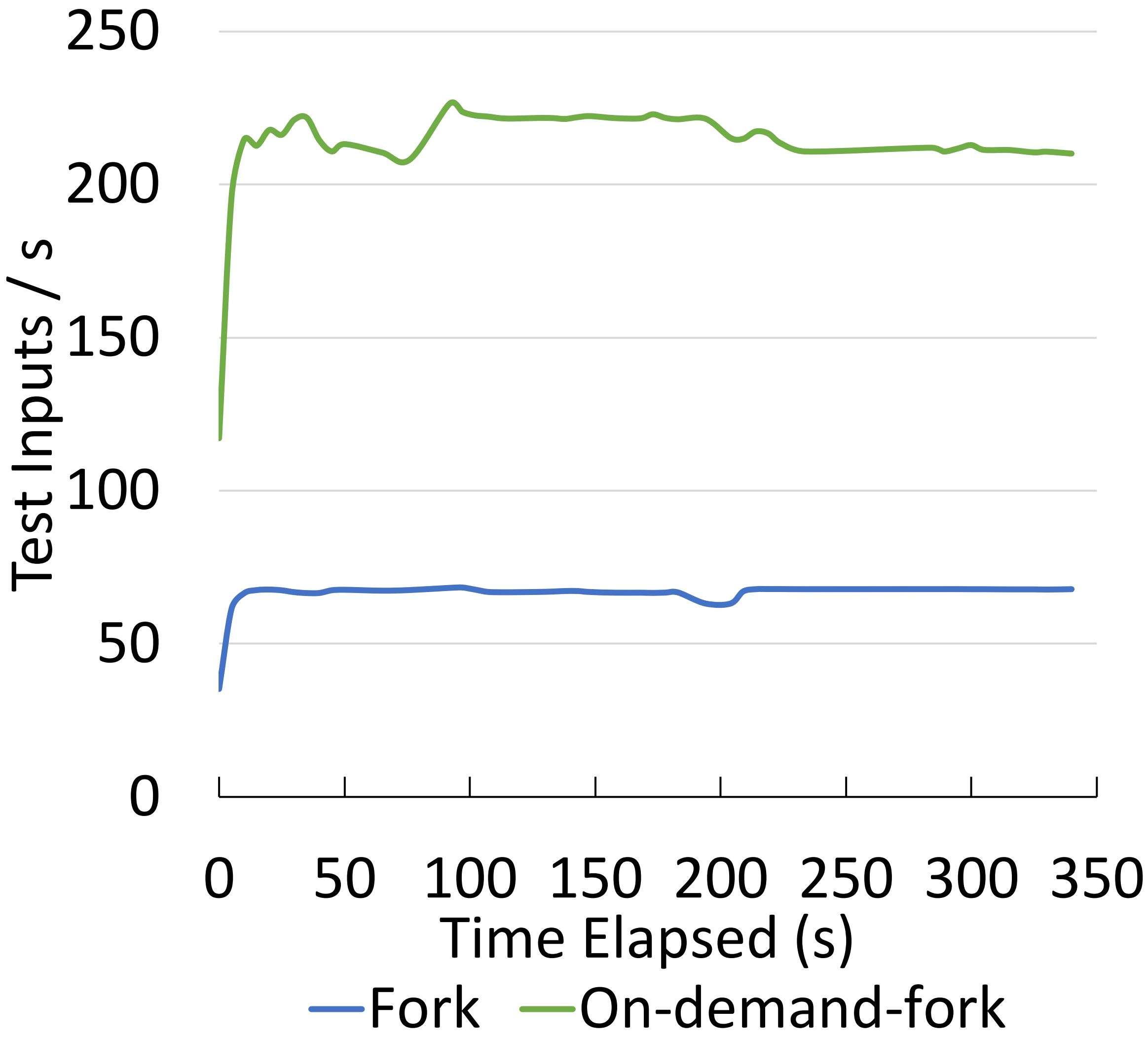




98% lower fork latency

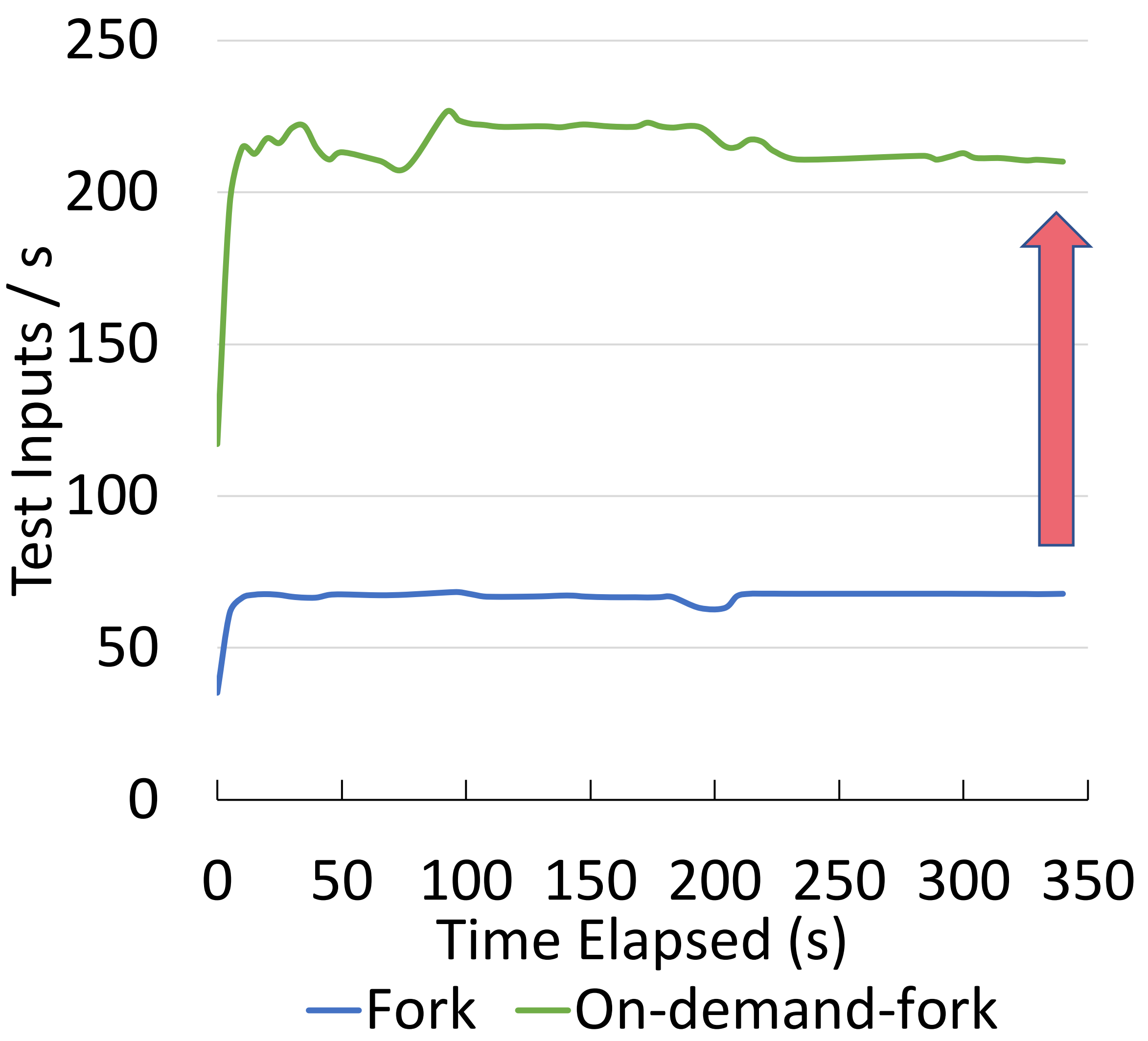
65.95% lower tail request latency

Real-world applications: AFL



AFL instruments the target program to repeatedly fork to take inputs

Real-world applications: AFL



AFL instruments the target program to repeatedly fork to take inputs

2.26 times higher fuzzing throughput

Conclusion

Traditional fork is slow

On-demand-fork is fast and efficient

https://github.com/rssys/on-demand-fork

PURDUE UNIVERSITY

RELIABLE & SECURE SYSTEMS

Conclusion

Traditional fork is slow

On-demand-fork is fast and efficient

270 times faster fork

https://github.com/rssys/on-demand-fork

Conclusion

Traditional fork is slow

On-demand-fork is fast and efficient

270 times faster fork

3.26 times fuzzing throughput

https://github.com/rssys/on-demand-fork

Conclusion

Traditional fork is slow

On-demand-fork is fast and efficient

270 times faster fork	3.26 times fuzzing throughput	65% lower Redis tail request latency

https://github.com/rssys/on-demand-fork