Funded by
the European Union

SnapBPF: Exploiting eBPF for Serverless Snapshot Prefetching

Stratos Psomadakis; Dimitrios Siakavaras; Chloe Alverti‡
Symeon Porgiotis; Orestis Lagkas Nikolos; Christos Katsakioris
Konstantinos Nikas; Georgios Goumas; Nectarios Koziris

National Technical University of Athens ‡ University of Illinois Urbana-Champaign

In a nutshell

In a nutshell

***FaaS Cold Start Overhead**:*
→ Sandbox creation dominates function latency
→ Exacerbated with microVM sandboxing

In a nutshell

***FaaS Cold Start Overhead**:*
→ Sandbox creation dominates function latency
→ Exacerbated with microVM sandboxing

microVM ***snapshotting:***
→ Reduces sandbox creation overhead
→ Incurs snapshot loading (I/O) overhead

In a nutshell

***FaaS Cold Start Overhead**:*
→ Sandbox creation dominates function latency
→ Exacerbated with microVM sandboxing

microVM ***snapshotting:***
→ Reduces sandbox creation overhead
→ Incurs snapshot loading (I/O) overhead

State-of-the-art (REAP, FaaSnap, Faast):
Capture and ***prefetch*** the function ***working set***
→ Minimize snapshot loading overhead
→ Implemented in userspace
→ *Storage and memory duplication*

In a nutshell

***FaaS Cold Start Overhead**:*
→ Sandbox creation dominates function latency
→ Exacerbated with microVM sandboxing

microVM ***snapshotting:***
→ Reduces sandbox creation overhead
→ Incurs snapshot loading (I/O) overhead

Our Proposal: ***SnapBPF***
→ ***eBPF*** capture and prefetch
→ Implemented in *kernelspace*
→ No working set serialization
→ *Working set deduplication*
→ PV free page filtering

Outline

→ FaaS Snapshotting

→ Working Set Prefetching

→ SnapBPF

i. eBPF capture and prefetch

ii. PV free page filtering

→ Evaluation

→ Conclusion

Outline

→ FaaS Snapshotting

→ Working Set Prefetching

→ SnapBPF

i. eBPF capture and prefetch

ii. PV free page filtering

→ Evaluation

→ Conclusion

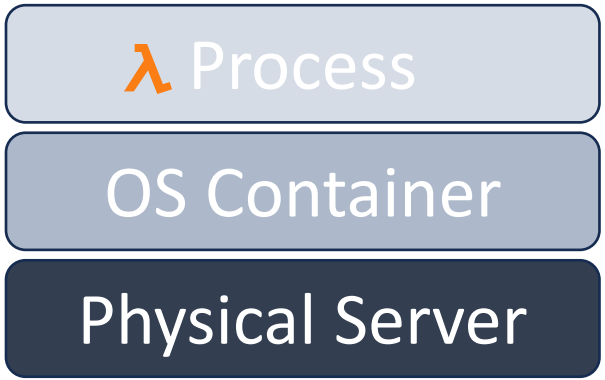
Sandboxing serverless functions

Sandboxing serverless functions

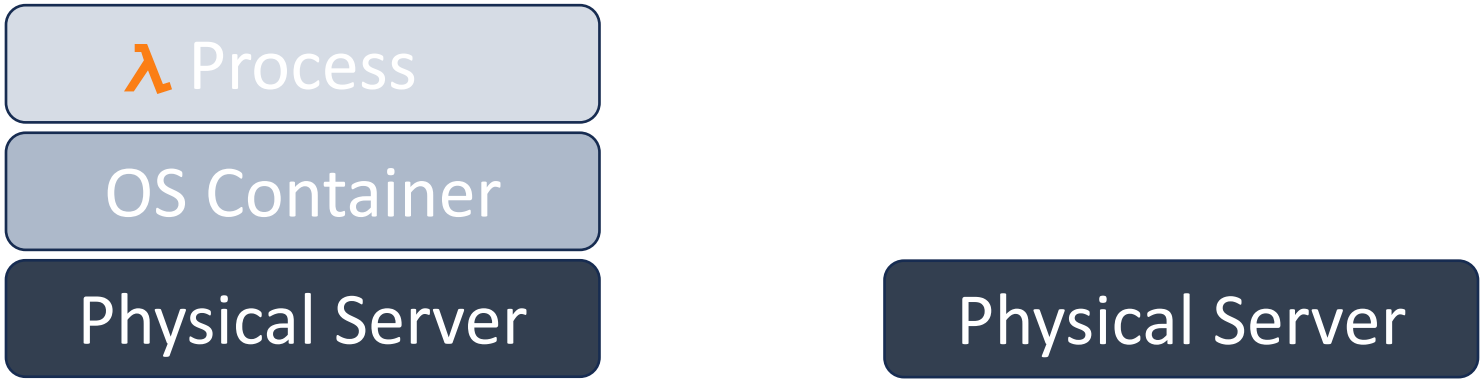
Physical Server

Sandboxing serverless functions

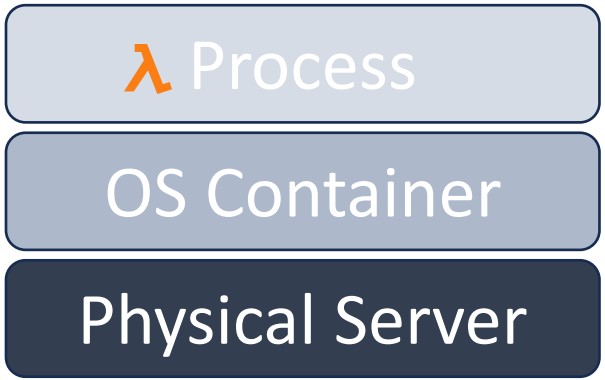
Sandboxing serverless functions



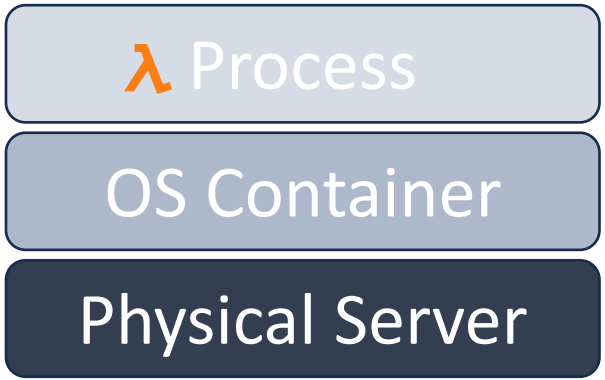
Sandboxing serverless functions

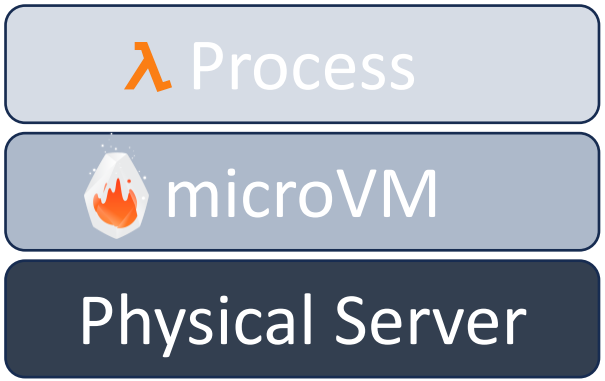


Sandboxing serverless functions

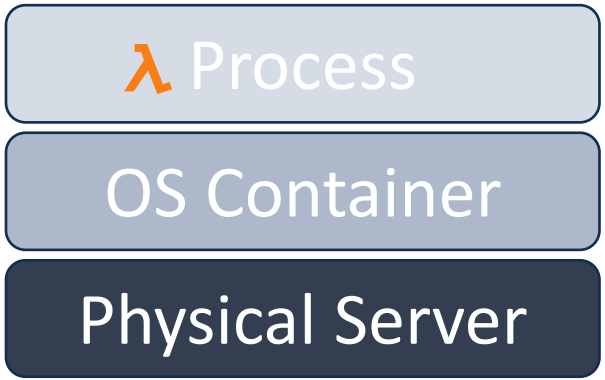


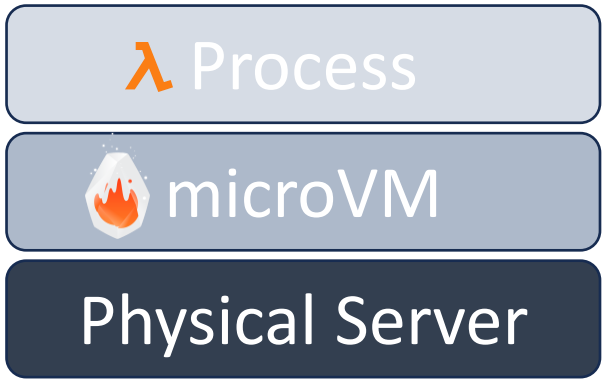
Sandboxing serverless functions



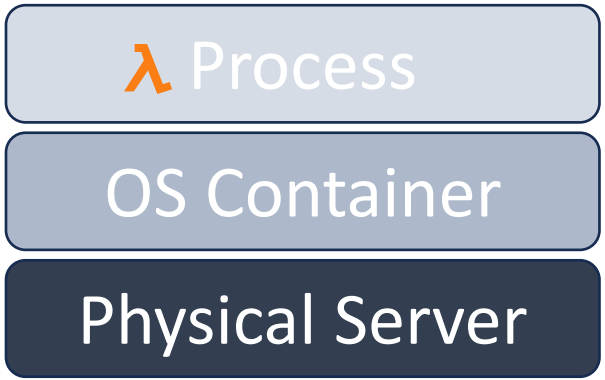


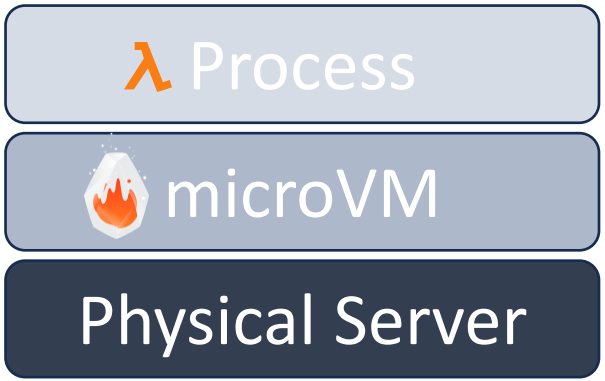
Sandboxing serverless functions





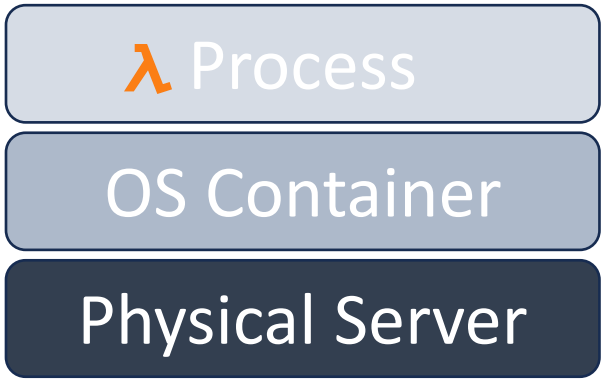
Sandboxing serverless functions

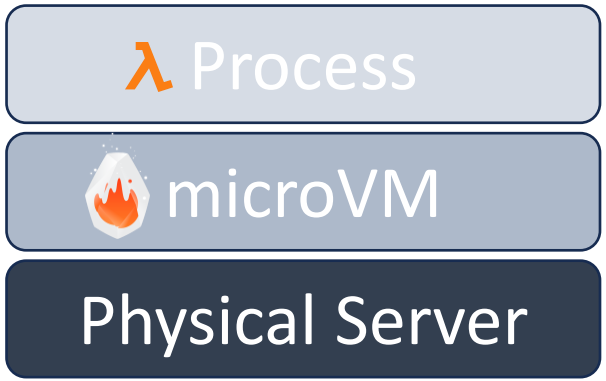




Isolation Performance

Sandboxing serverless functions





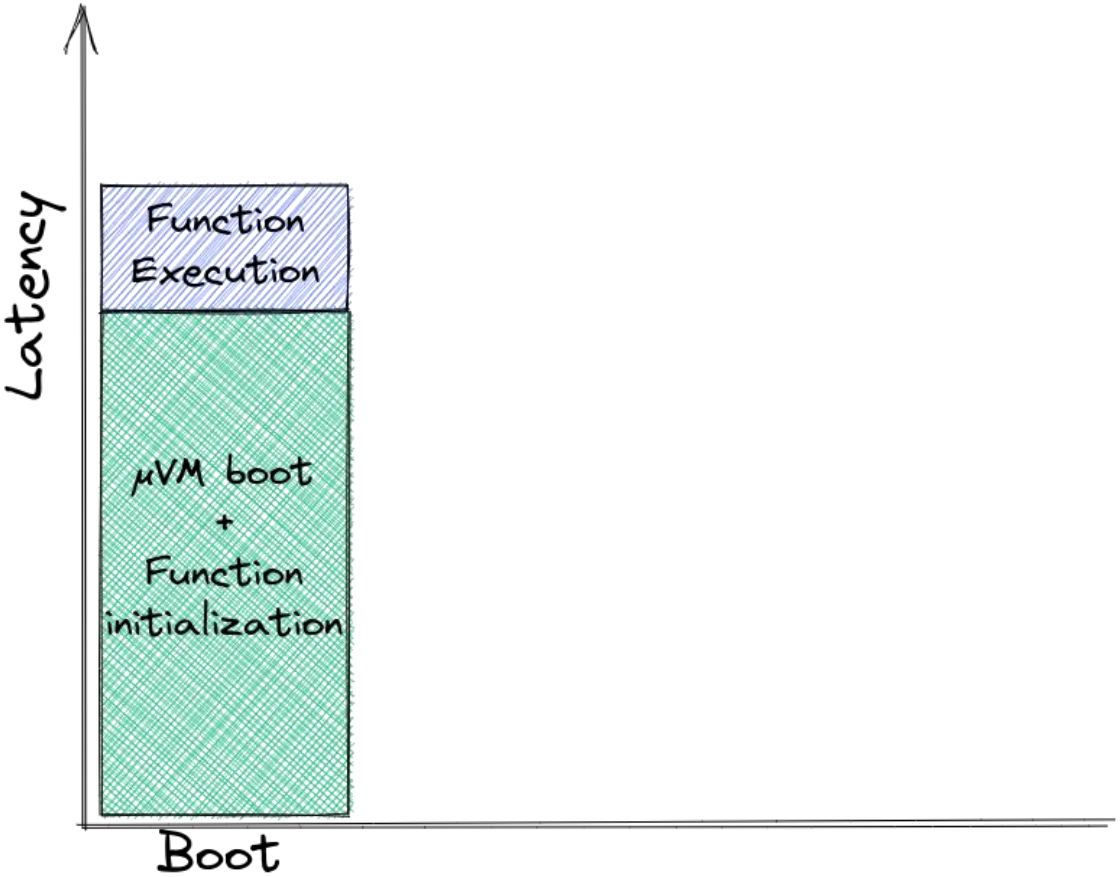
Isolation Performance

Isolation Guarantees

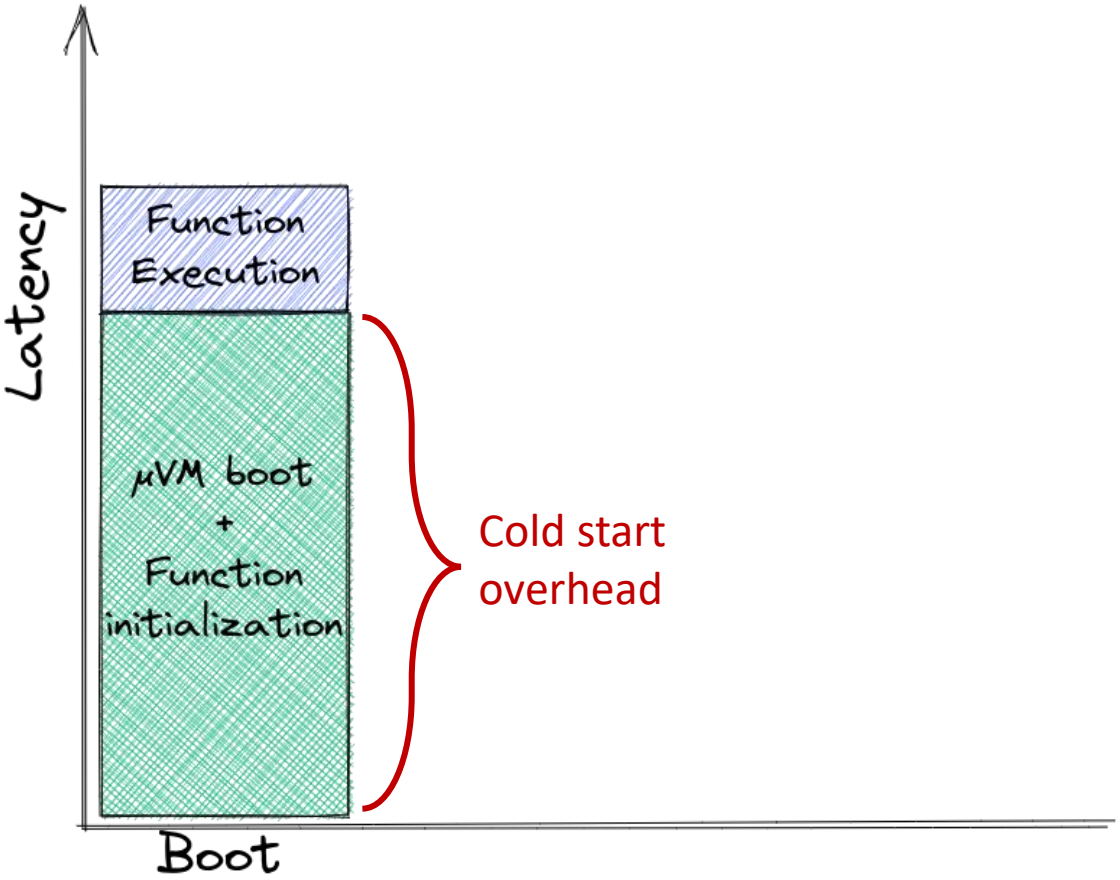
The cold-start pain

The cold-start pain

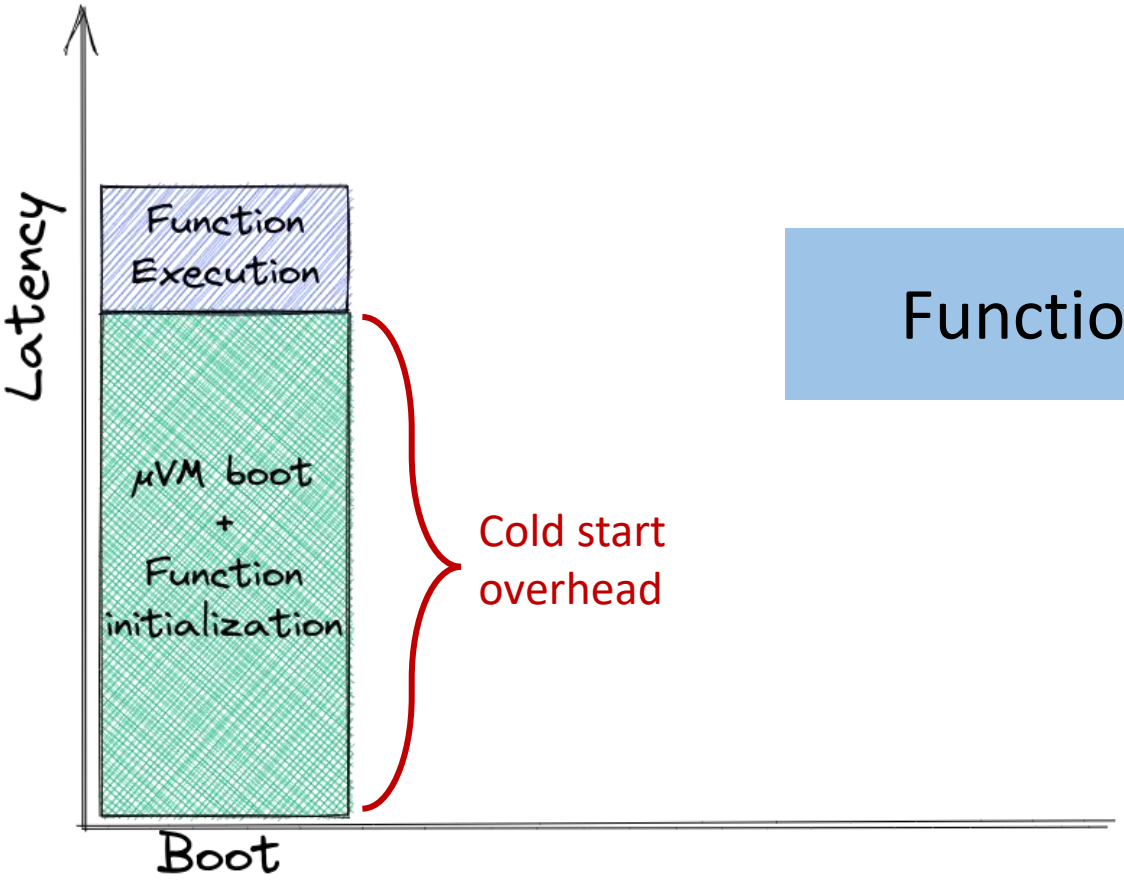
The cold-start pain



The cold-start pain

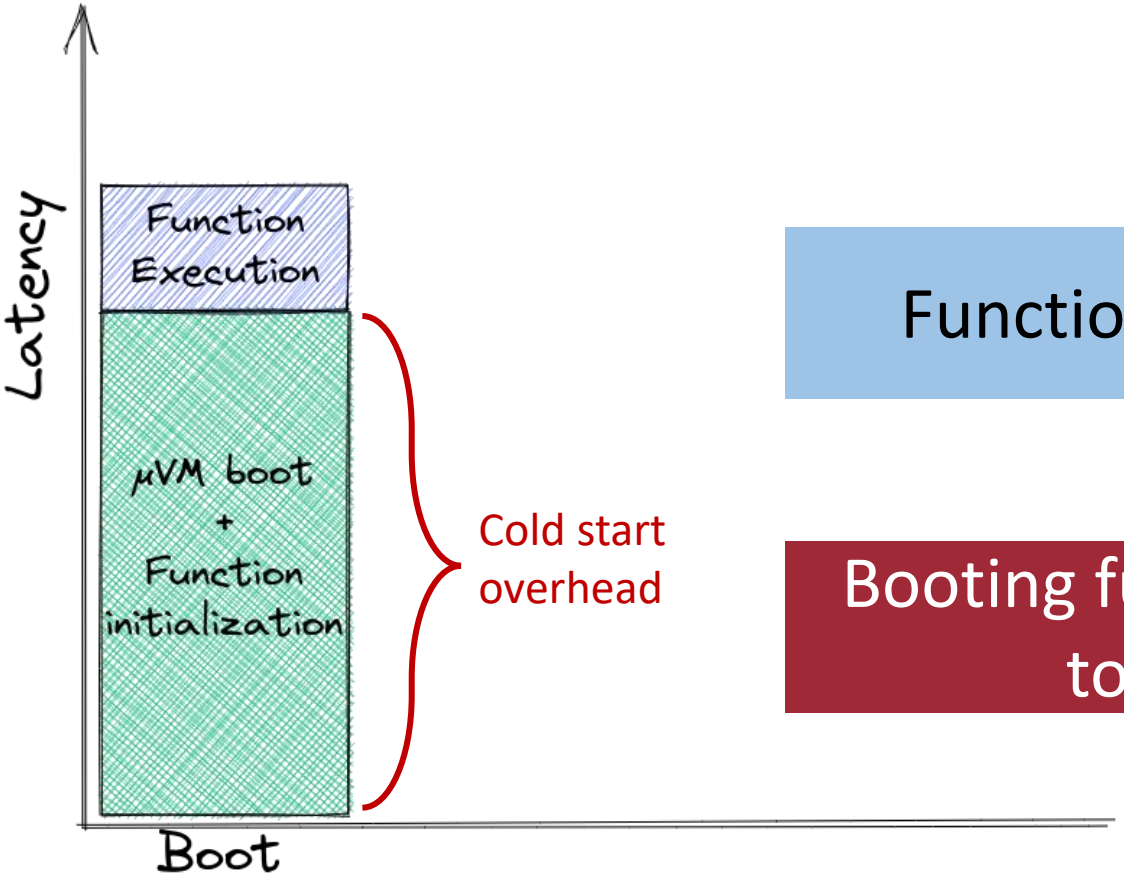


The cold-start pain



Functions are short-lived

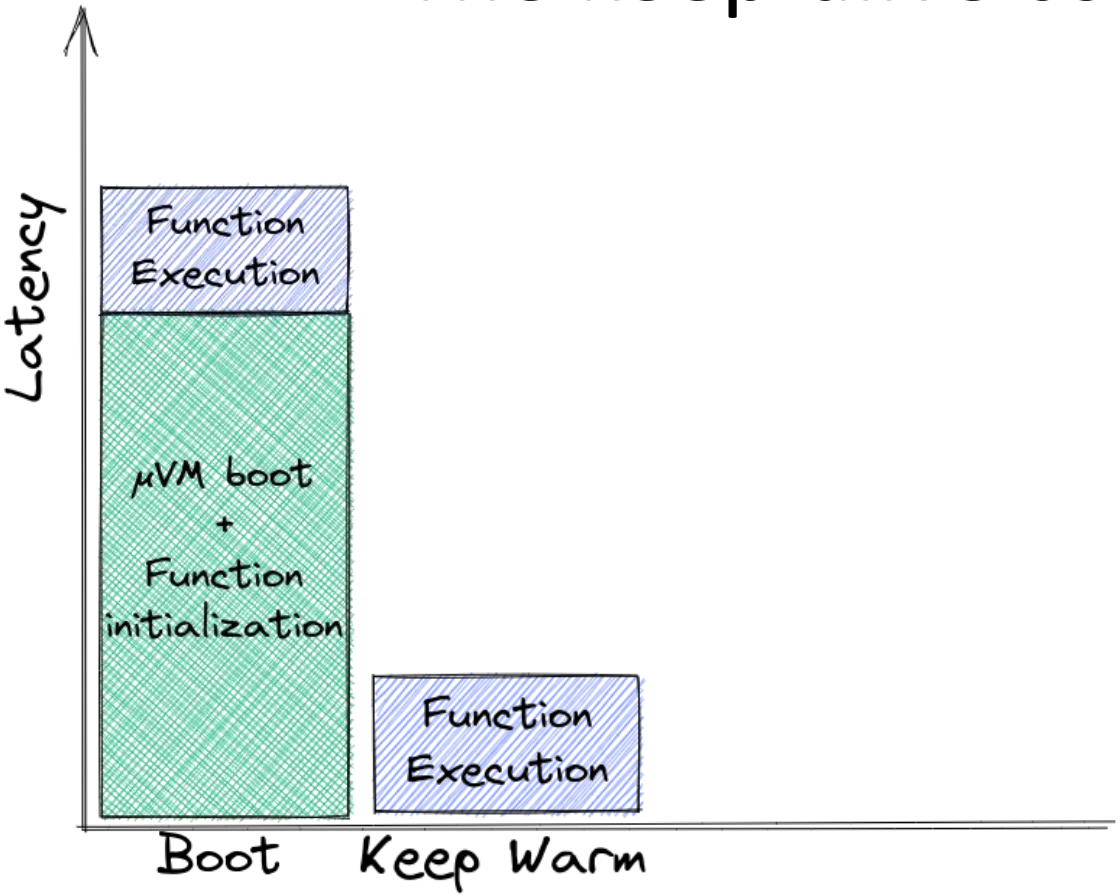
The cold-start pain



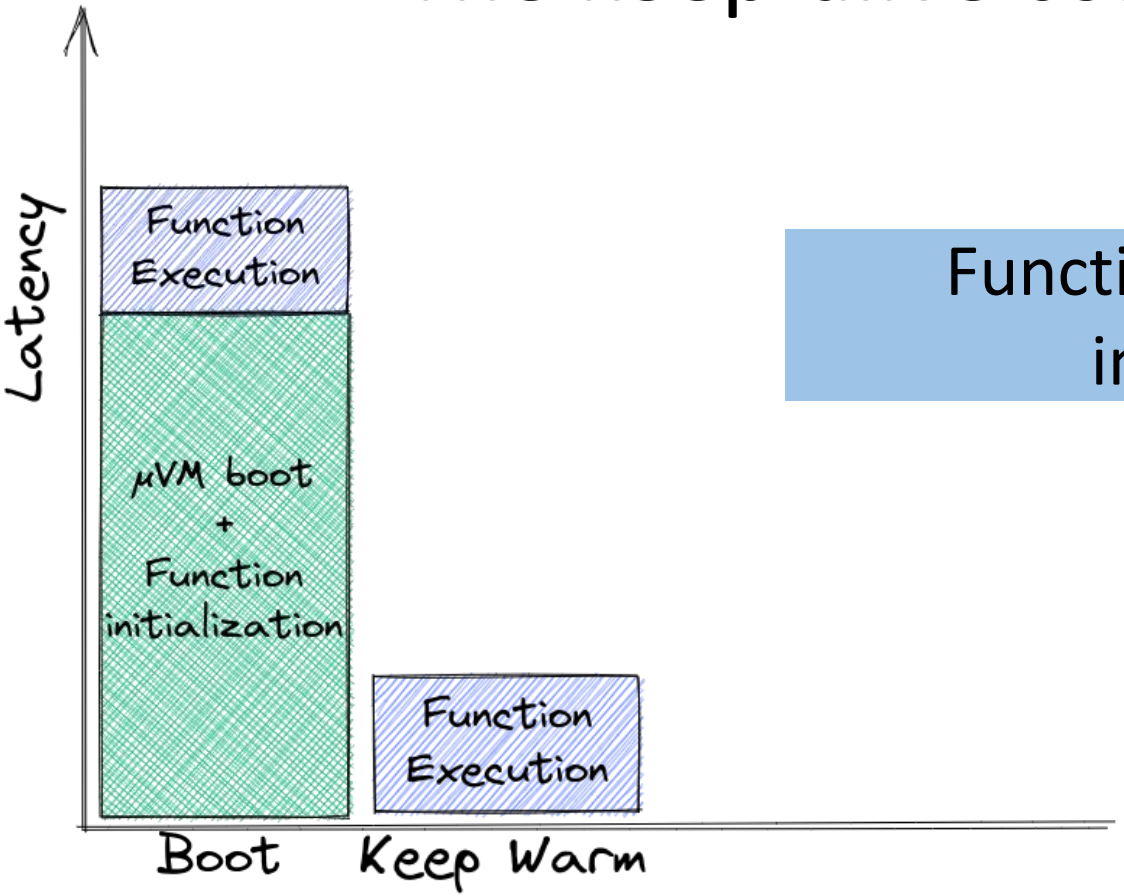
Functions are short-lived

Booting function instances is too expensive

The keep-alive conundrum

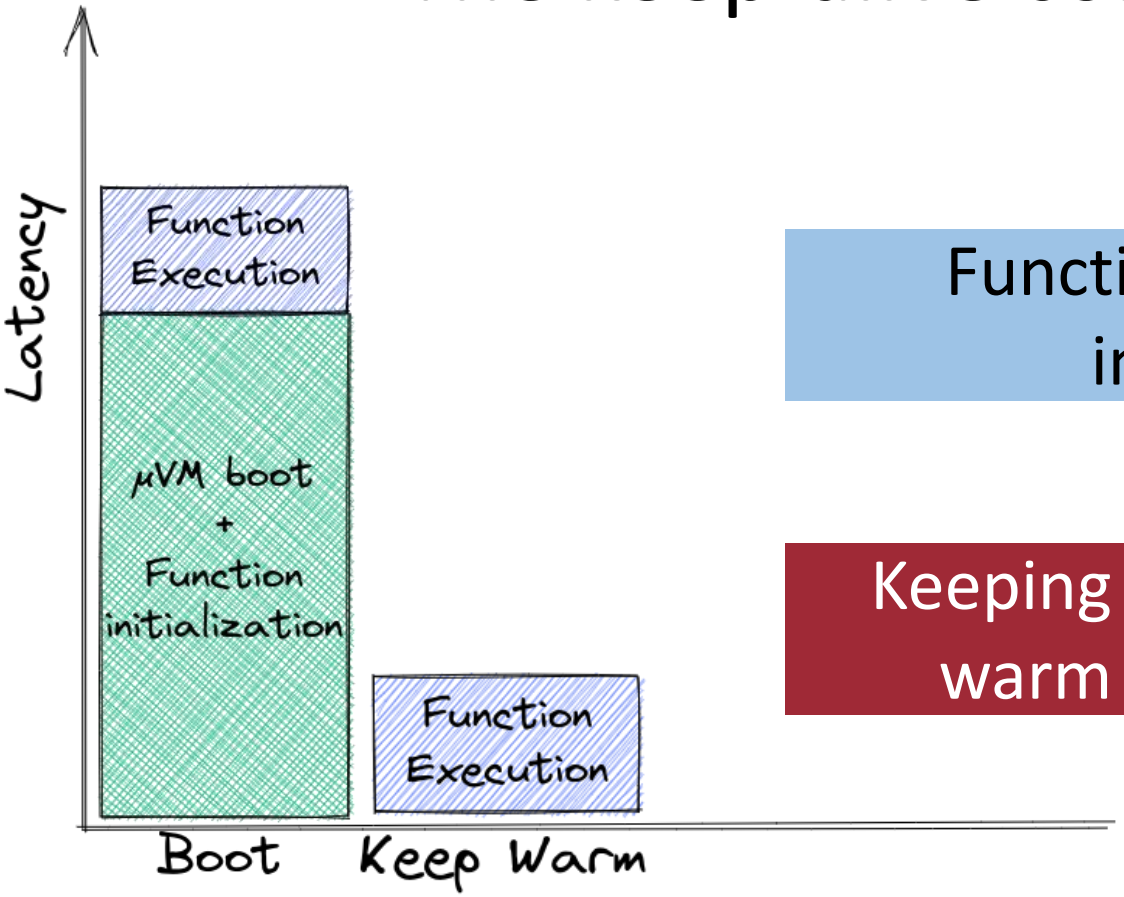


The keep-alive conundrum



Functions are invoked infrequently

The keep-alive conundrum

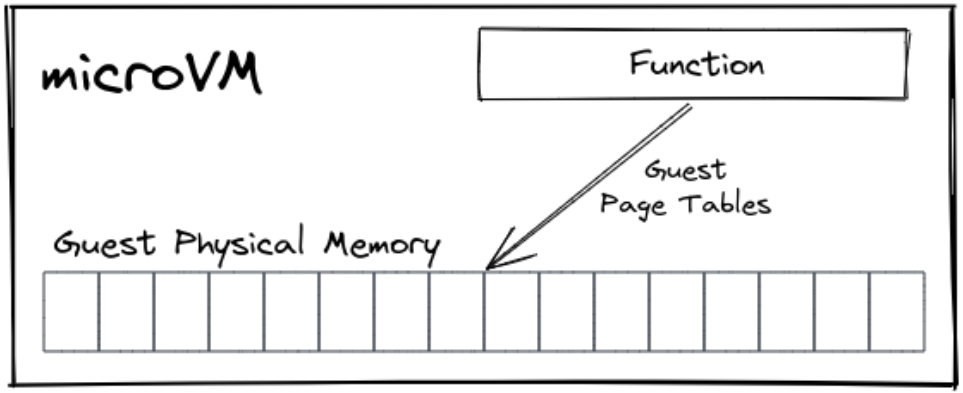


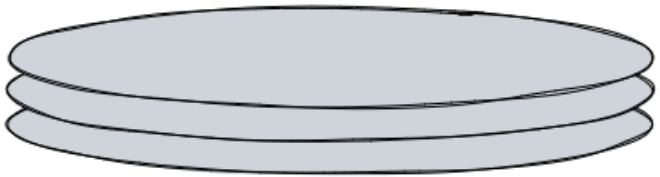
Functions are invoked infrequently

Keeping function instances warm is too expensive

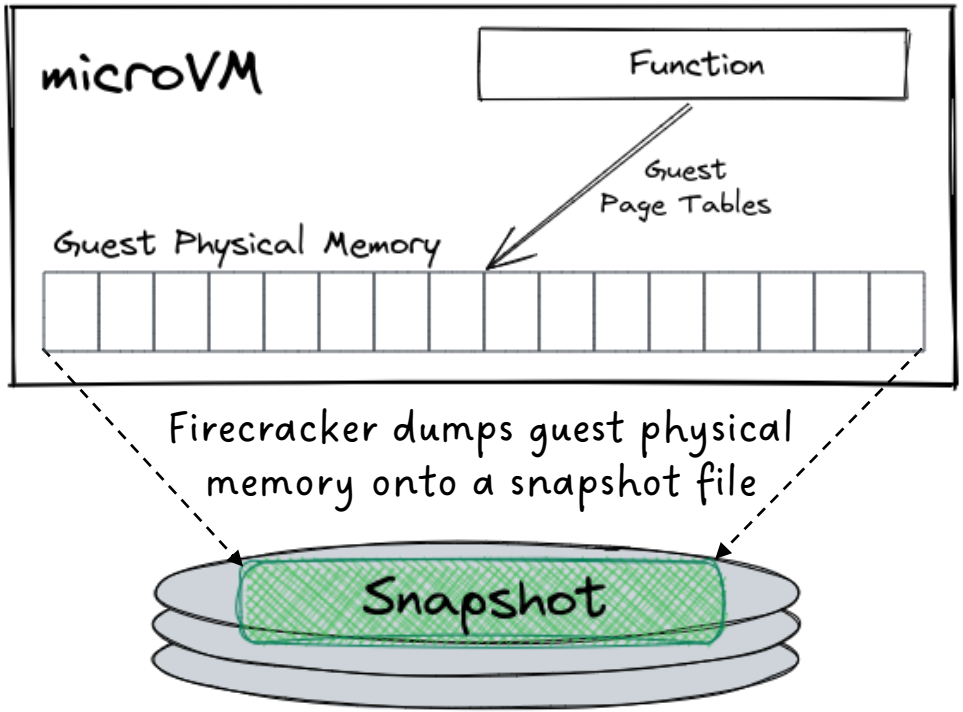
Snapshots to the rescue!

Snapshots to the rescue!



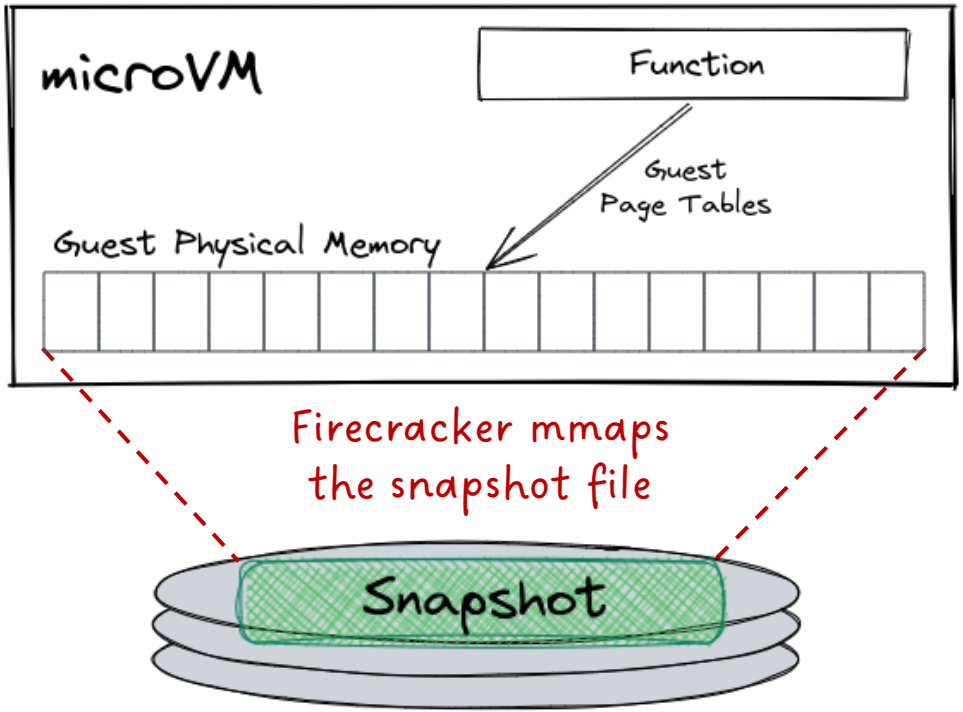


Snapshots to the rescue!

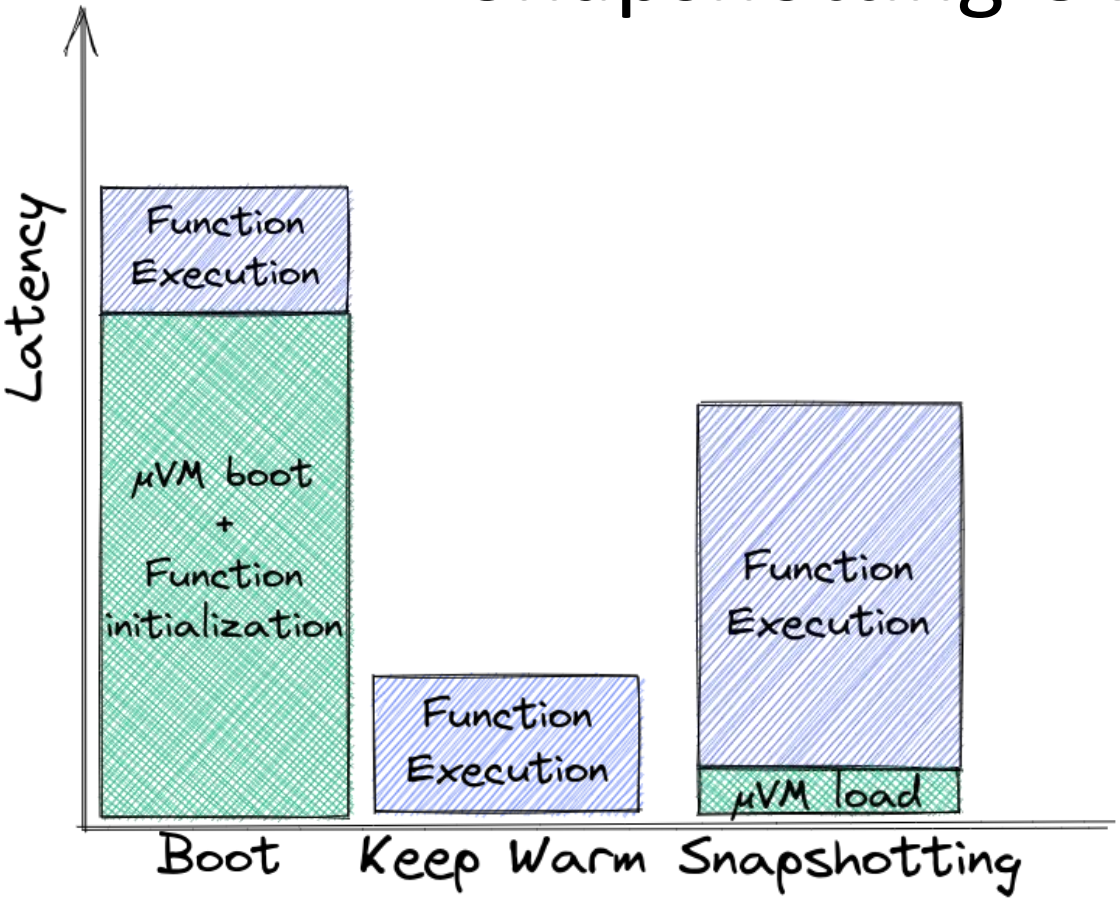


Snapshots to the rescue!

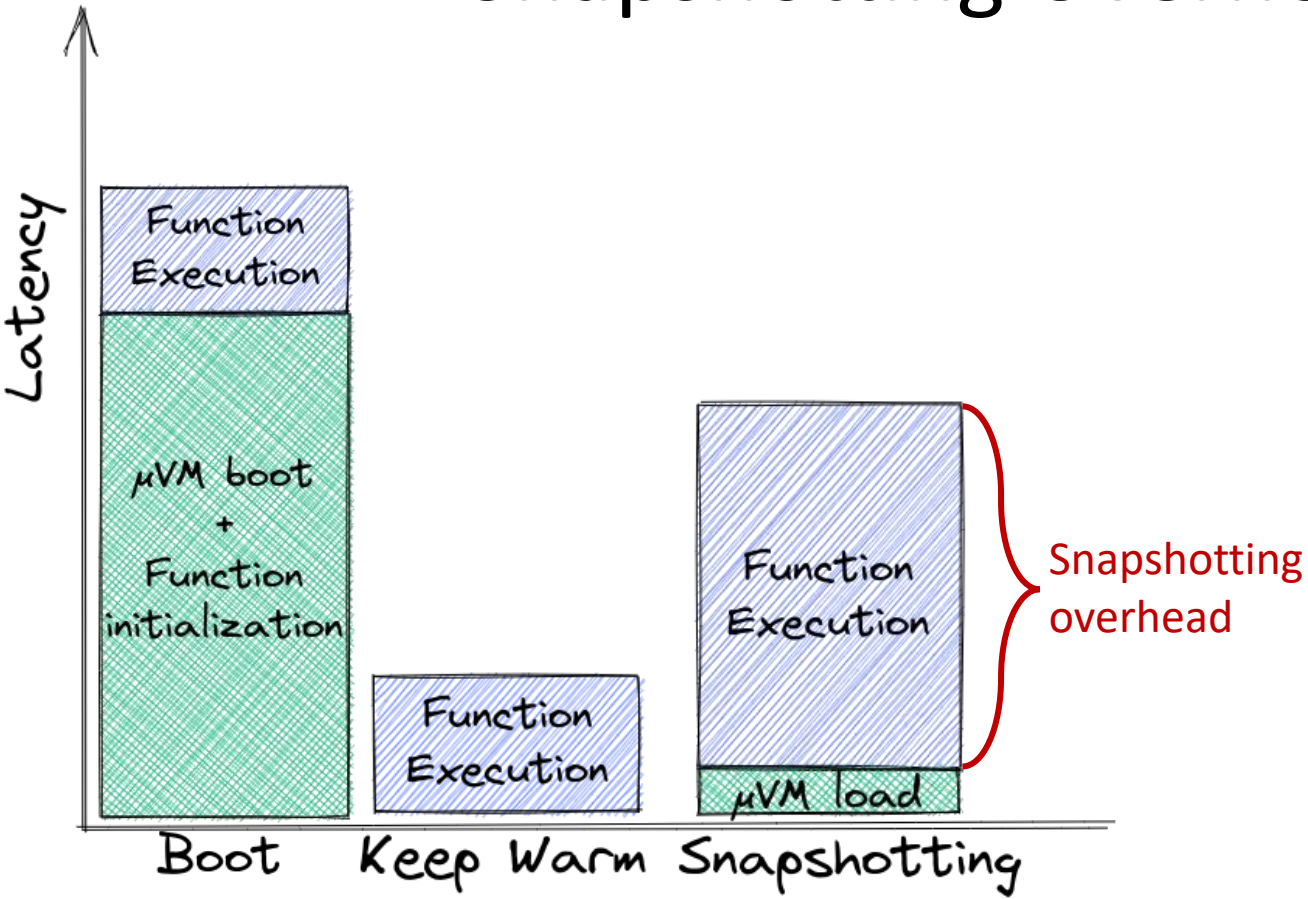
Snapshots to the rescue!



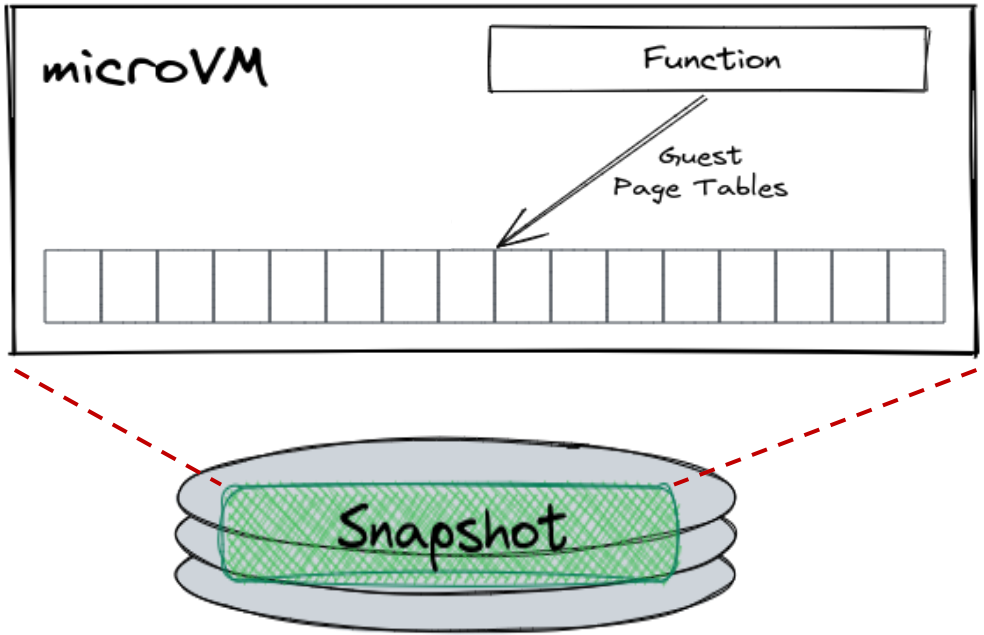
Snapshotting Overhead



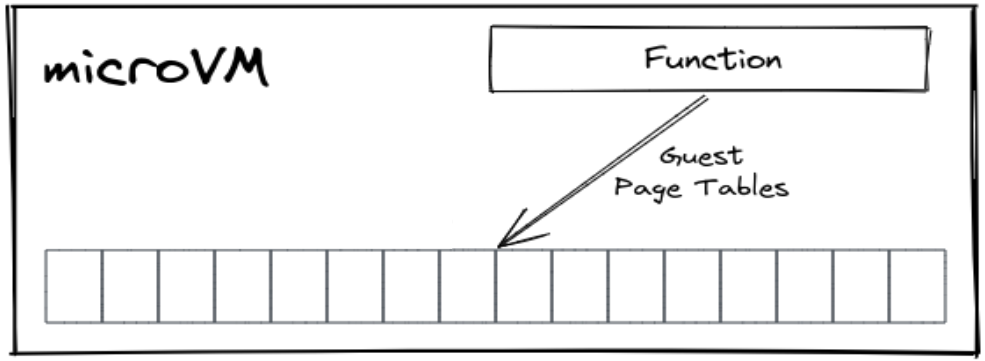
Snapshotting Overhead



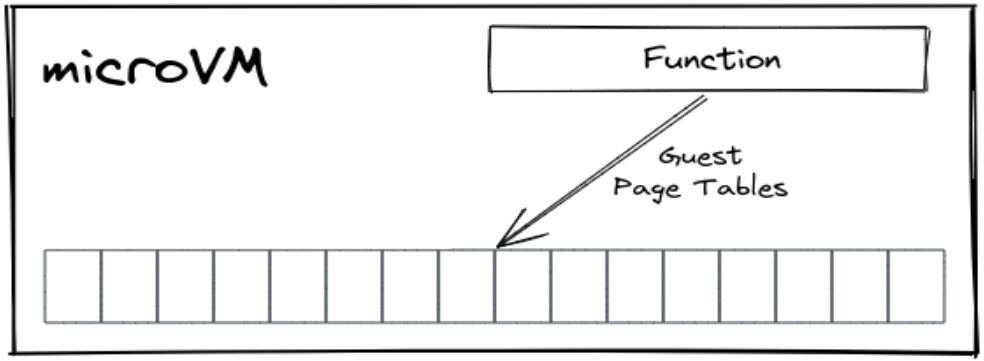
Snapshotting Overhead



Snapshotting Overhead

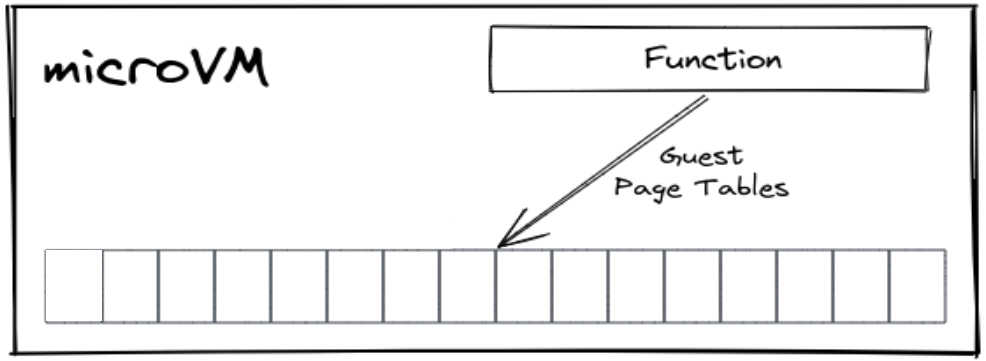


Snapshotting Overhead

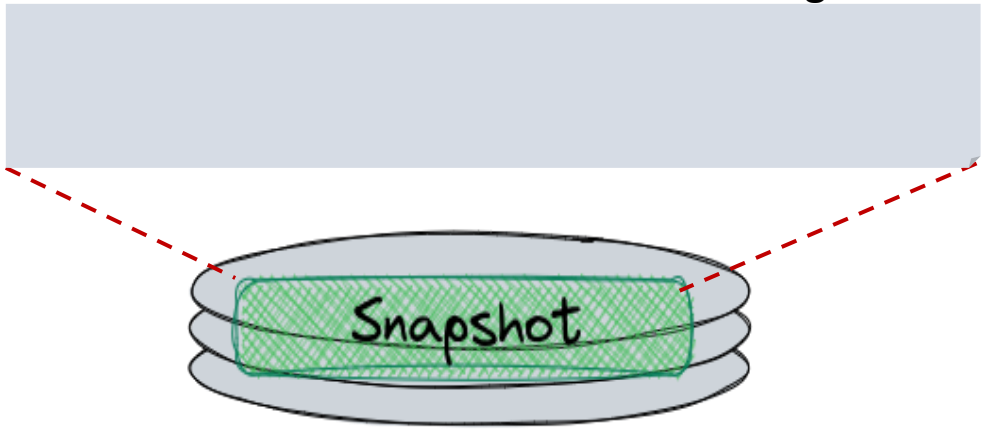


OS Page Cache

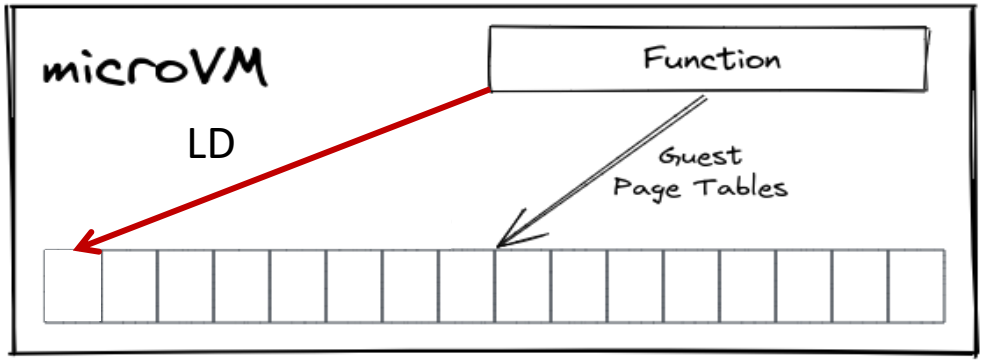
Snapshotting Overhead

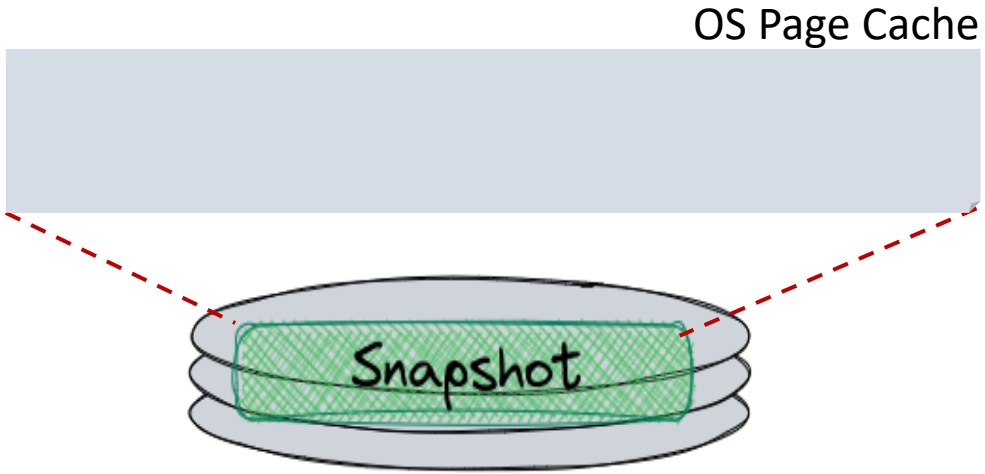


OS Page Cache

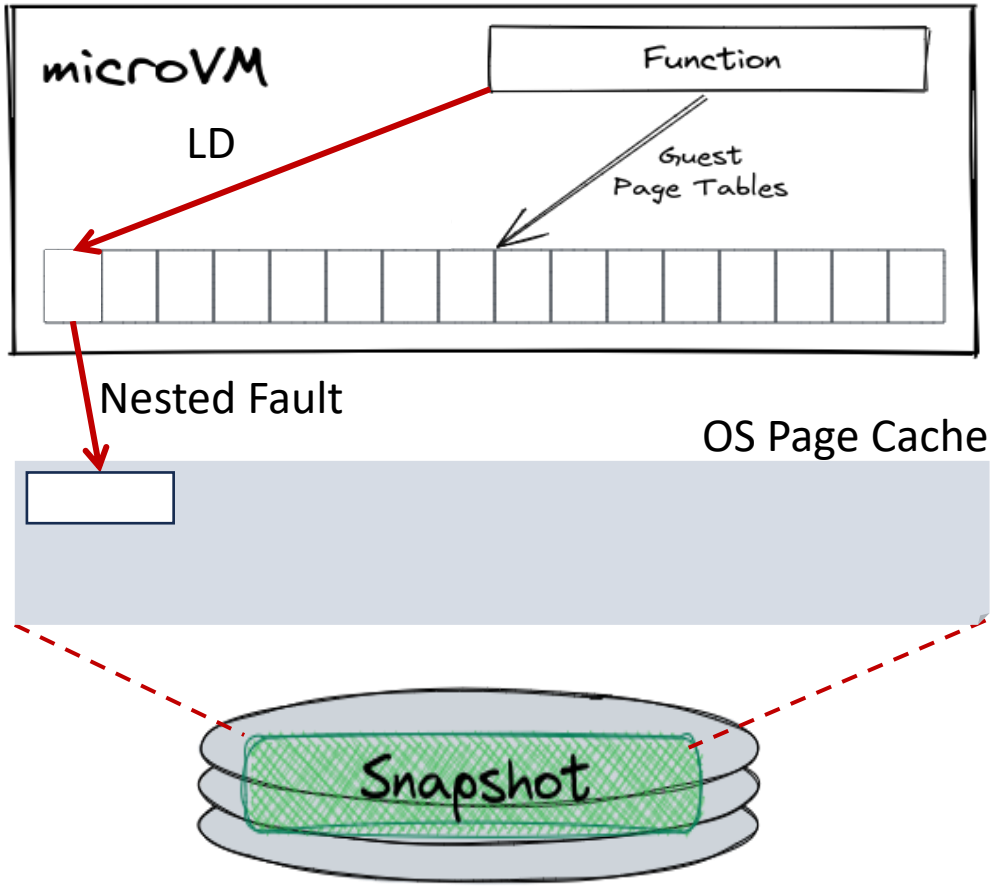


Snapshotting Overhead

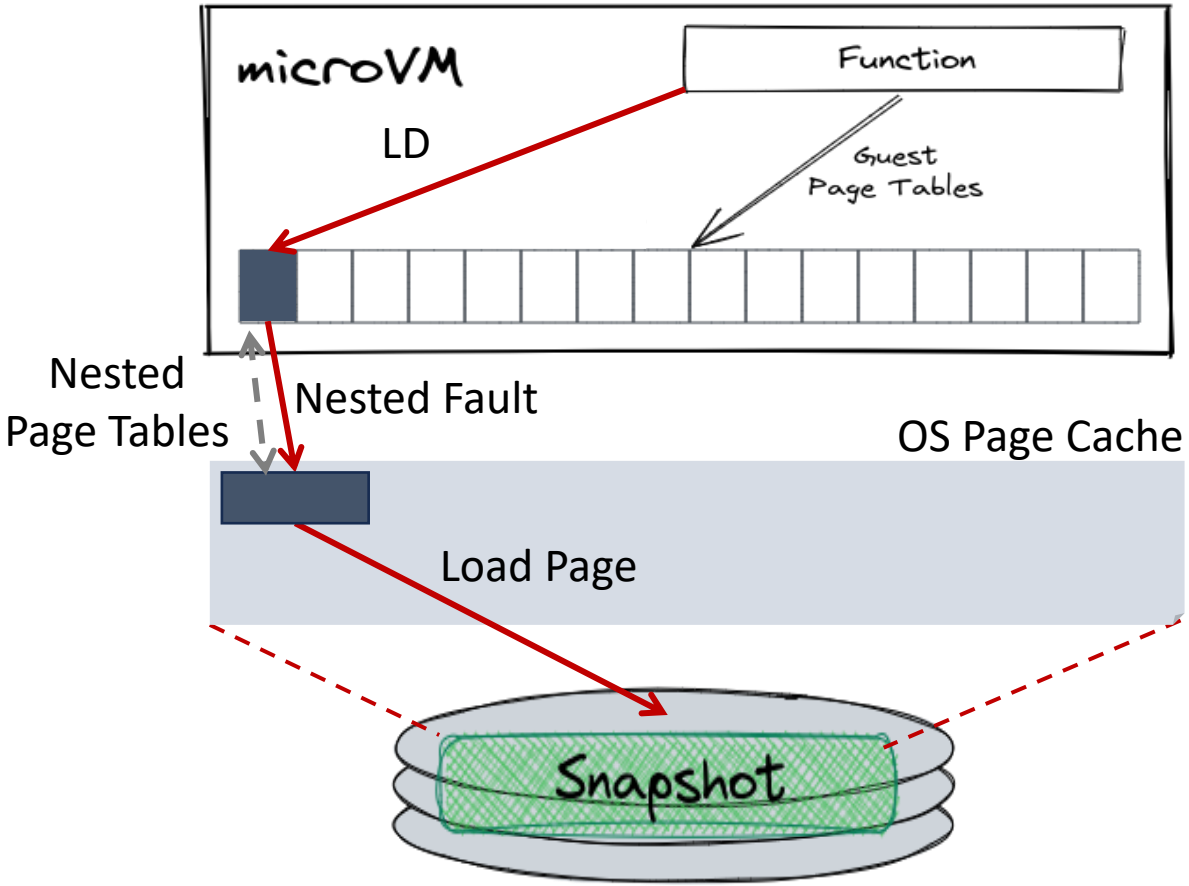




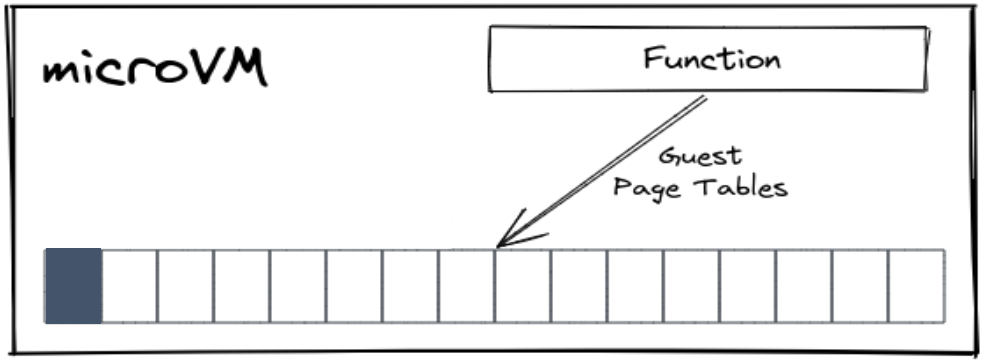
Snapshotting Overhead

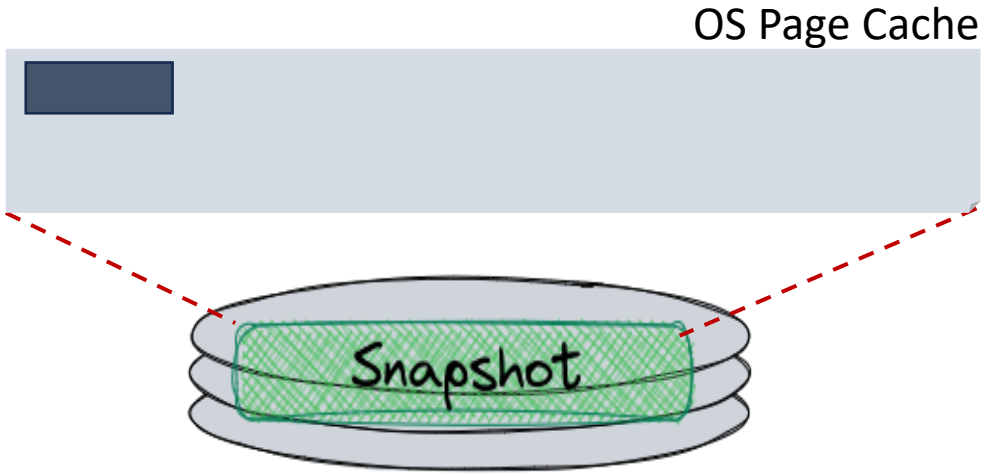


Snapshotting Overhead

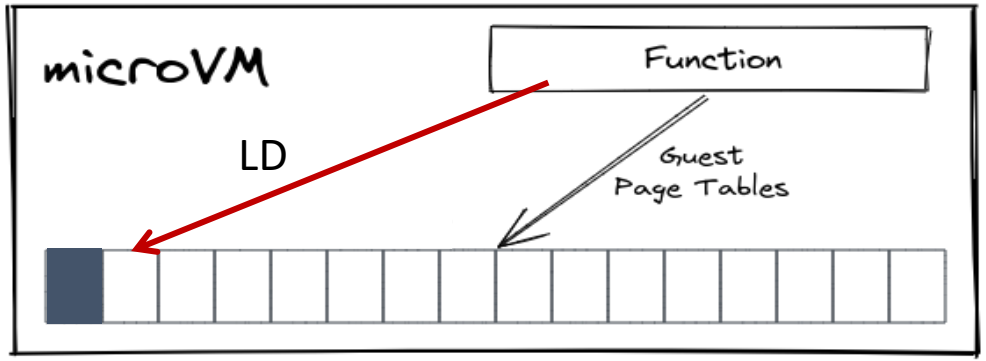


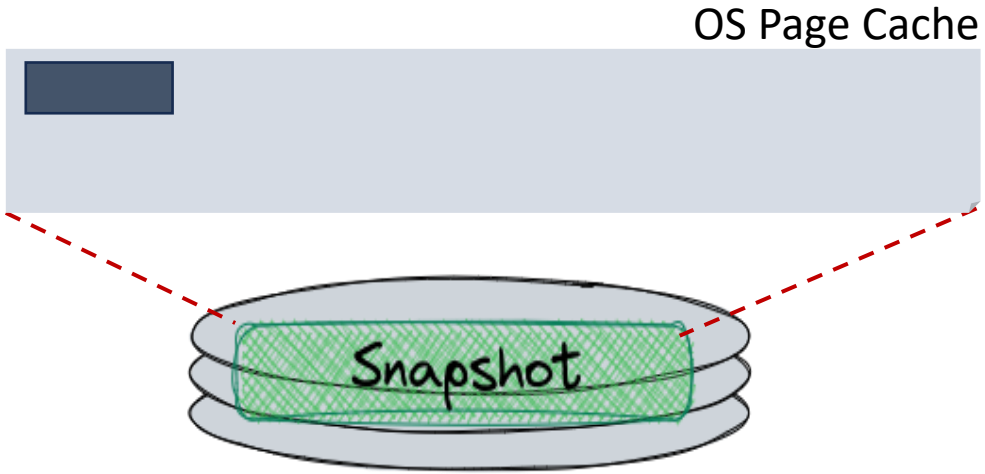
Snapshotting Overhead



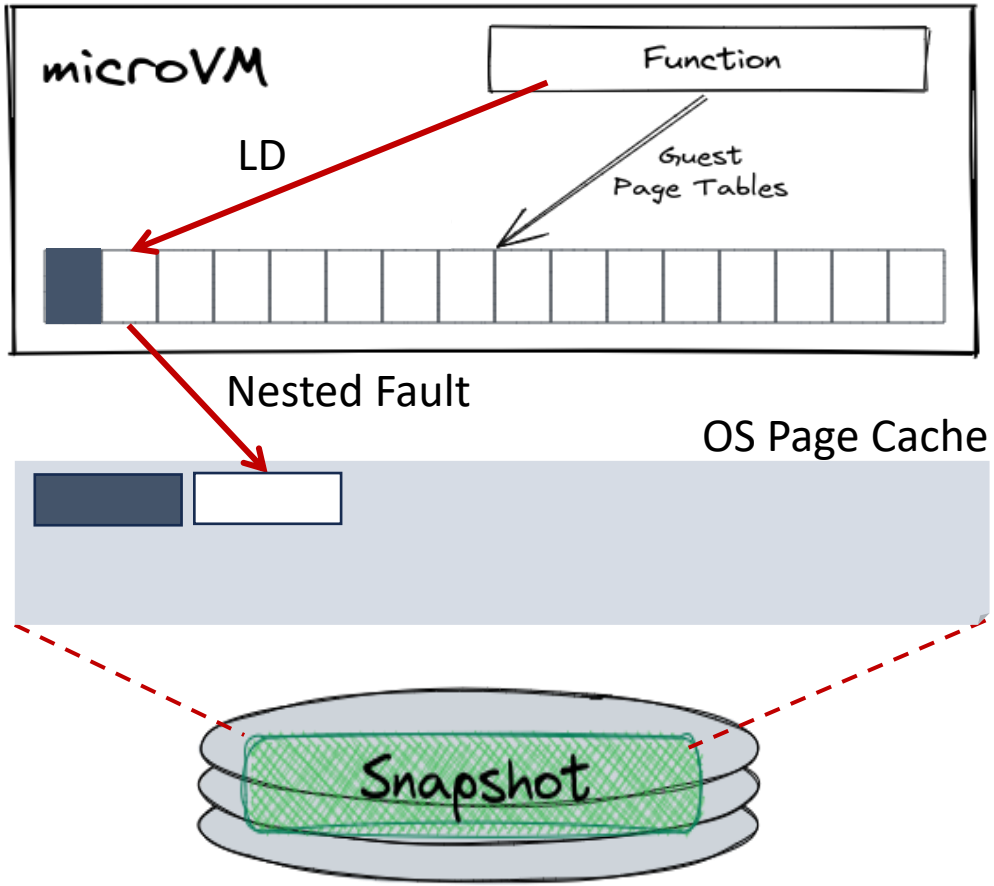


Snapshotting Overhead

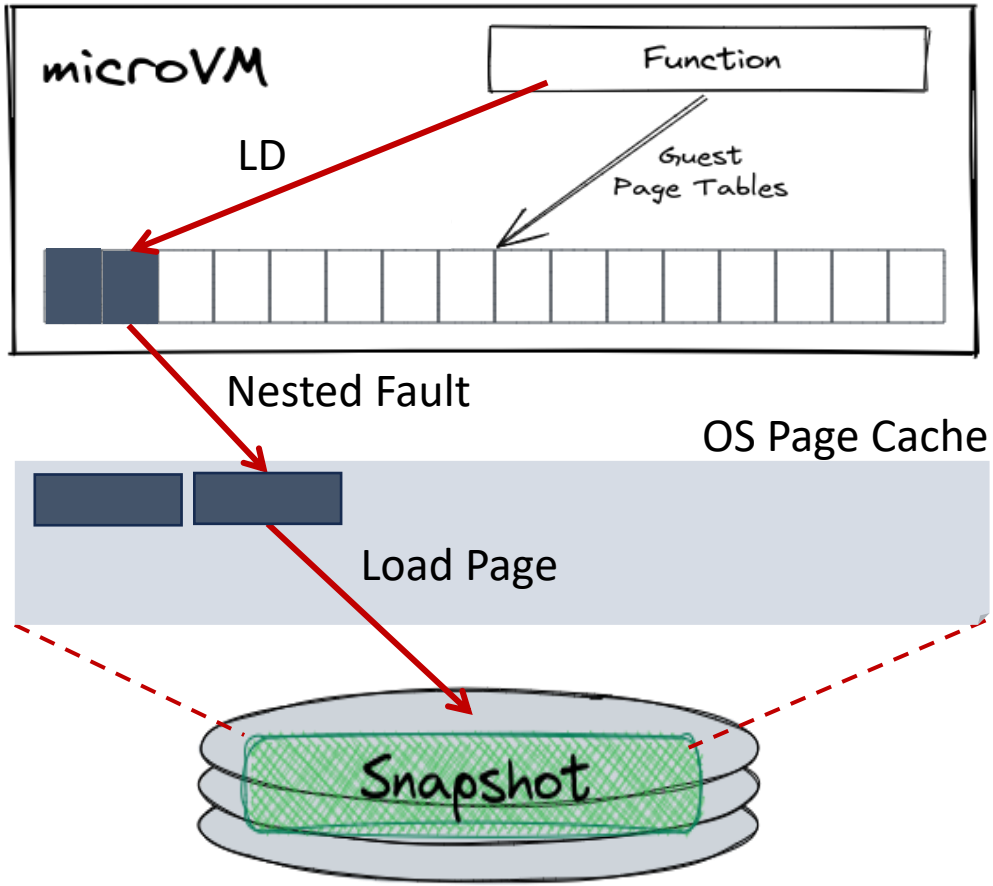




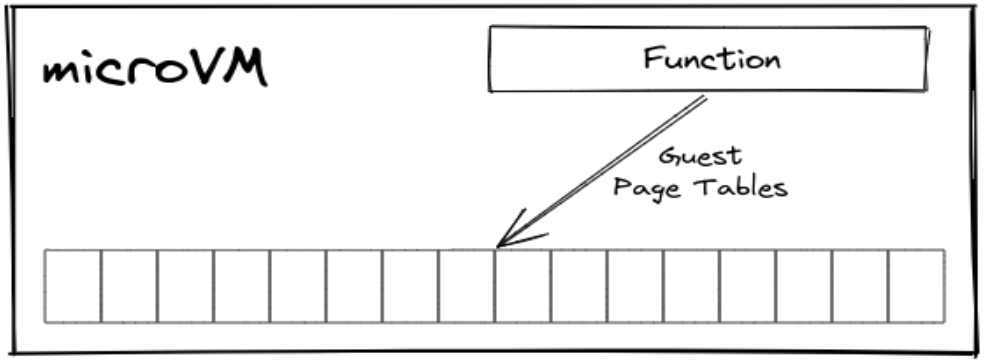
Snapshotting Overhead

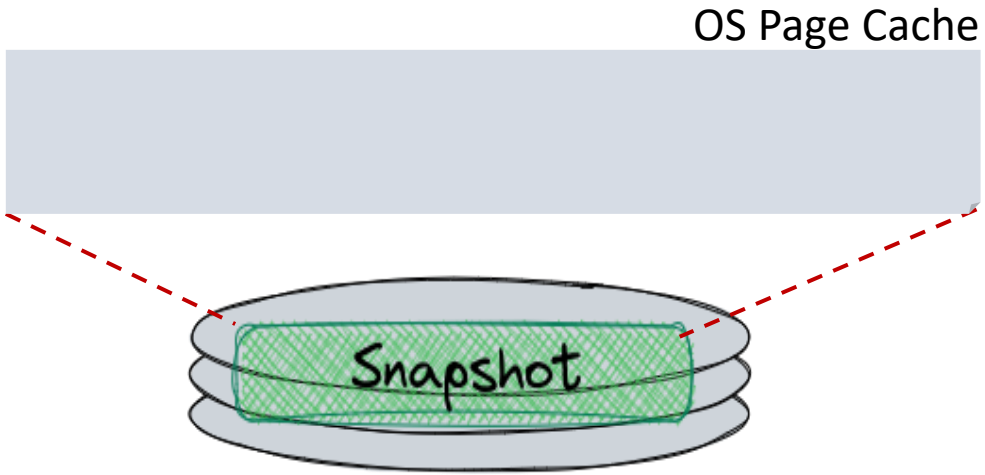


Snapshotting Overhead

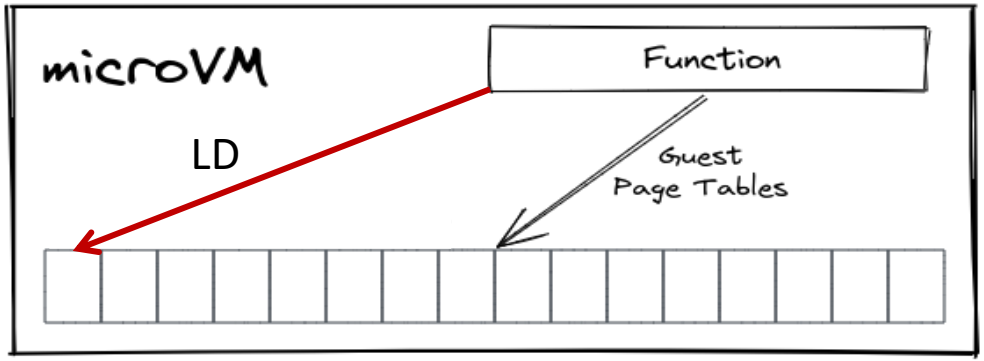


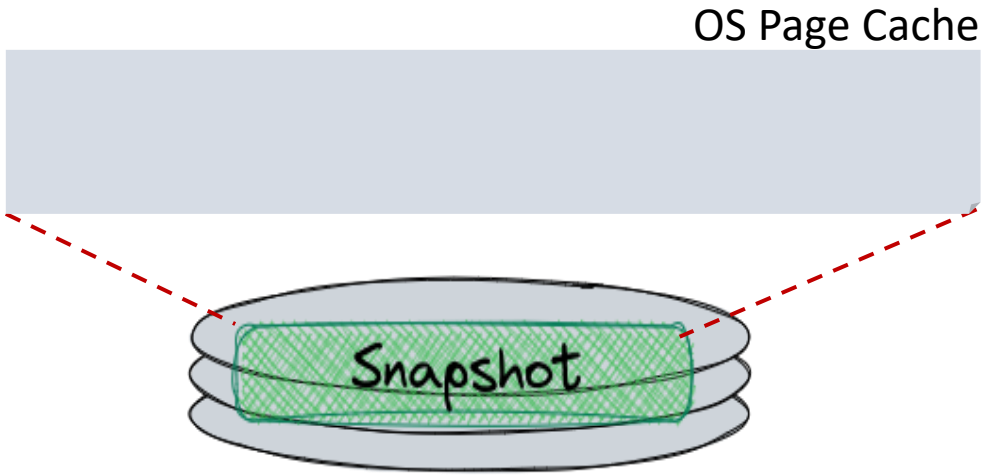
Readahead to the rescue?



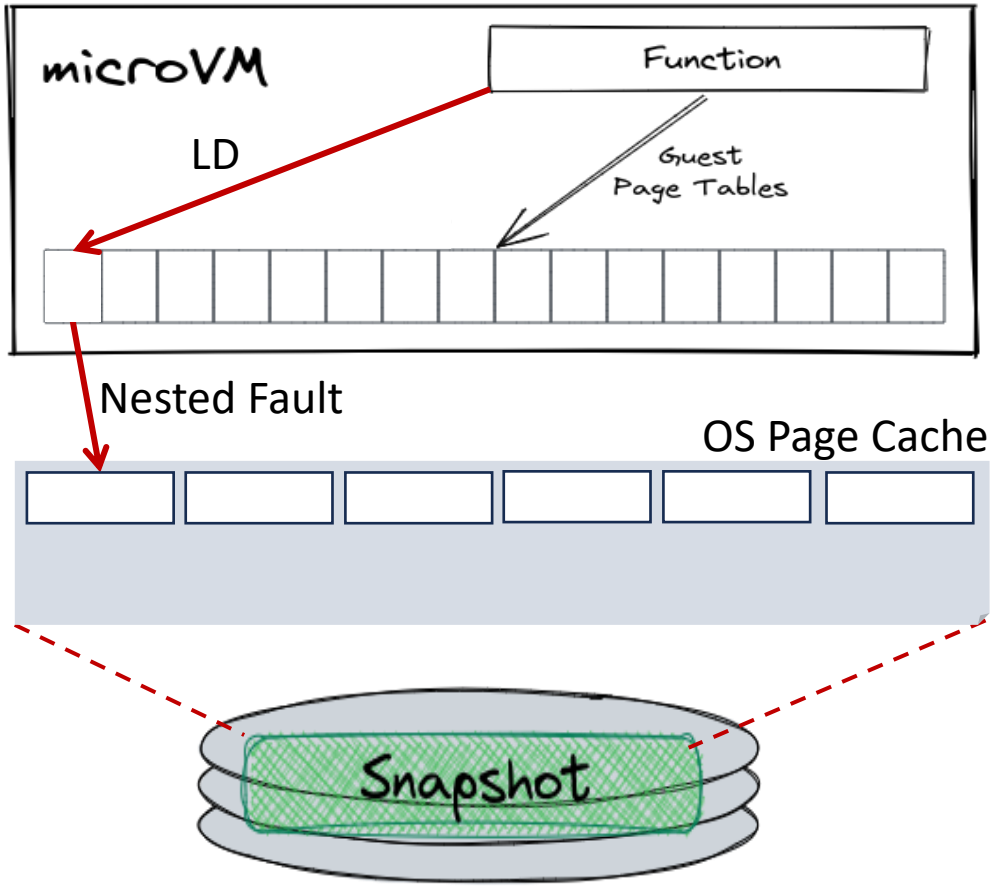


Readahead to the rescue?

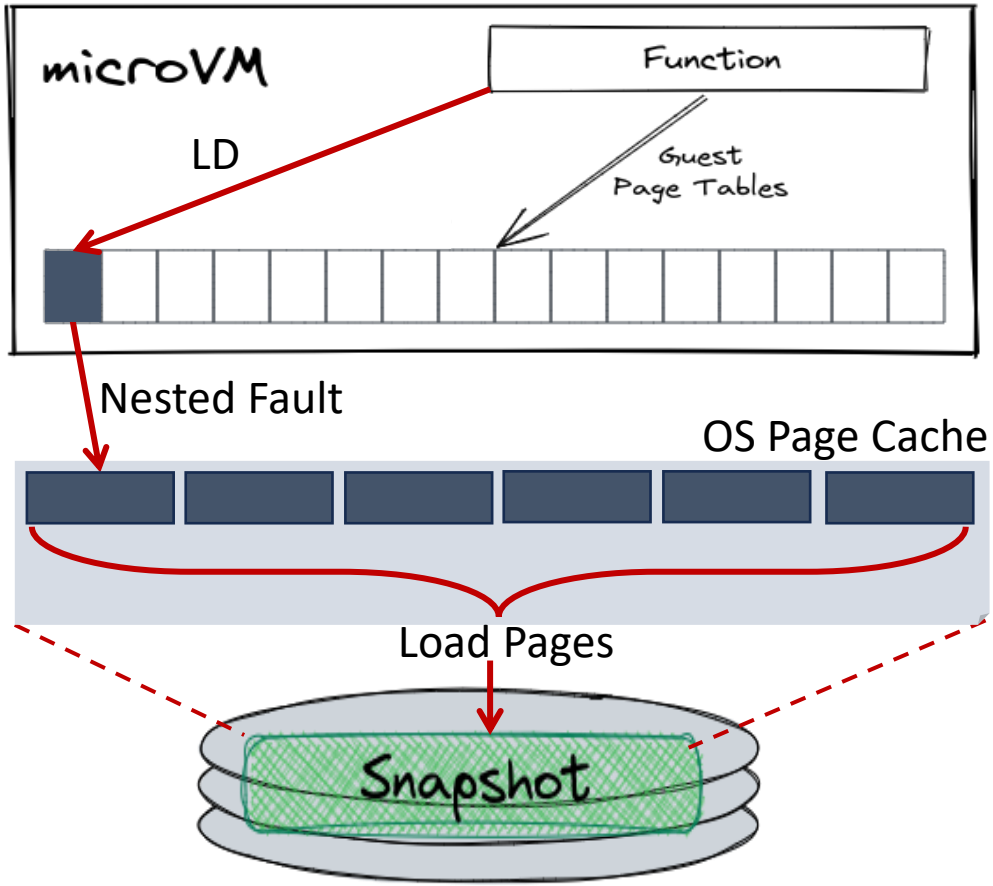




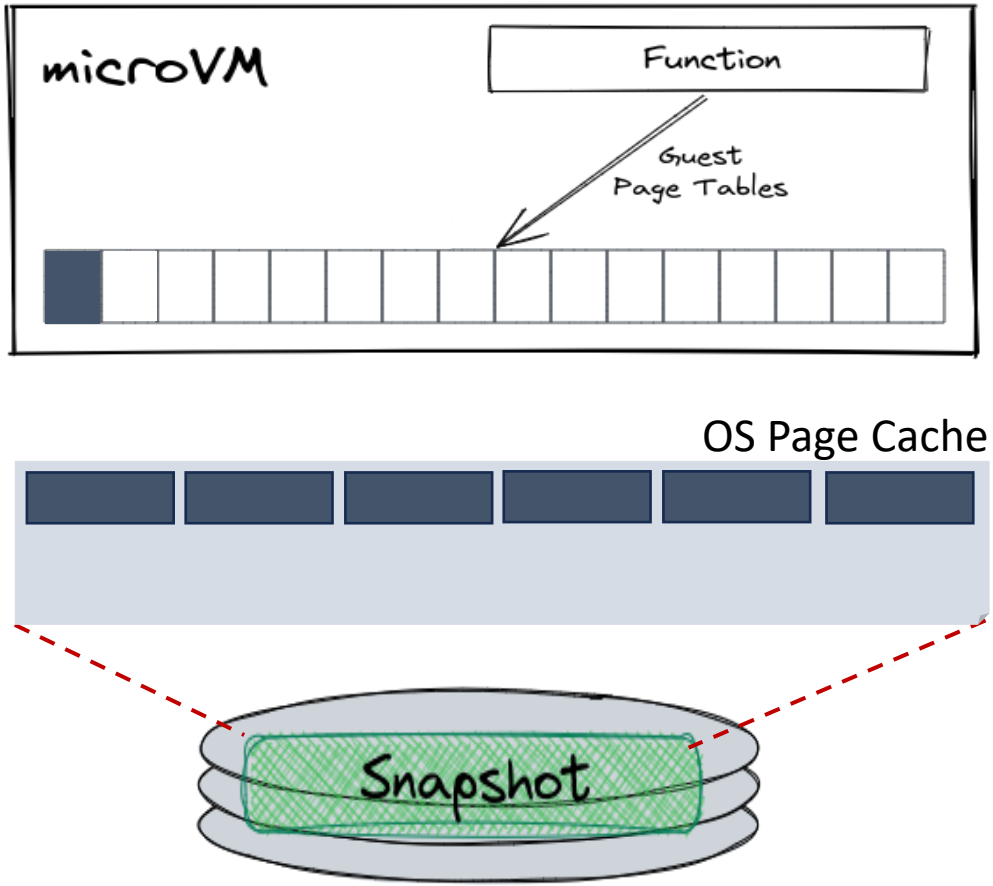
Readahead to the rescue?



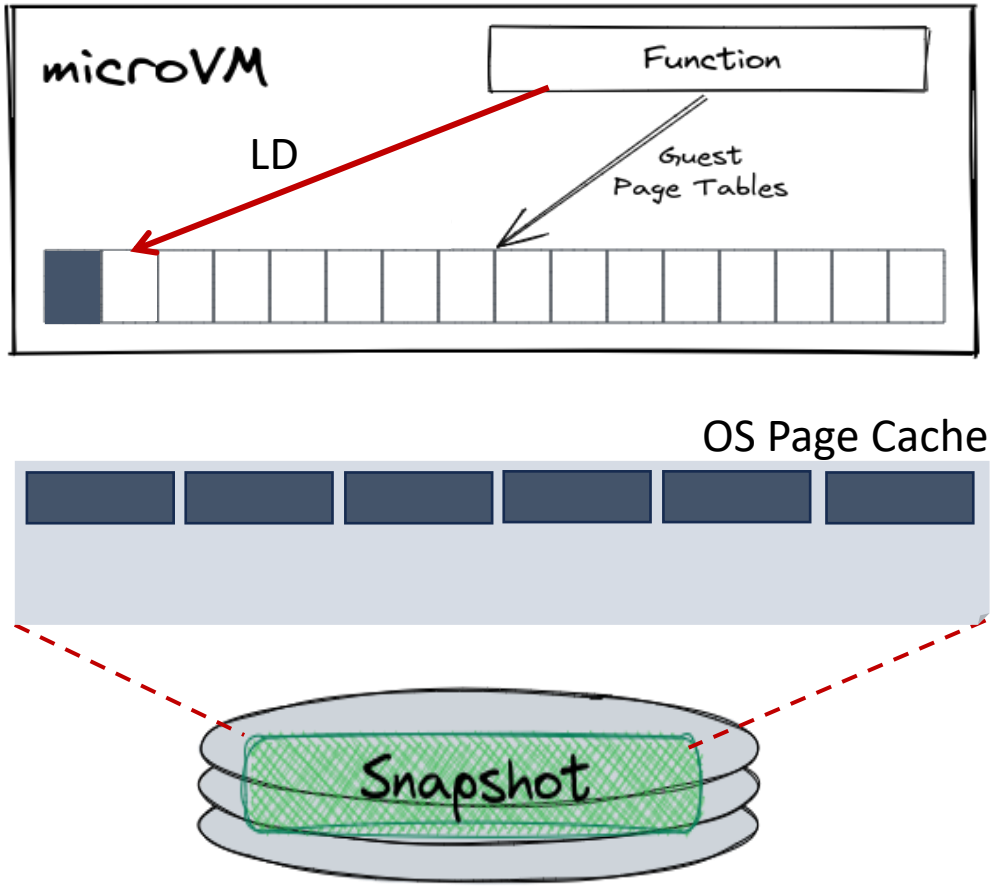
Readahead to the rescue?



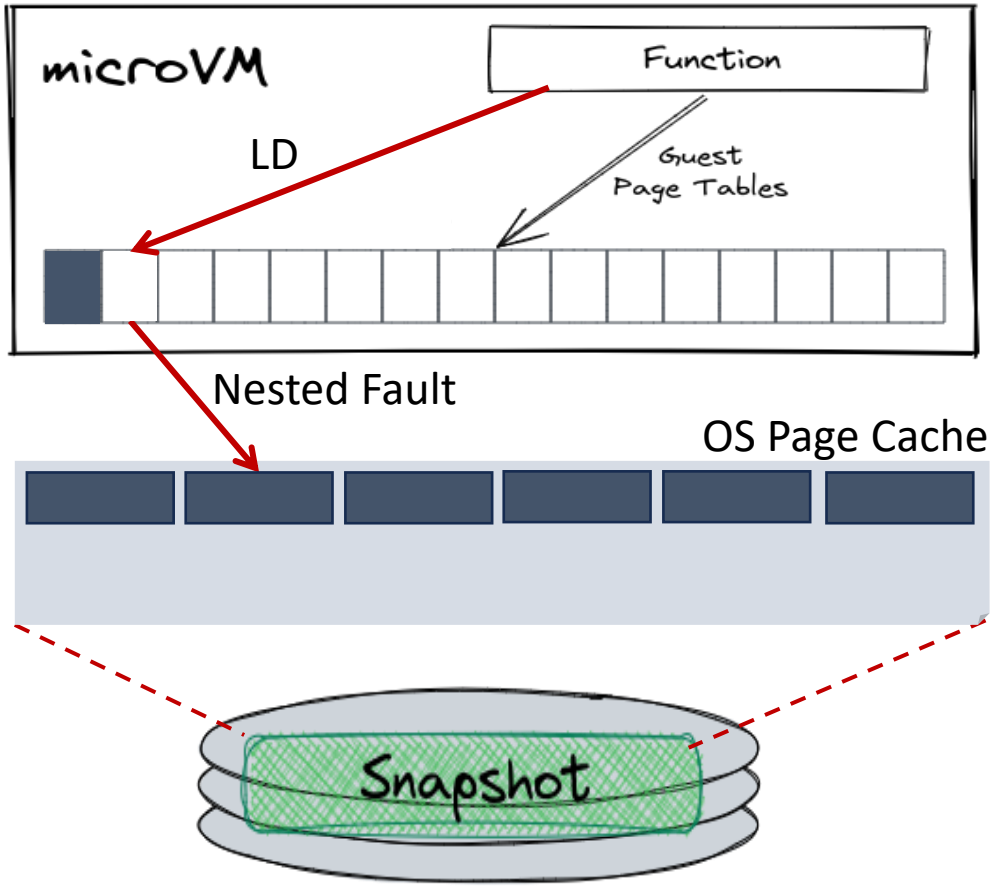
Readahead to the rescue?



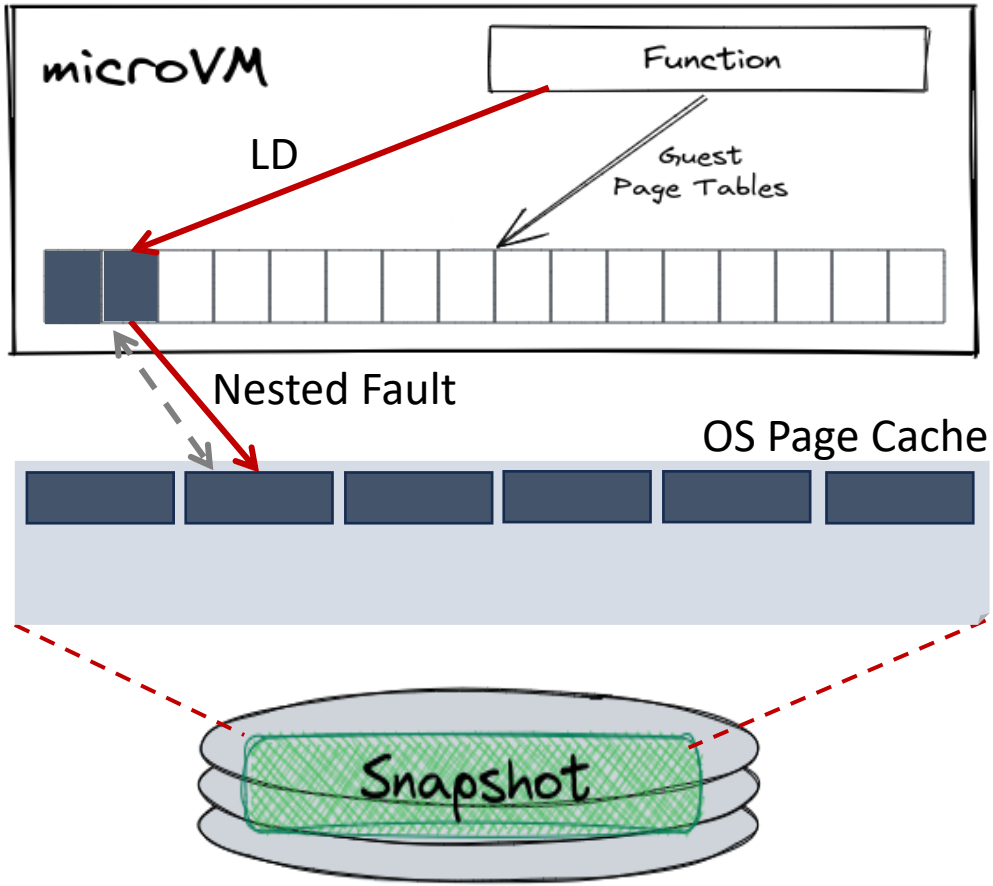
Readahead to the rescue?



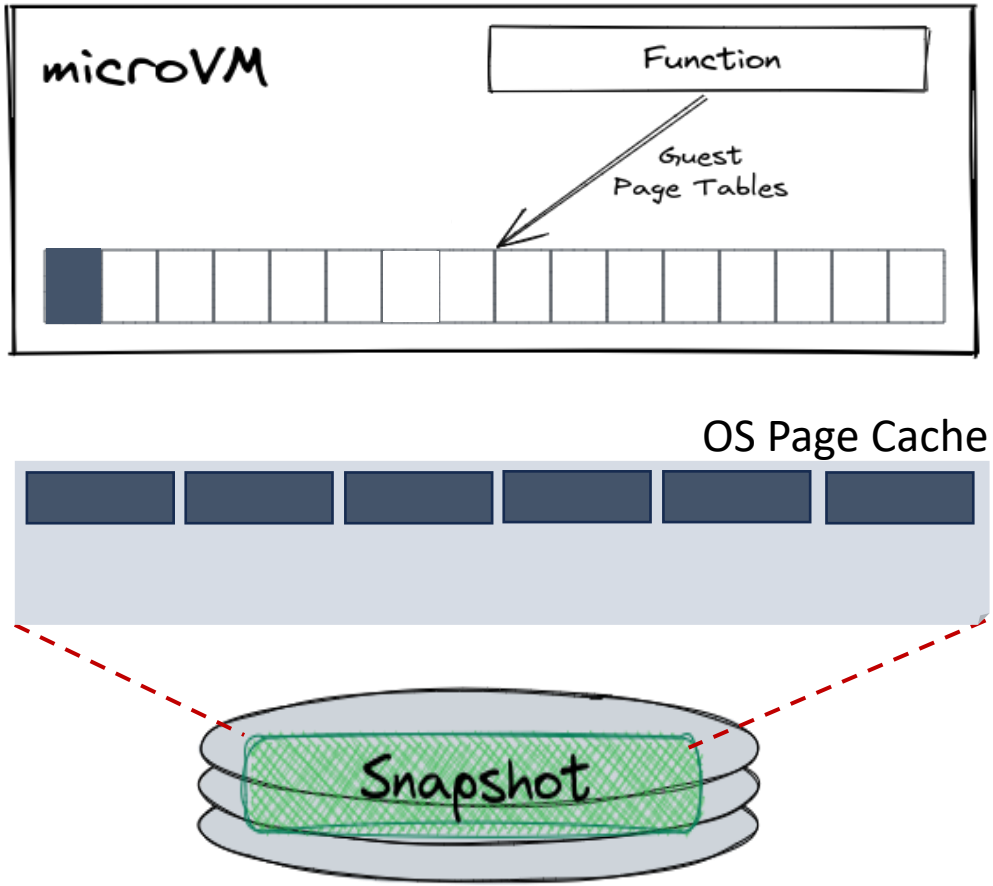
Readahead to the rescue?



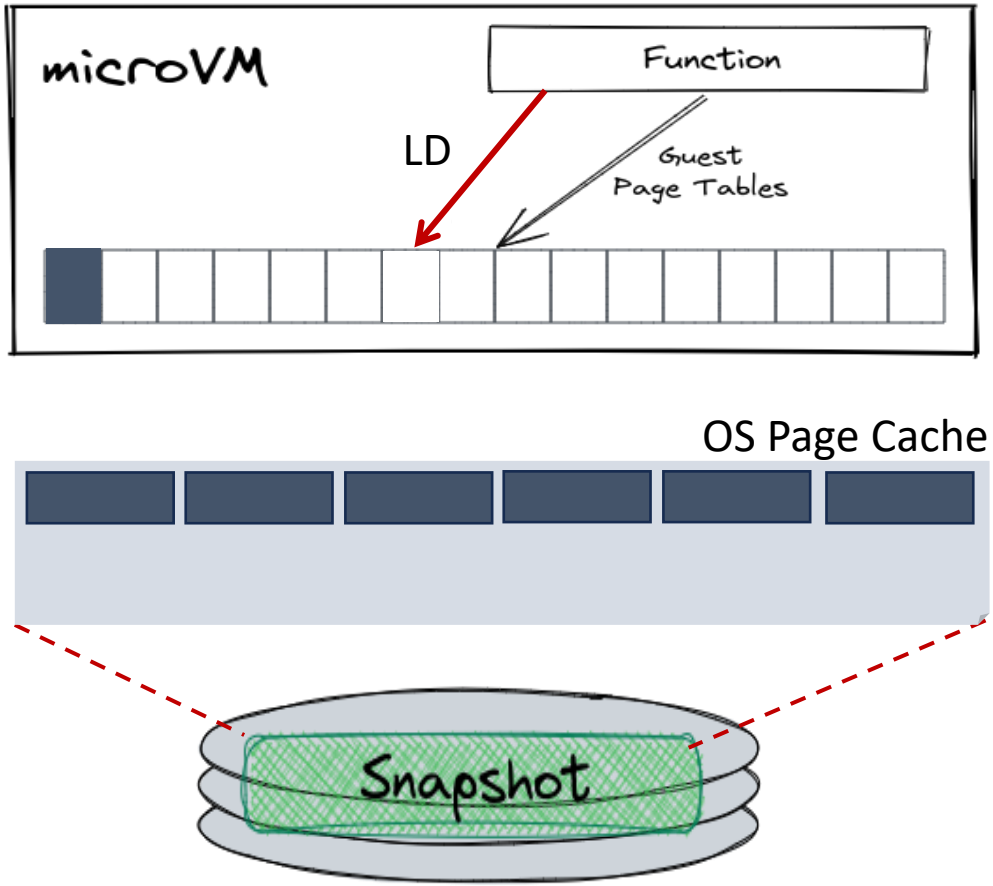
Readahead to the rescue?



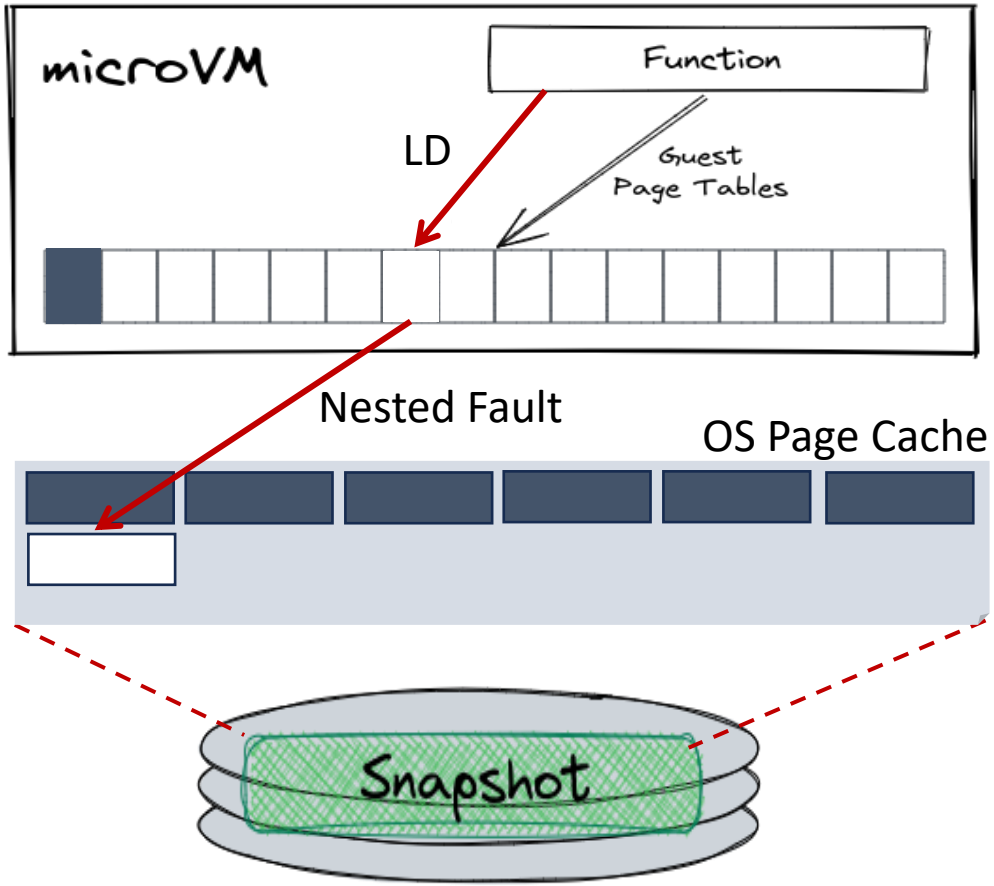
Readahead to the rescue?



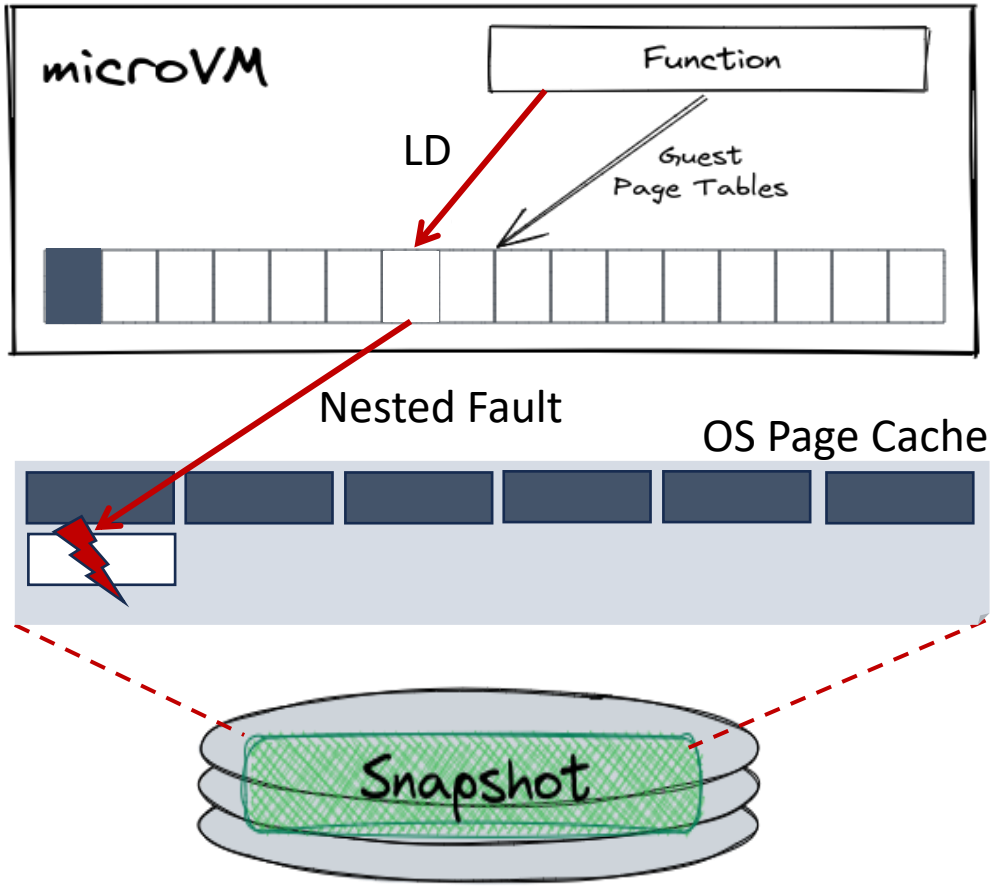
Readahead to the rescue?



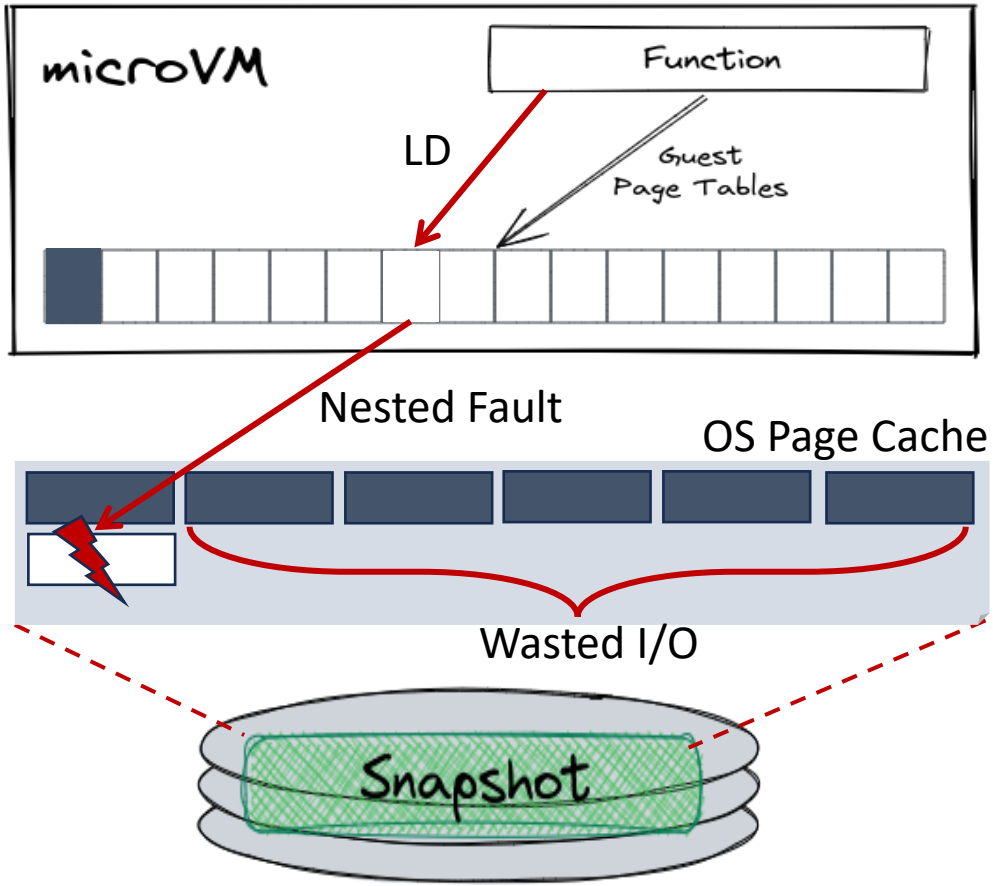
Readahead to the rescue?



Readahead to the rescue?



Readahead to the rescue?



Outline

→ FaaS Snapshotting

→ Working Set Prefetching

→ SnapBPF

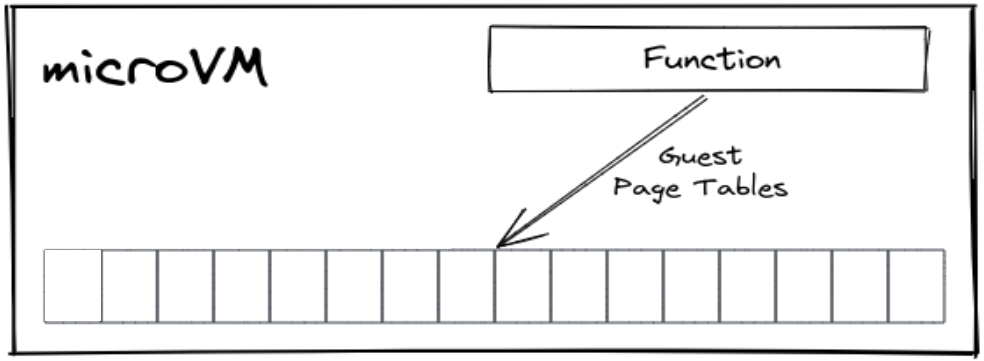
i. eBPF capture and prefetch

ii. PV free page filtering

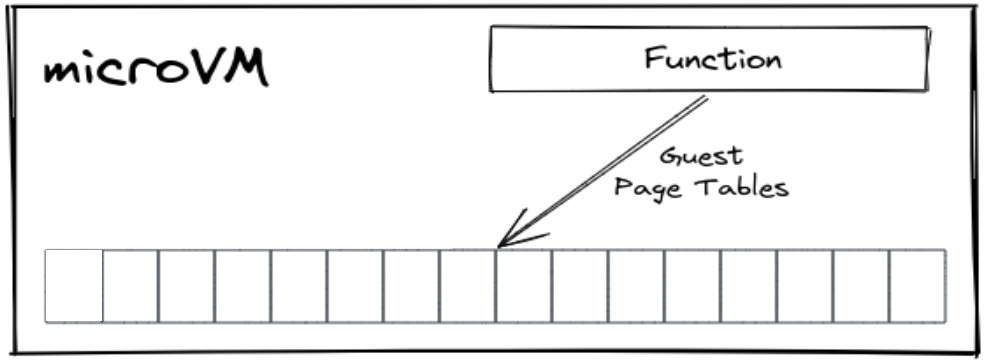
→ Evaluation

→ Conclusion

Working Set Prefetching: REAP / Faast



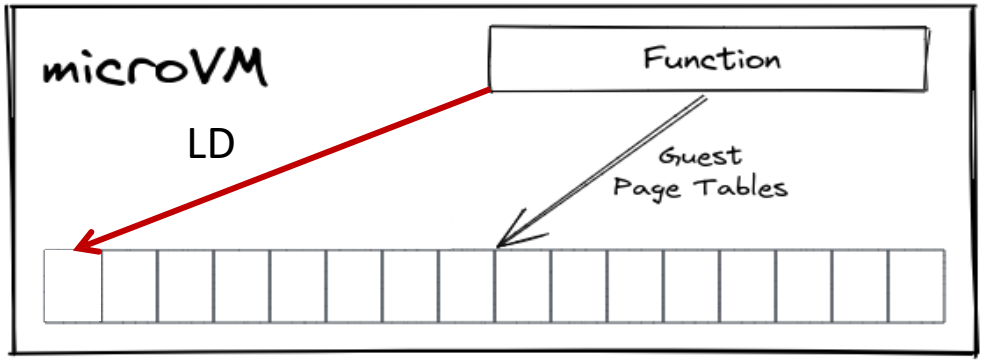
Working Set Prefetching: REAP / Faast



Userfaultfd
Agent

Anonymous Memory

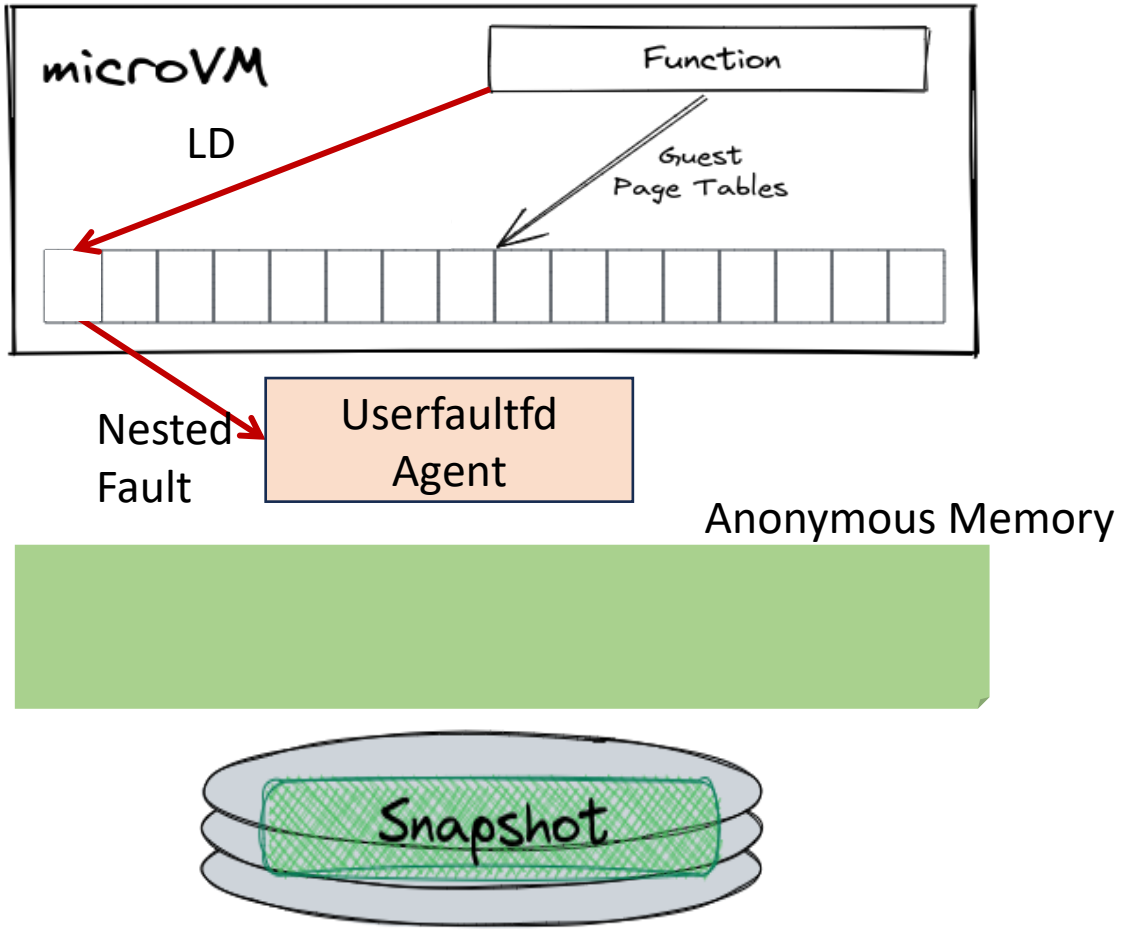
Working Set Prefetching: REAP / Faast



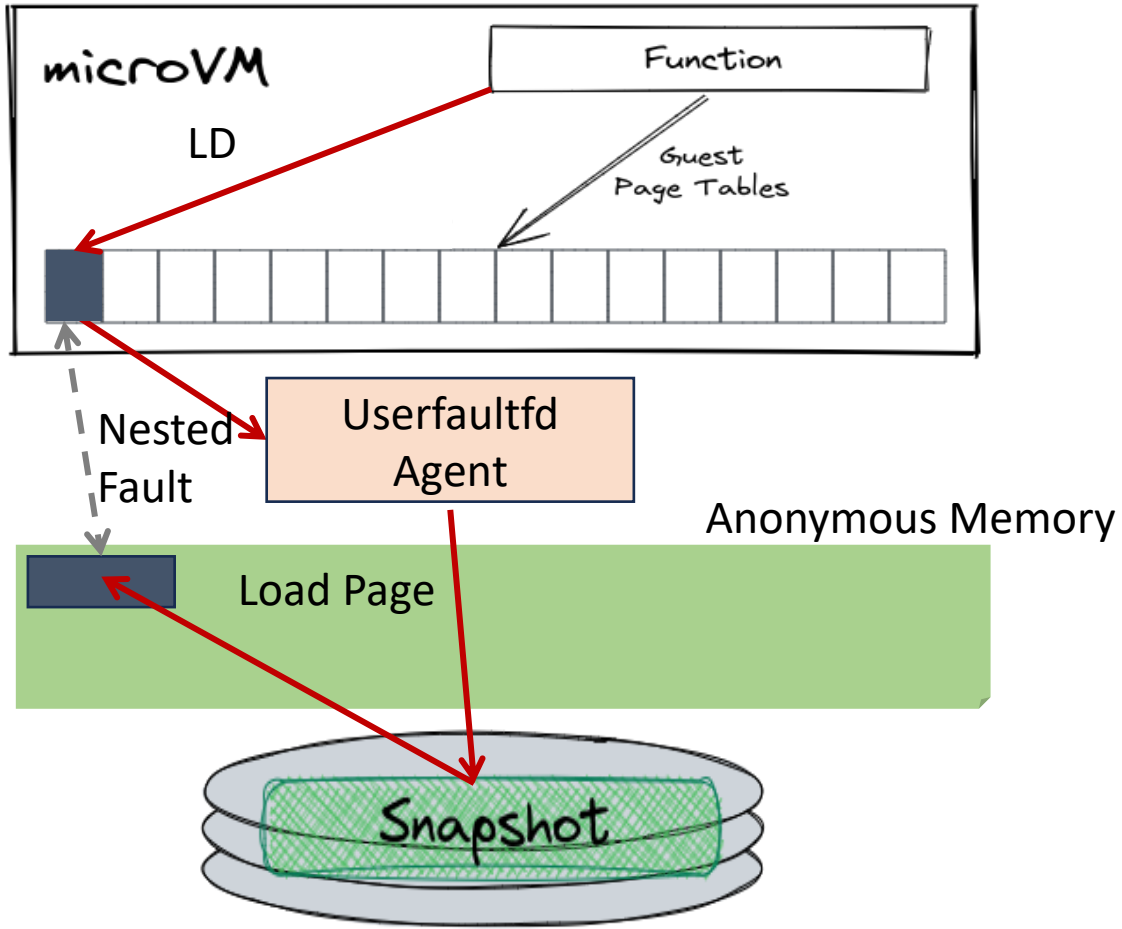
Userfaultfd
Agent

Anonymous Memory

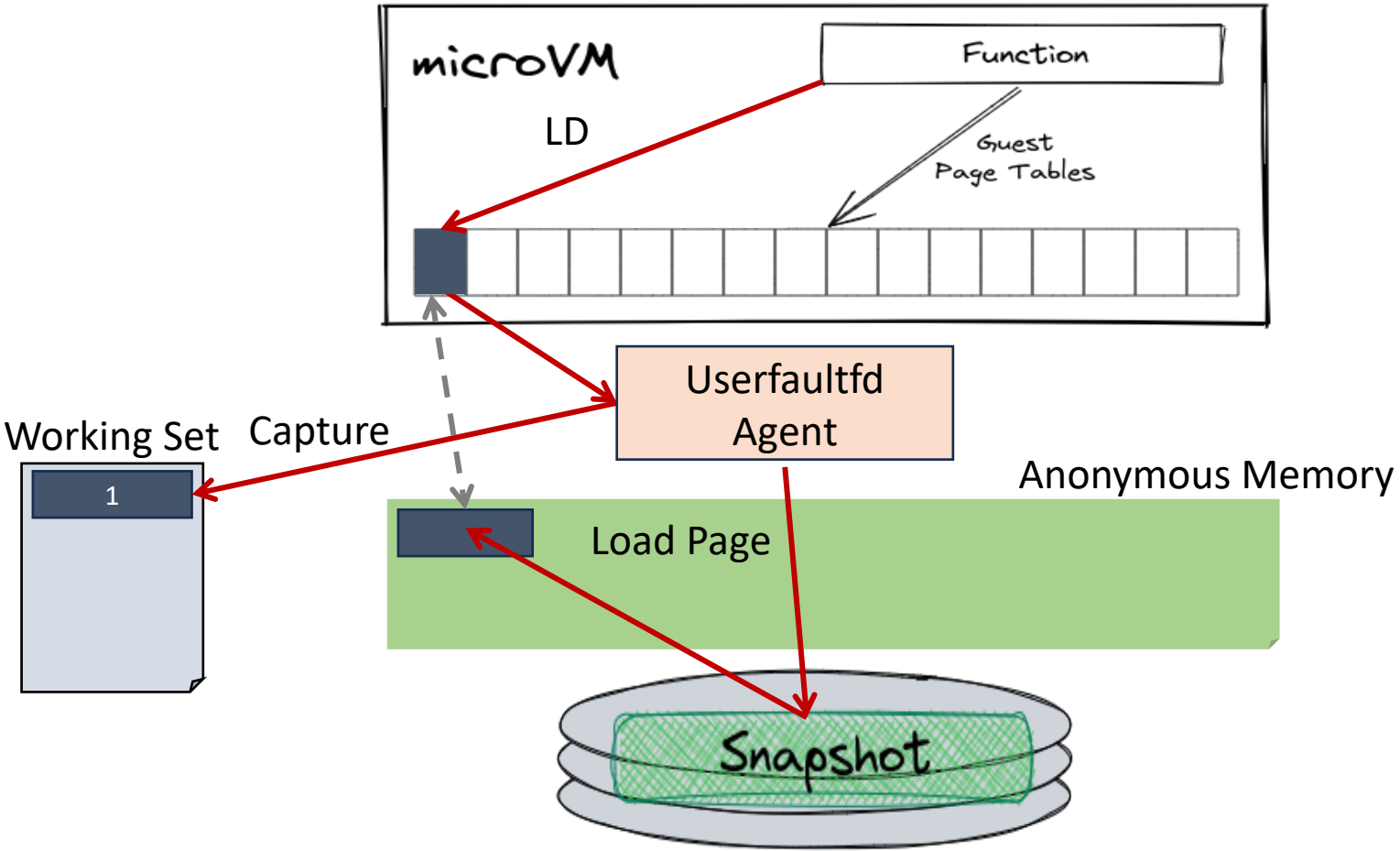
Working Set Prefetching: REAP / Faast



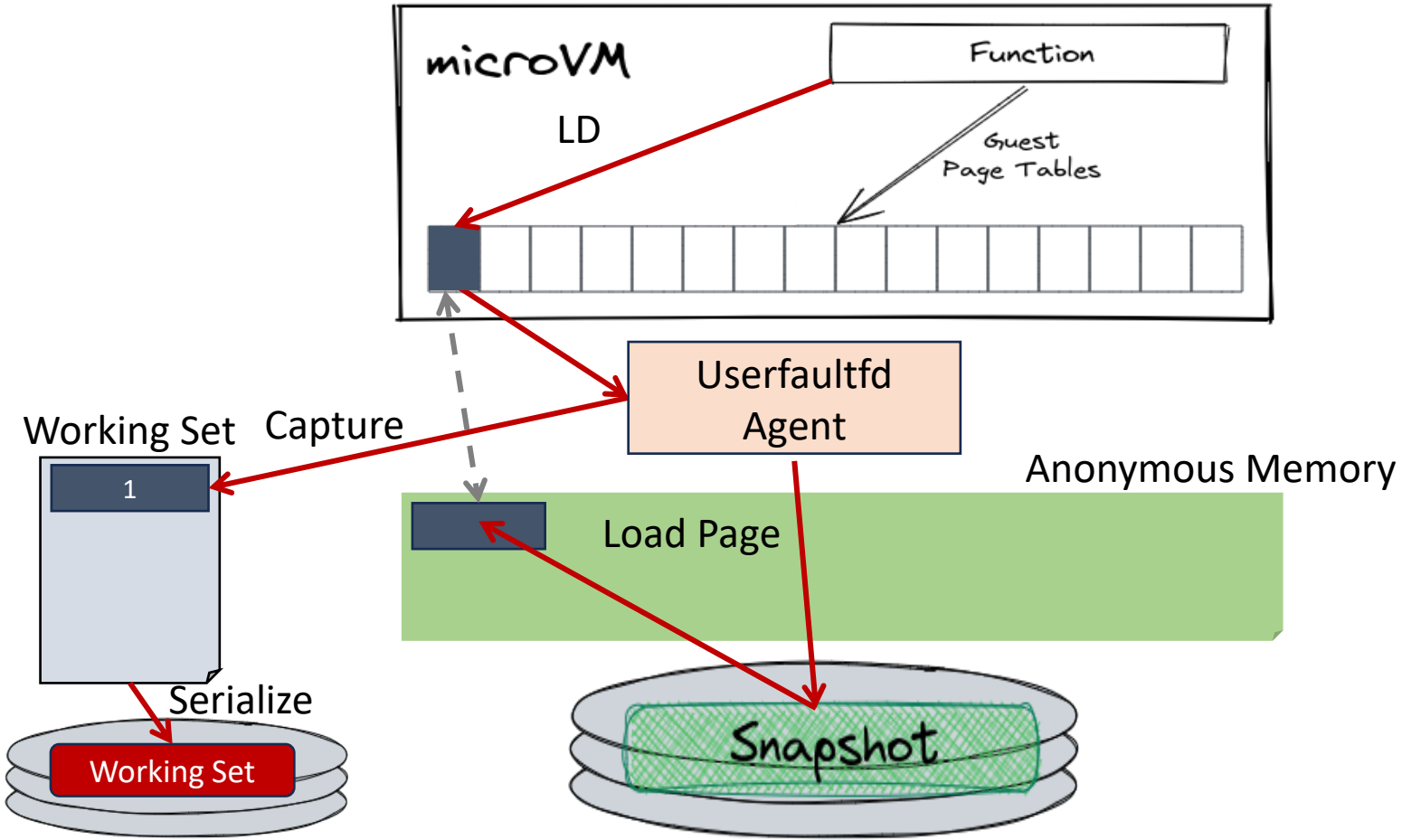
Working Set Prefetching: REAP / Faast



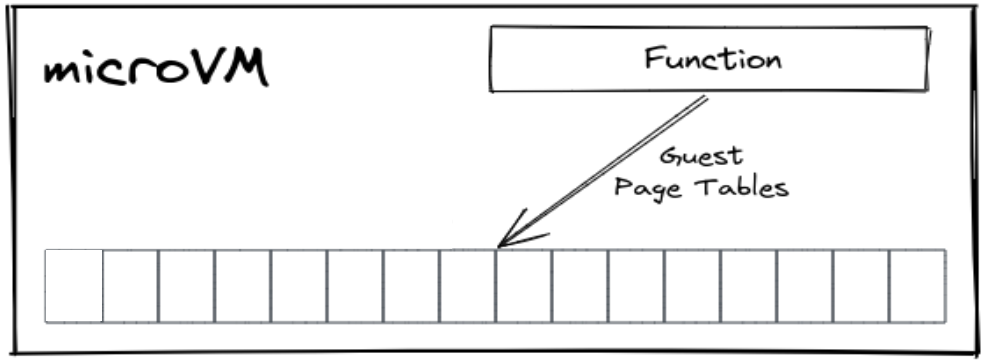
Working Set Prefetching: REAP / Faast



Working Set Prefetching: REAP / Faast

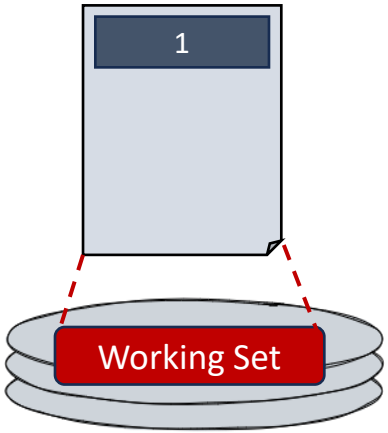


Working Set Prefetching: REAP / Faast



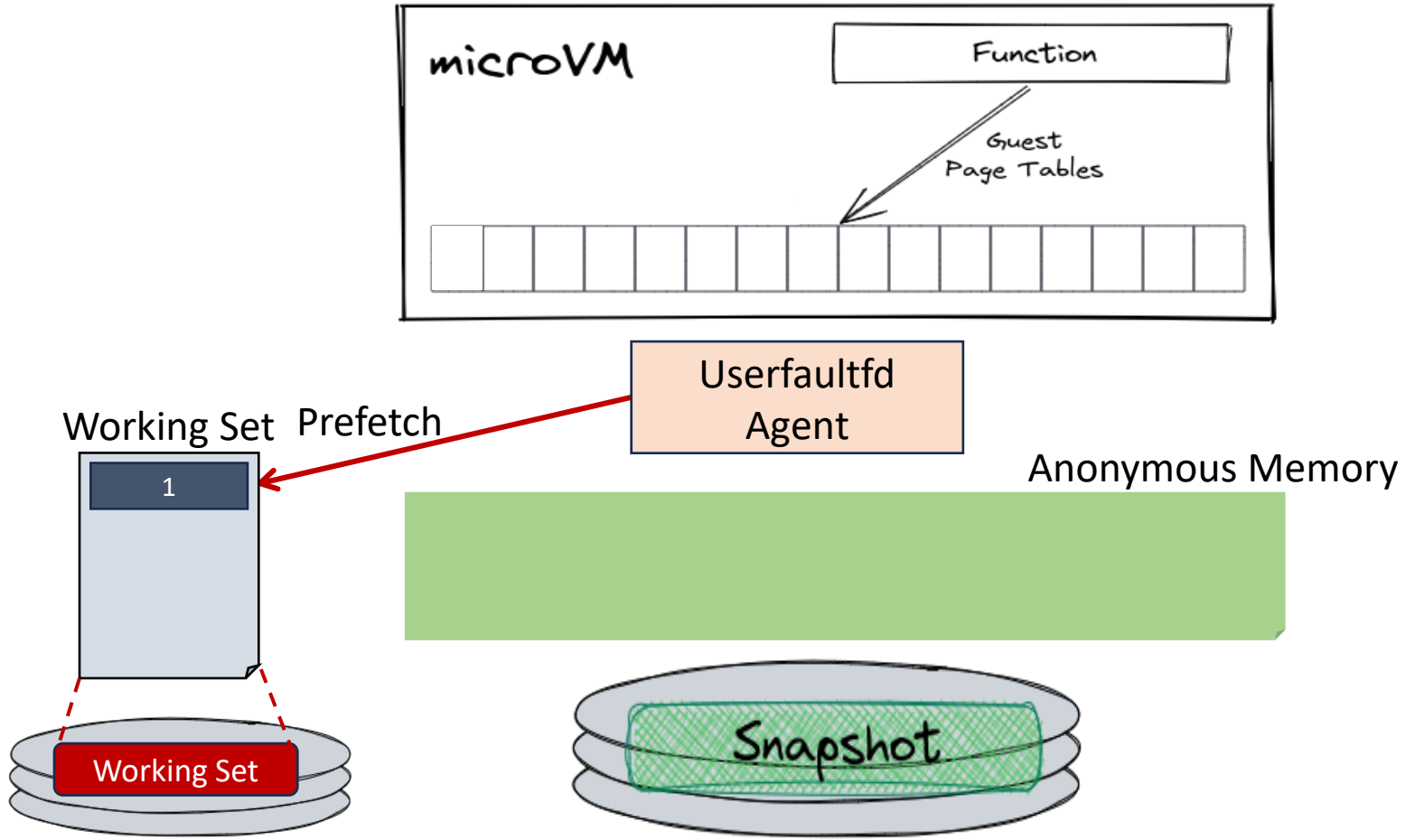
Userfaultfd
Agent

Working Set

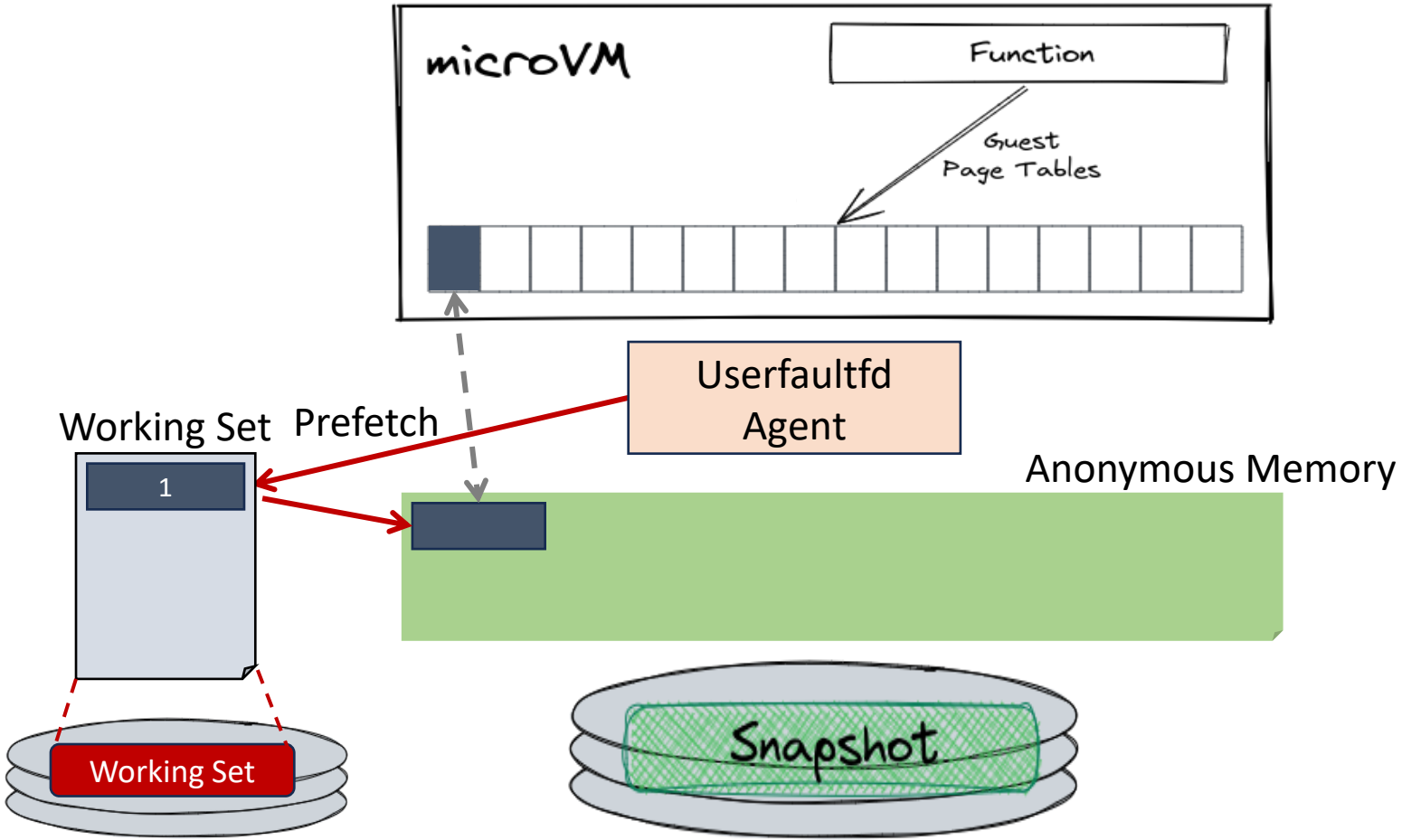


Anonymous Memory

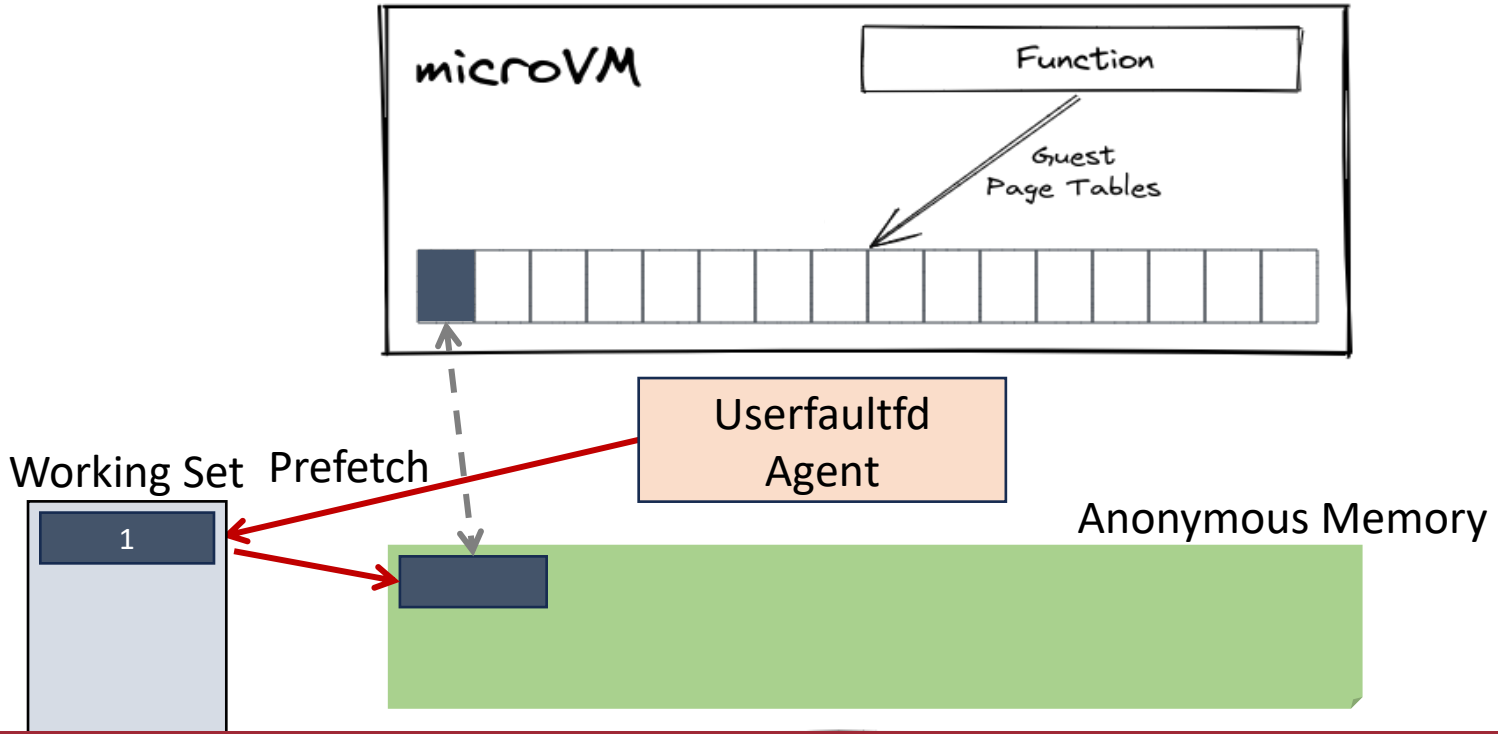
Working Set Prefetching: REAP / Faast



Working Set Prefetching: REAP / Faast

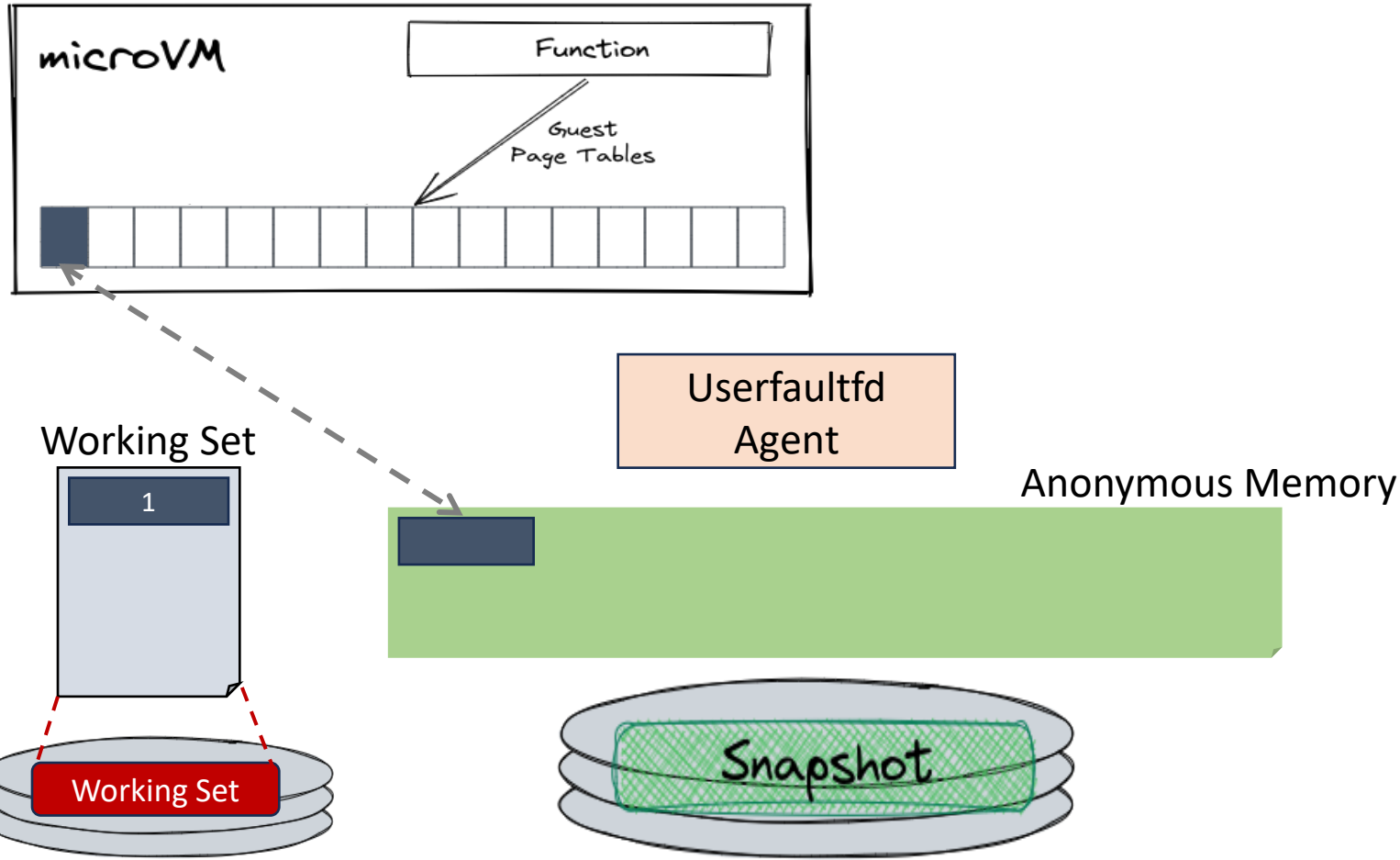


Working Set Prefetching: REAP / Faast

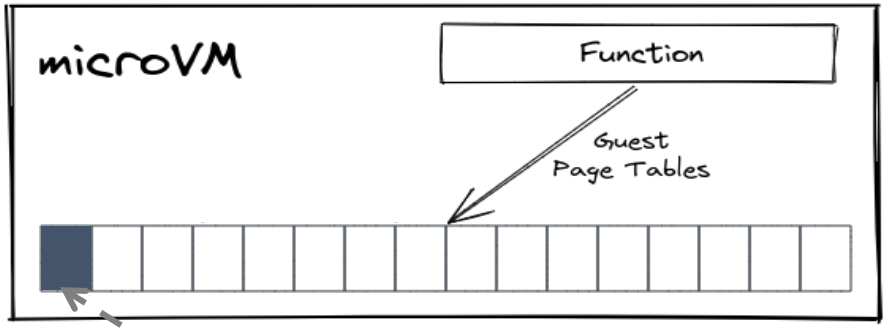


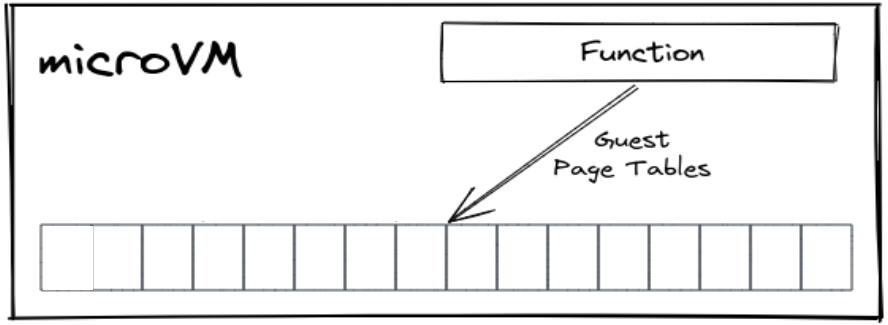
User <-> Kernel transition overhead

Working Set Prefetching: REAP / Faast



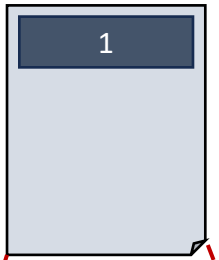
Working Set Prefetching: REAP / Faast





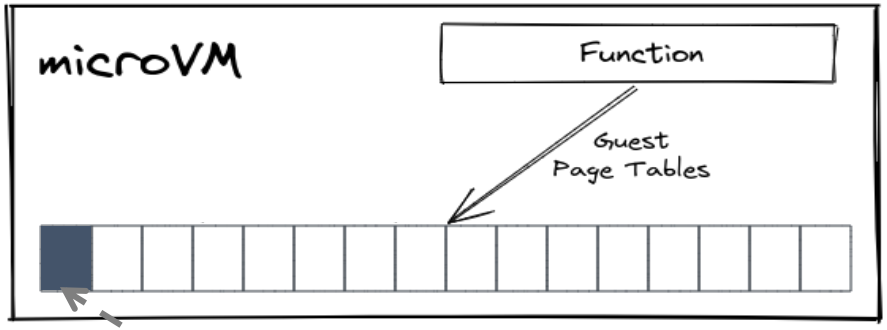
Userfaultfd
Agent

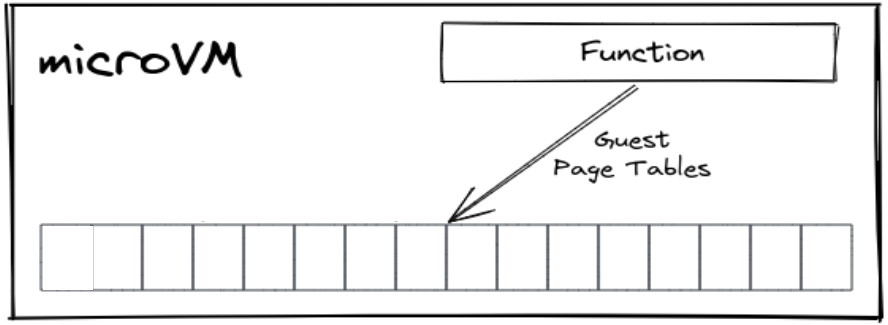
Working Set

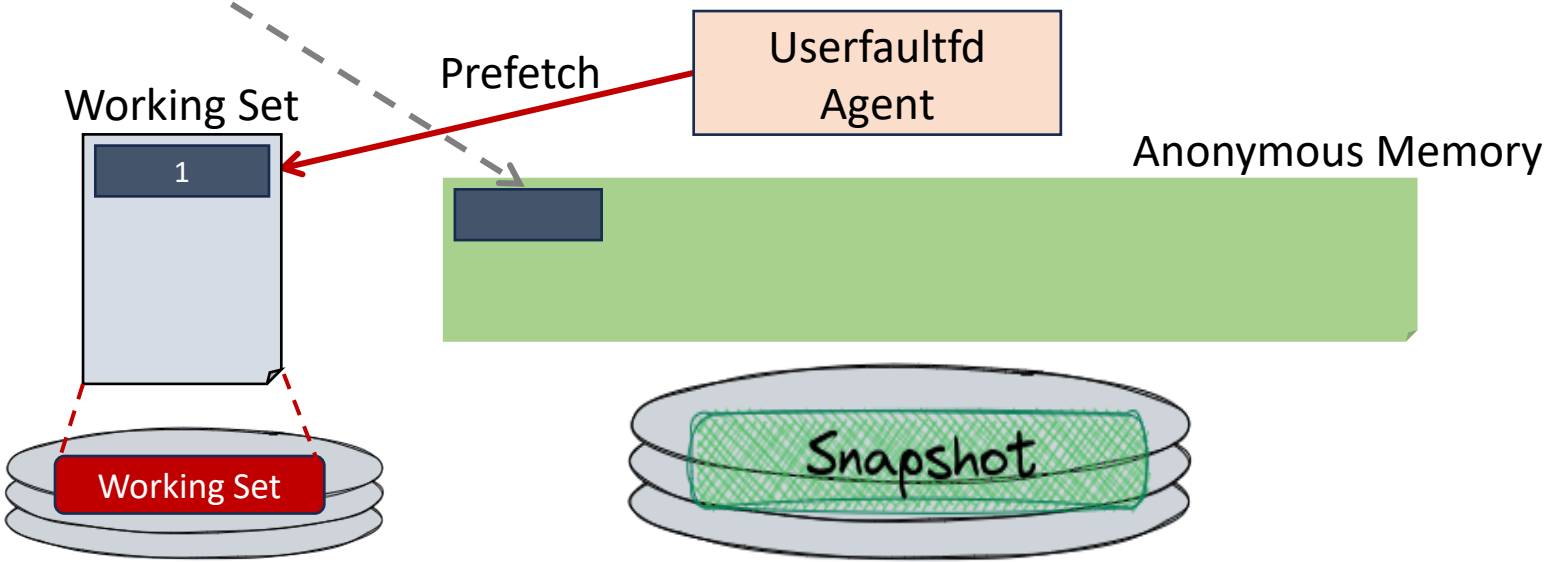


Anonymous Memory

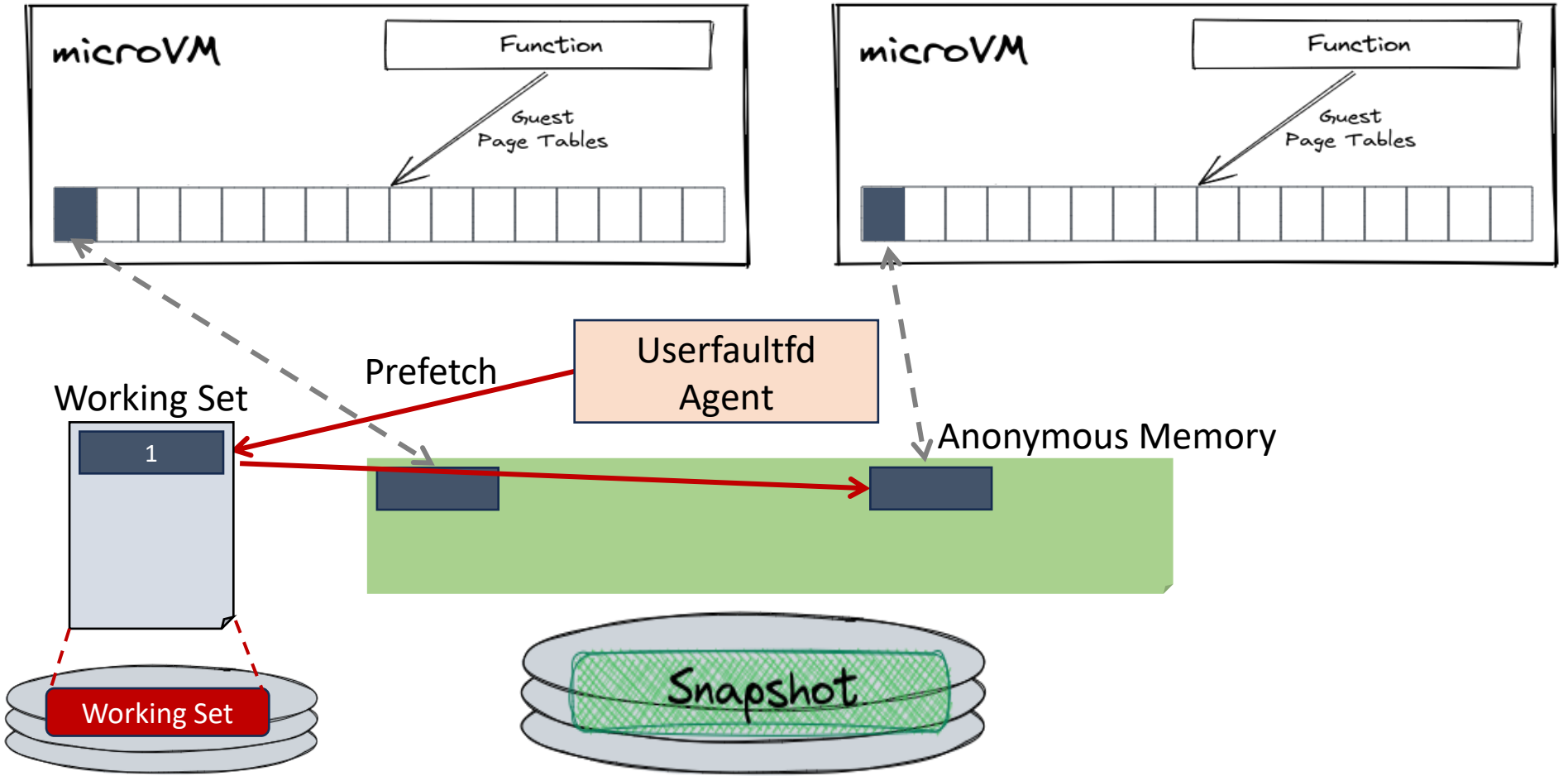
Working Set Prefetching: REAP / Faast



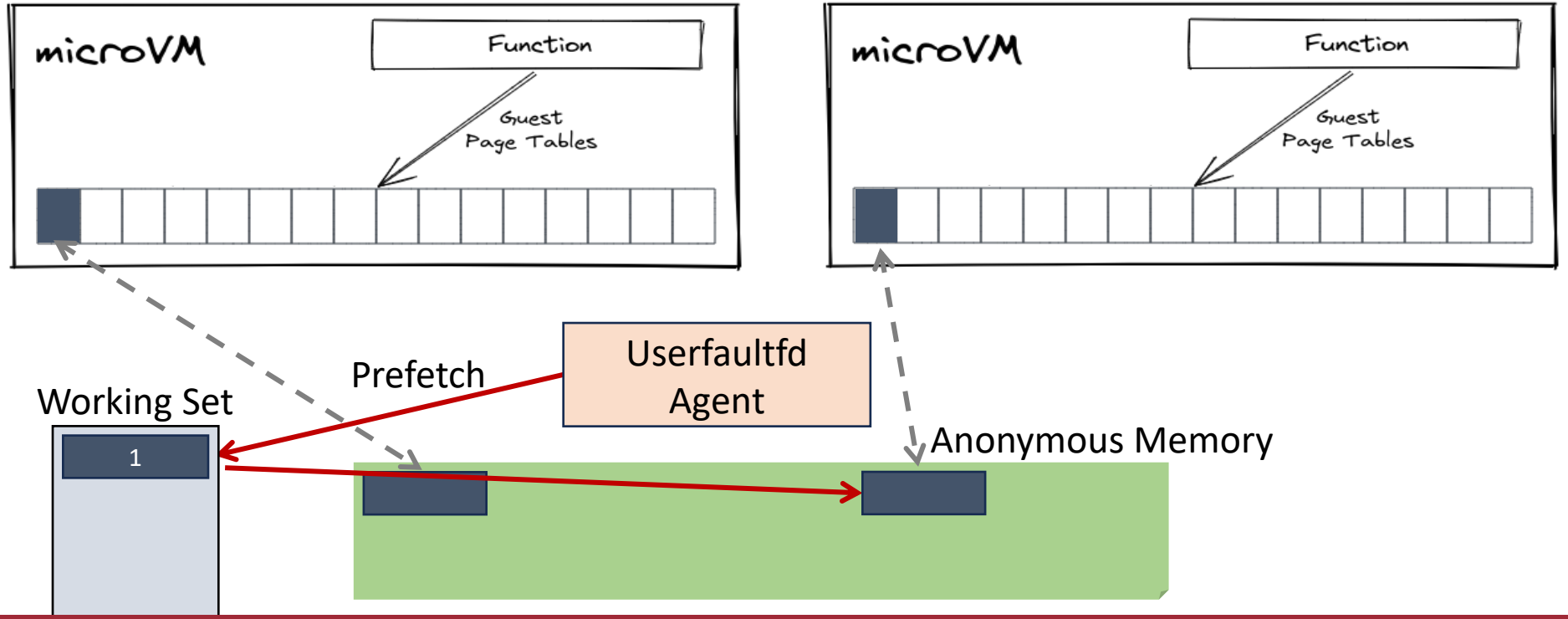




Working Set Prefetching: REAP / Faast

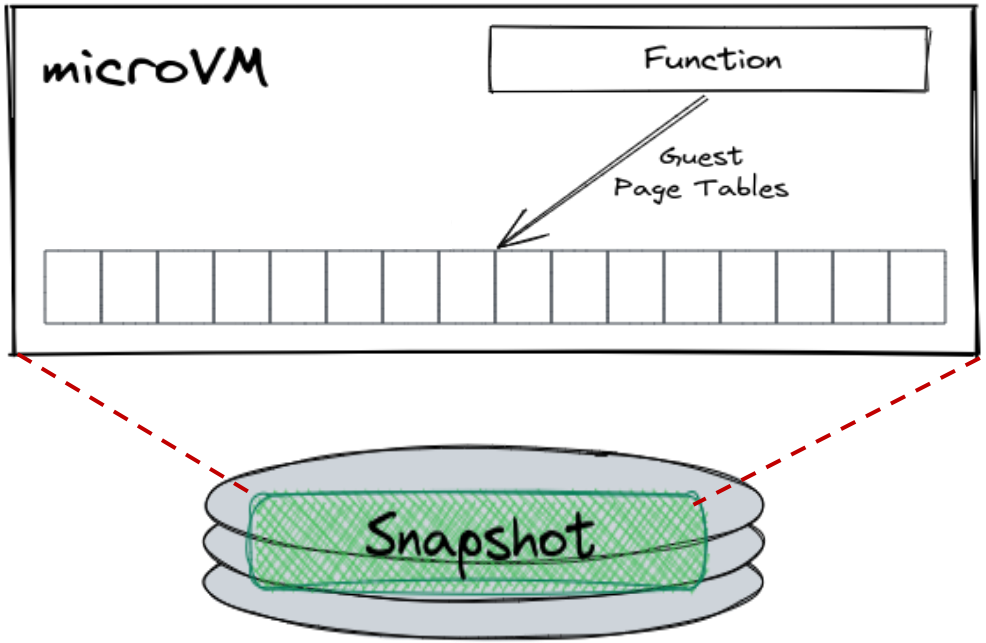


Working Set Prefetching: REAP / Faast

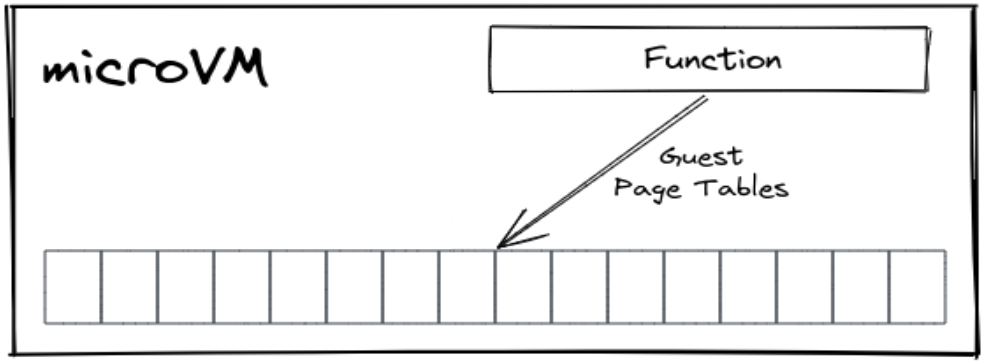


Bypasses page cache == No sharing

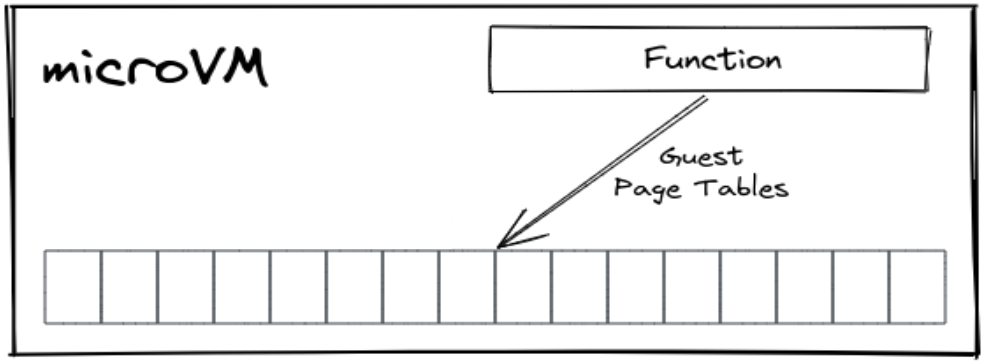
Working Set Prefetching: FaaSnap



Working Set Prefetching: FaaSnap

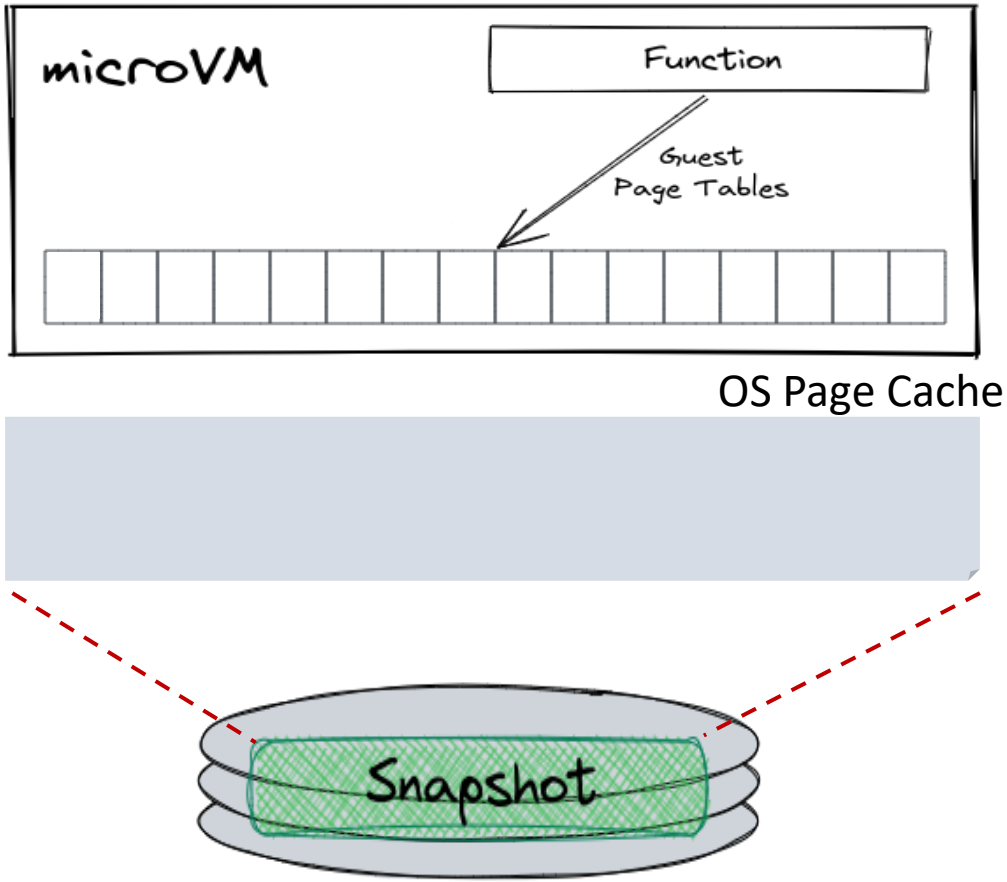


Working Set Prefetching: FaaSnap

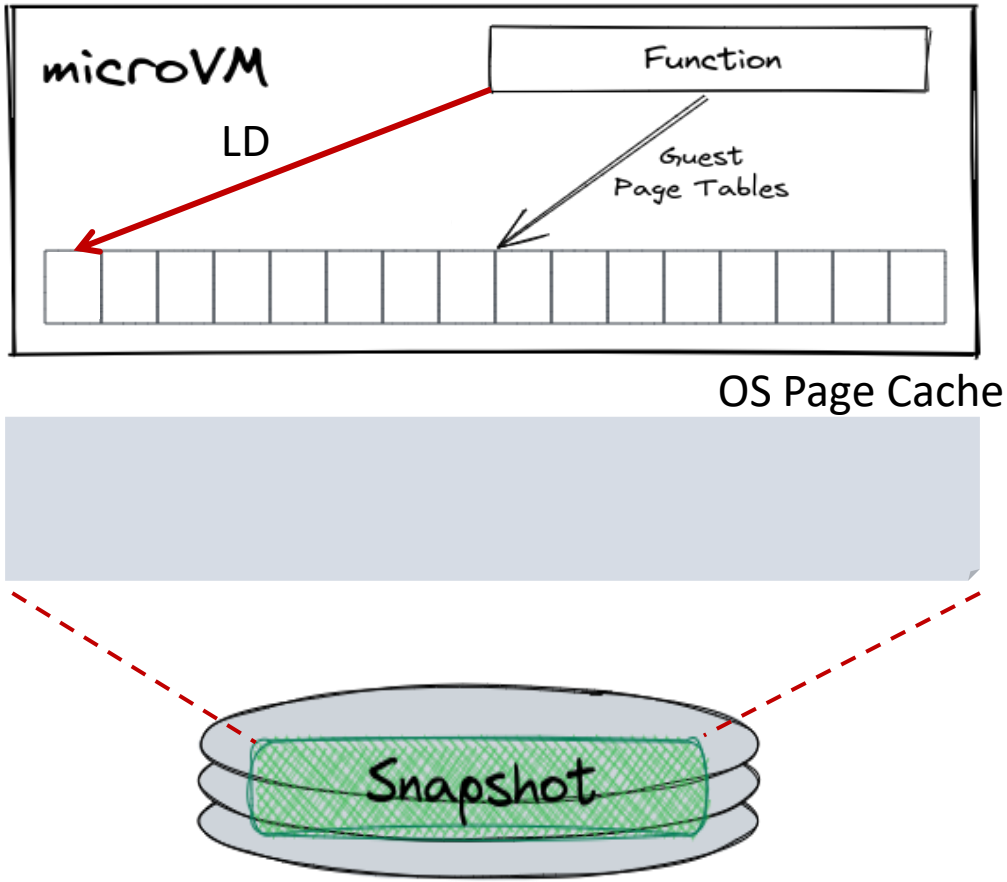


OS Page Cache

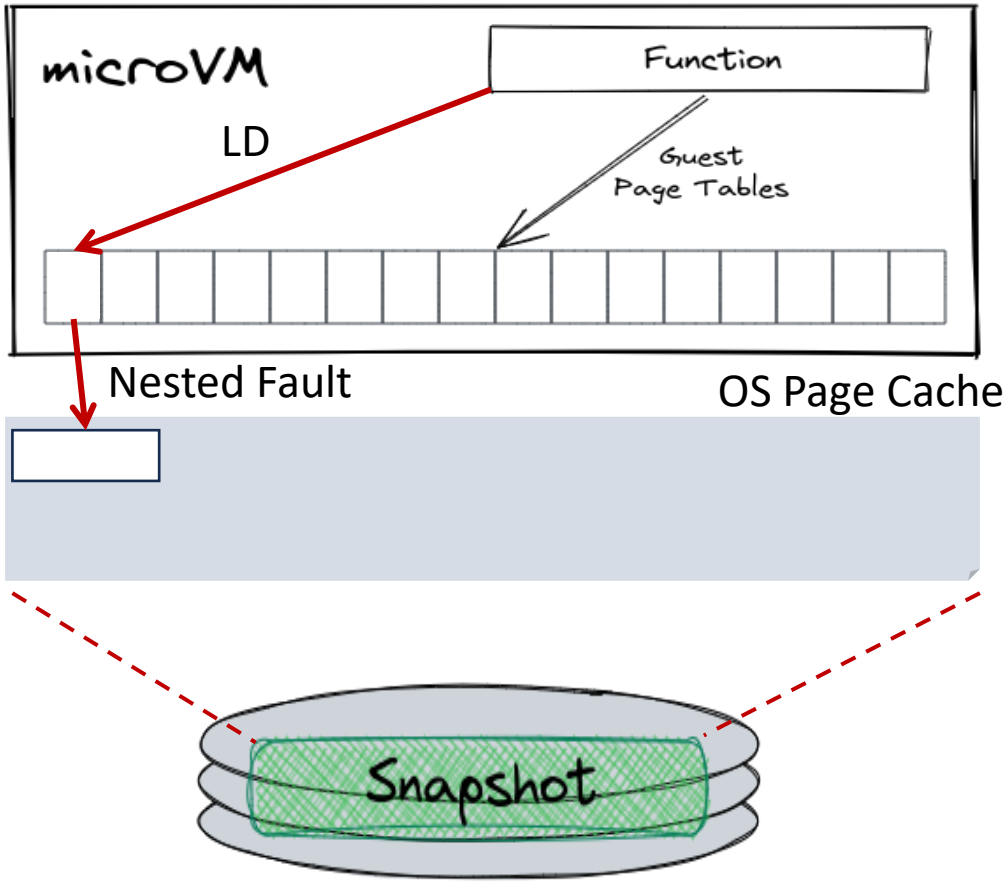
Working Set Prefetching: FaaSnap



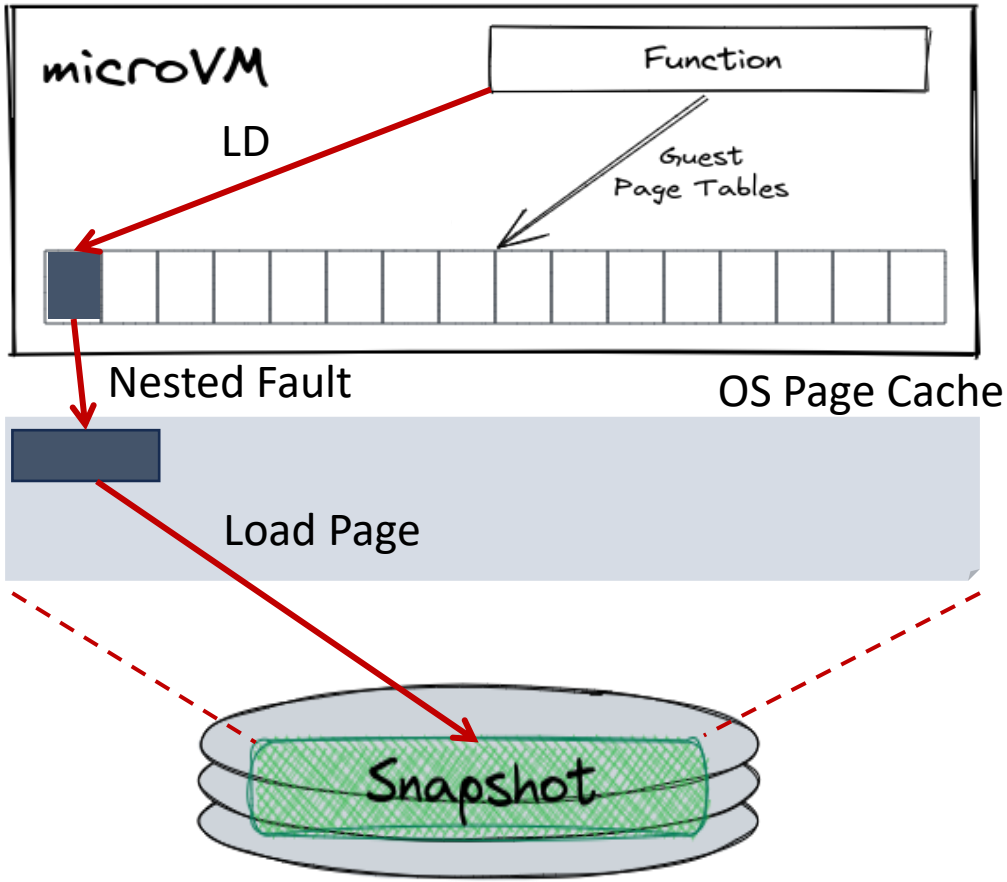
Working Set Prefetching: FaaSnap



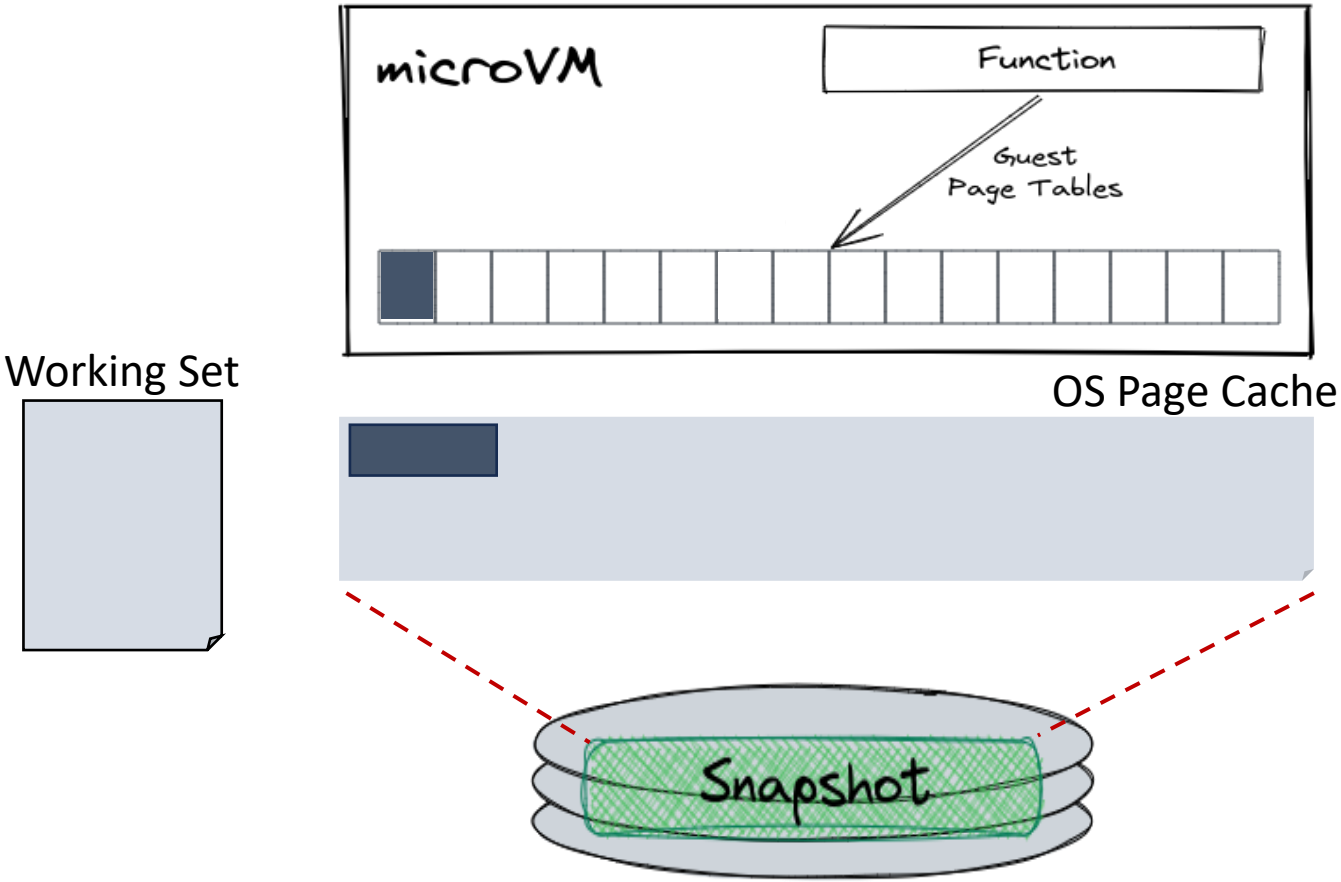
Working Set Prefetching: FaaSnap



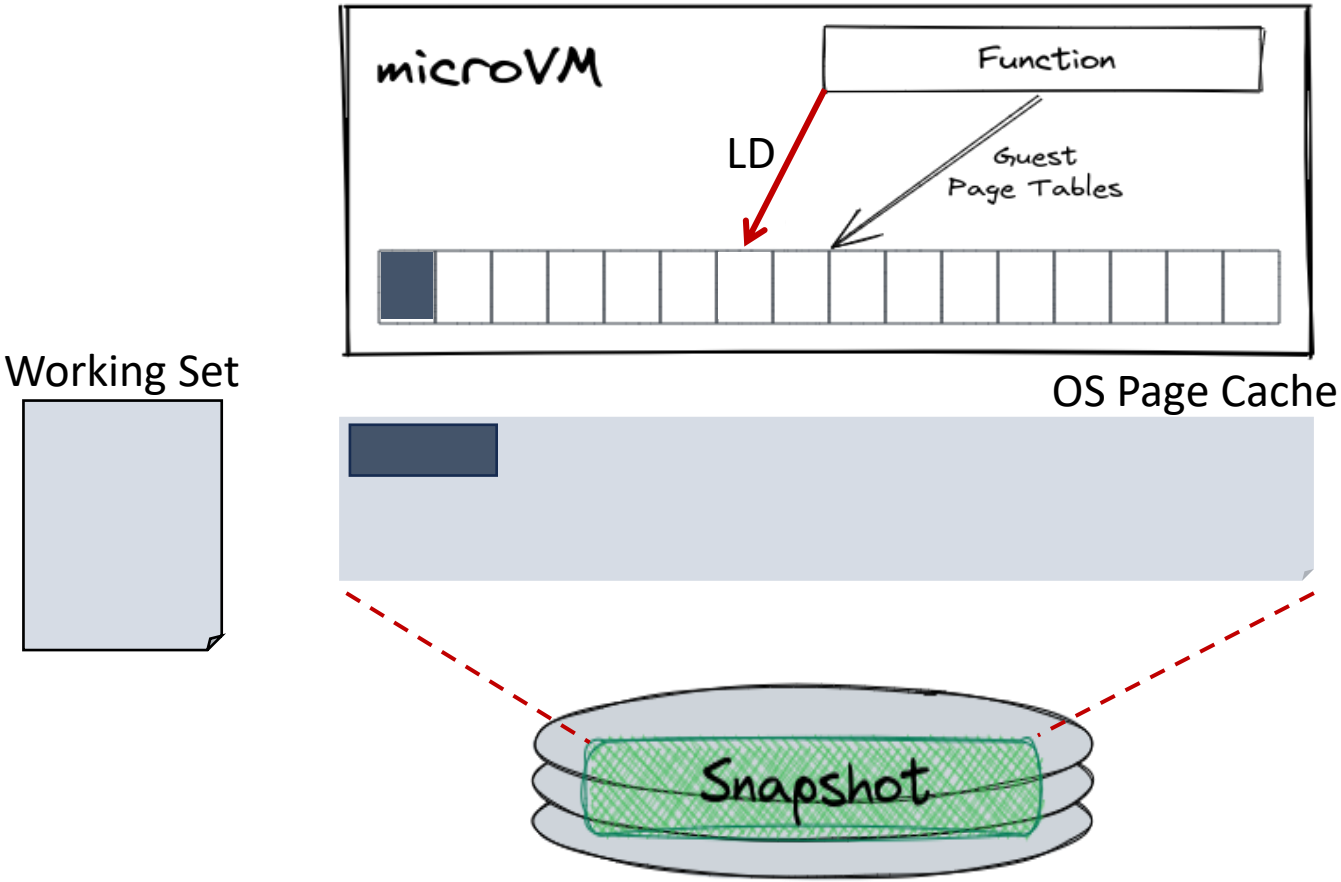
Working Set Prefetching: FaaSnap



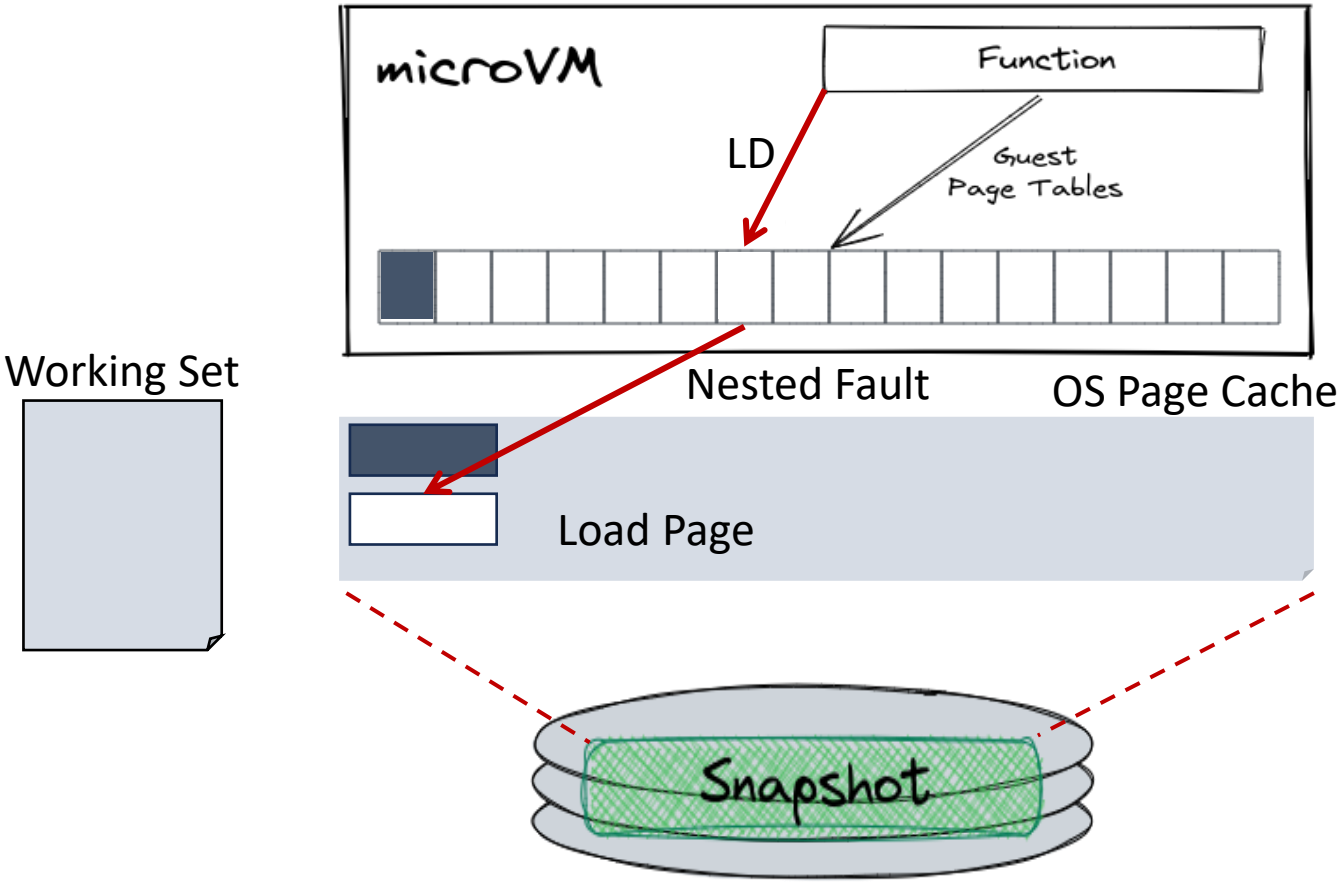
Working Set Prefetching: FaaSnap



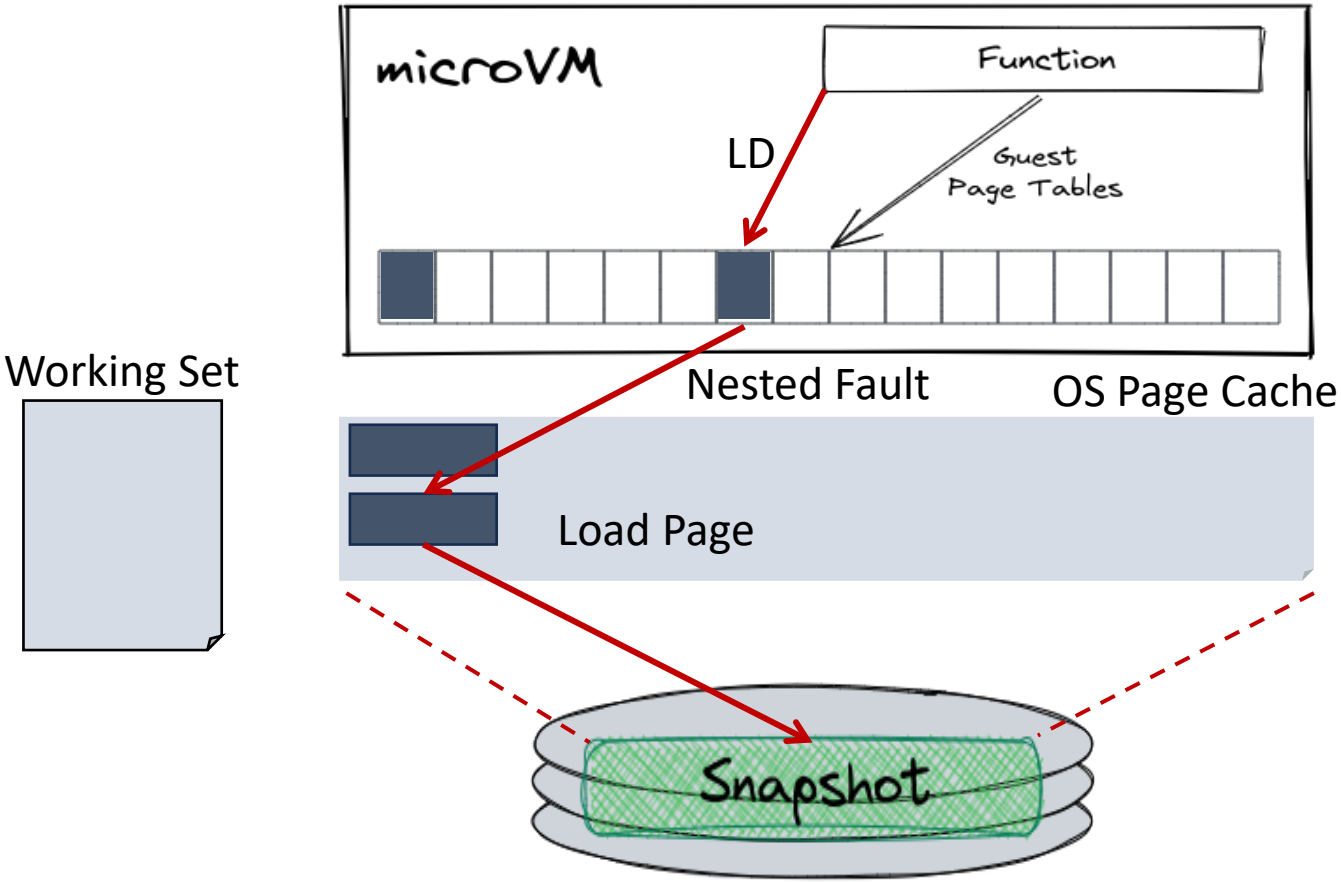
Working Set Prefetching: FaaSnap



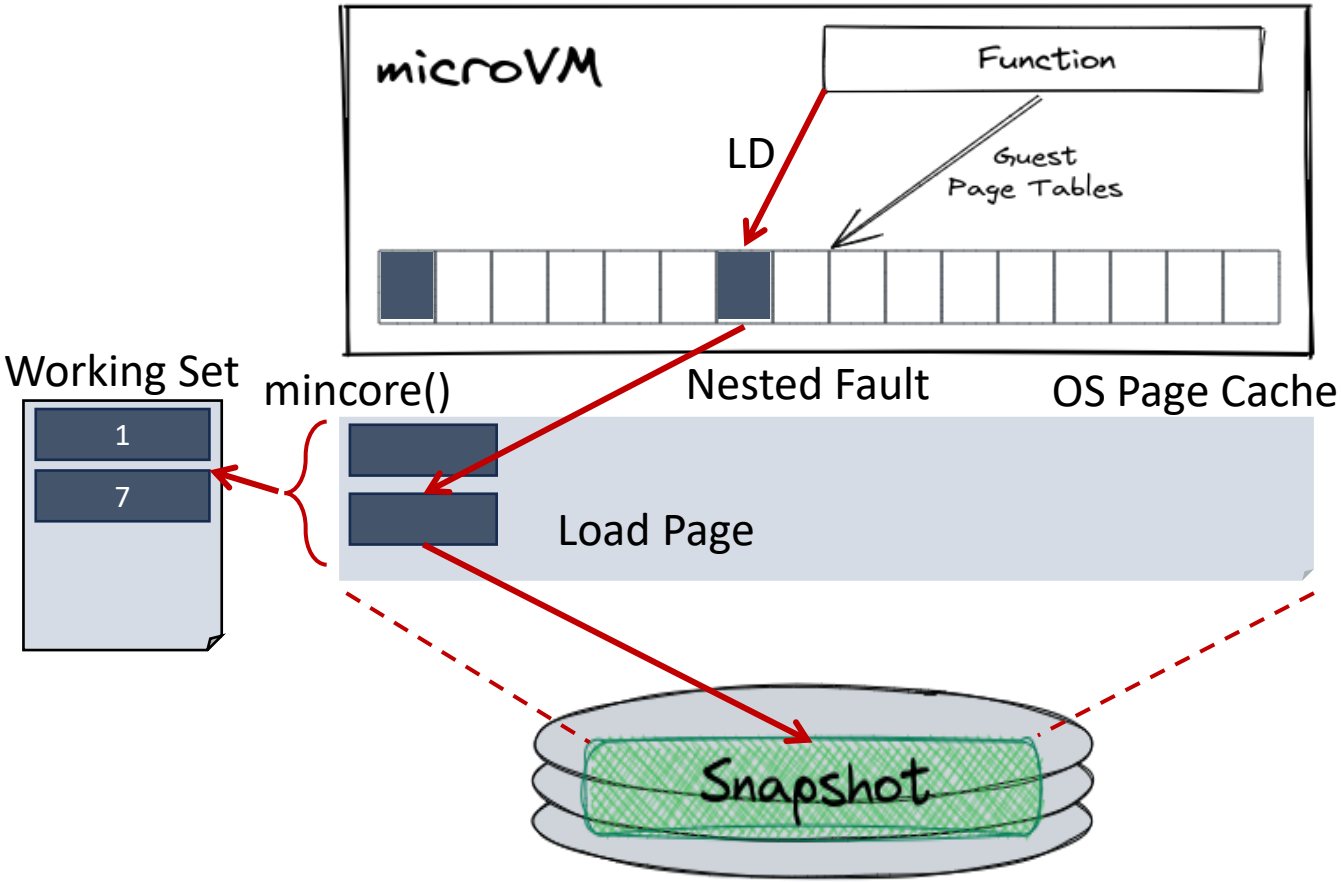
Working Set Prefetching: FaaSnap



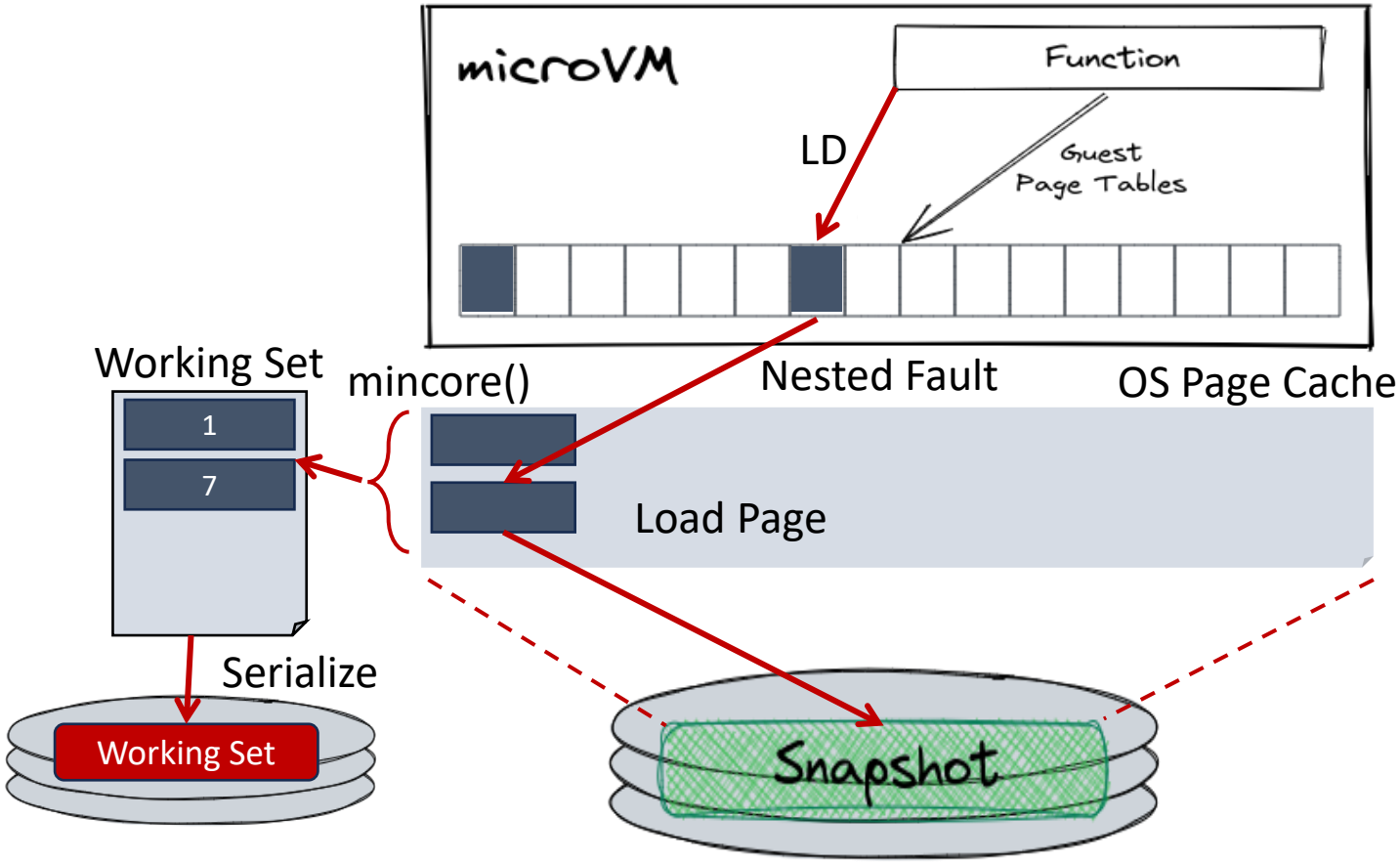
Working Set Prefetching: FaaSnap



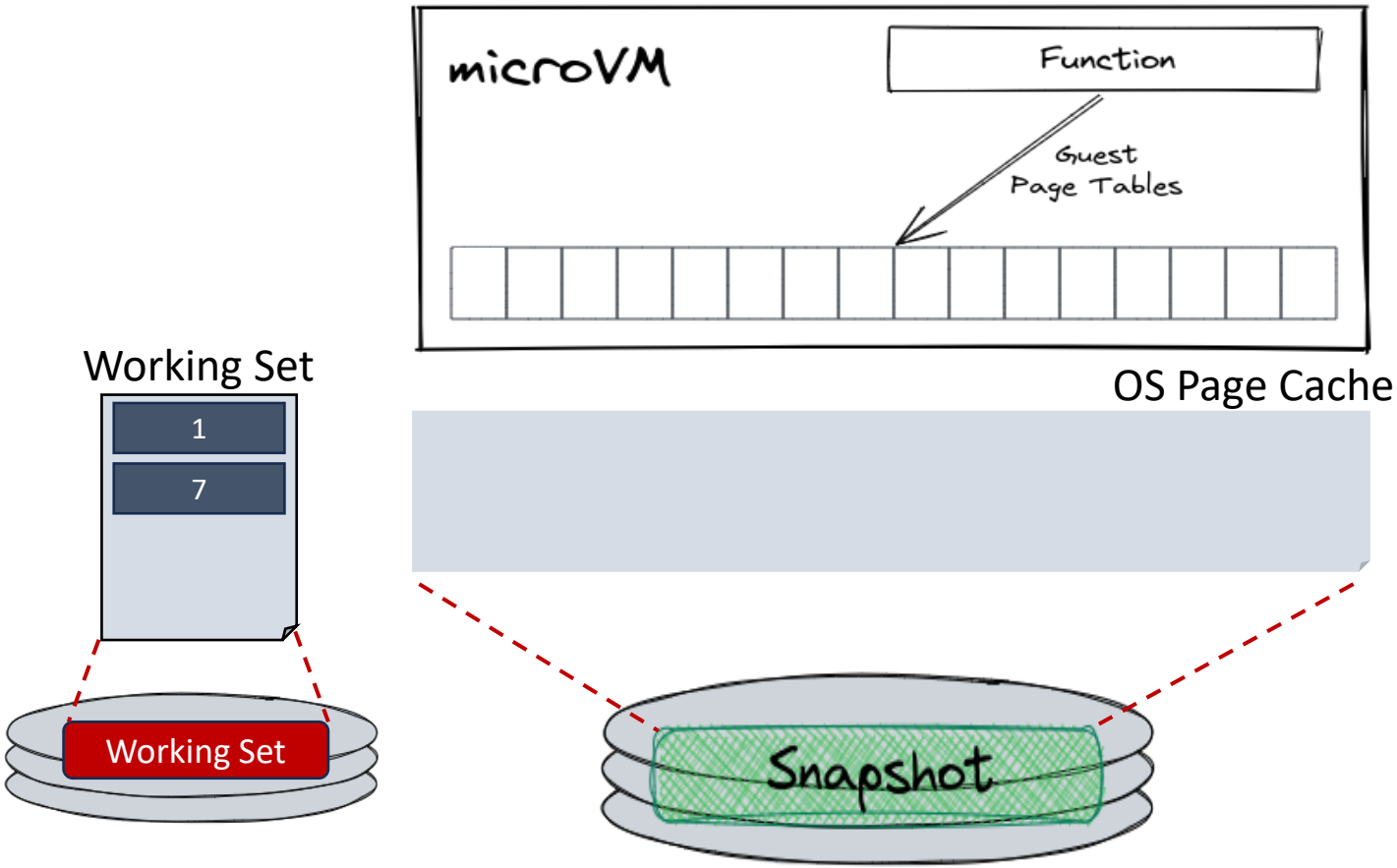
Working Set Prefetching: FaaSnap



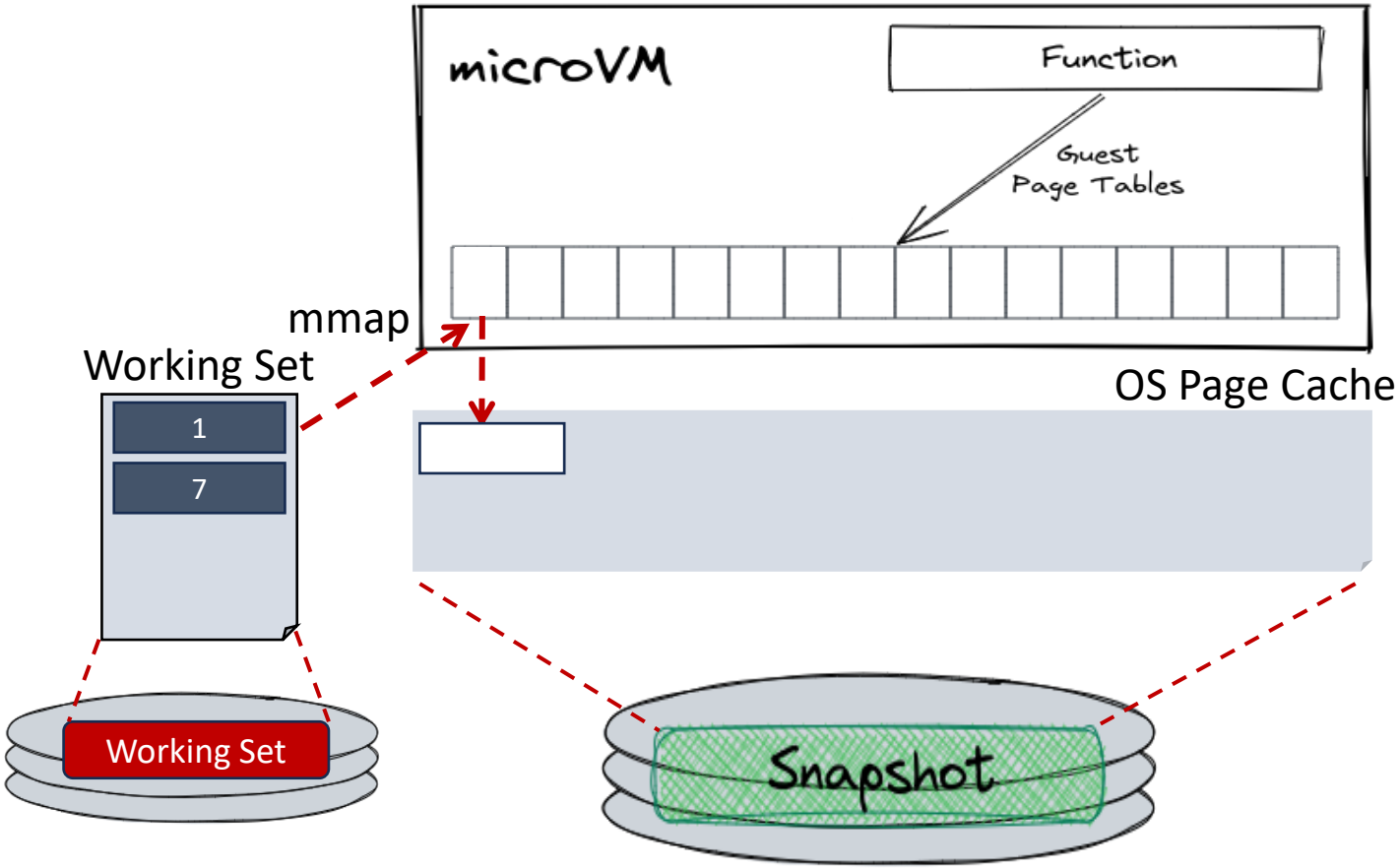
Working Set Prefetching: FaaSnap



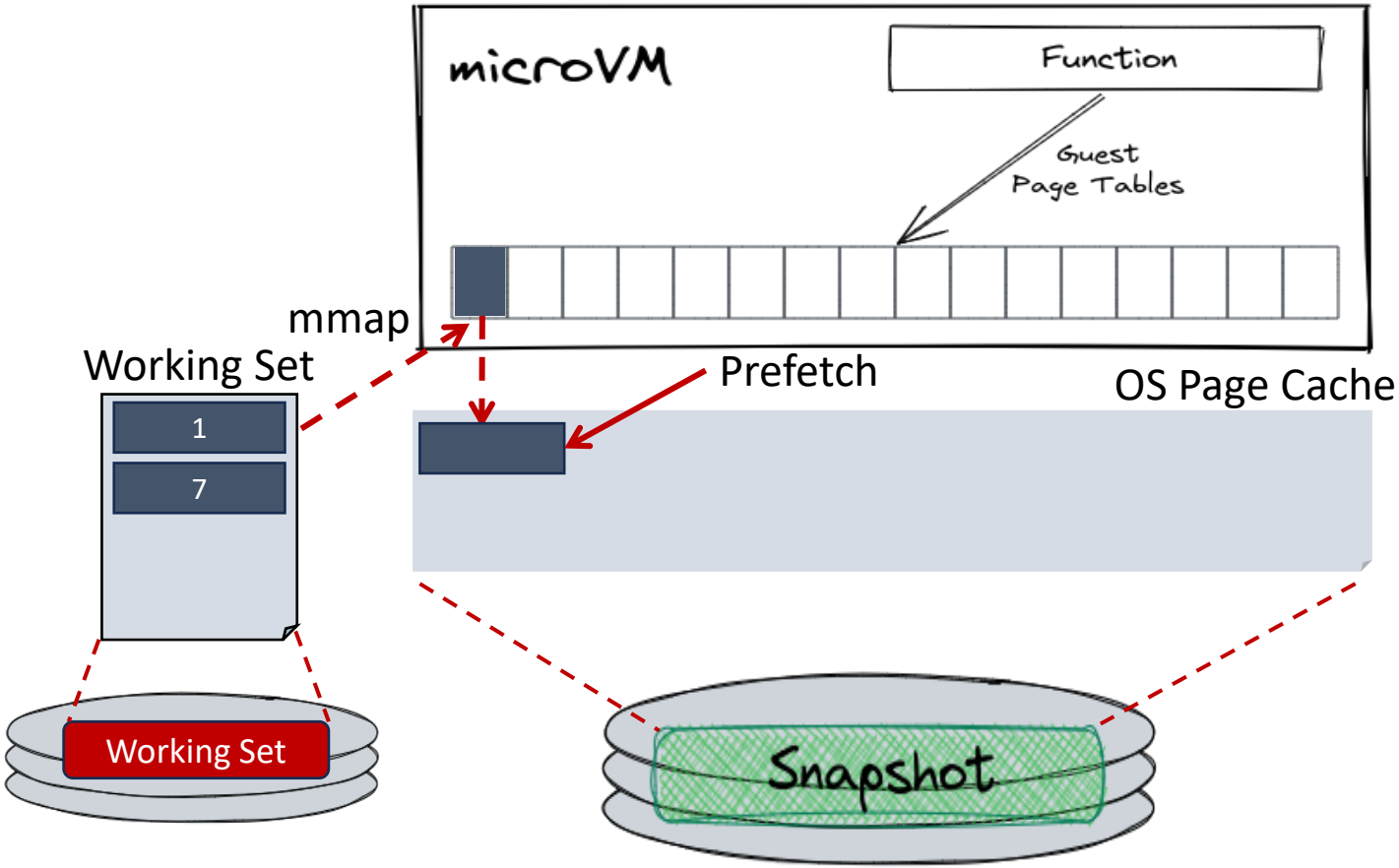
Working Set Prefetching: FaaSnap



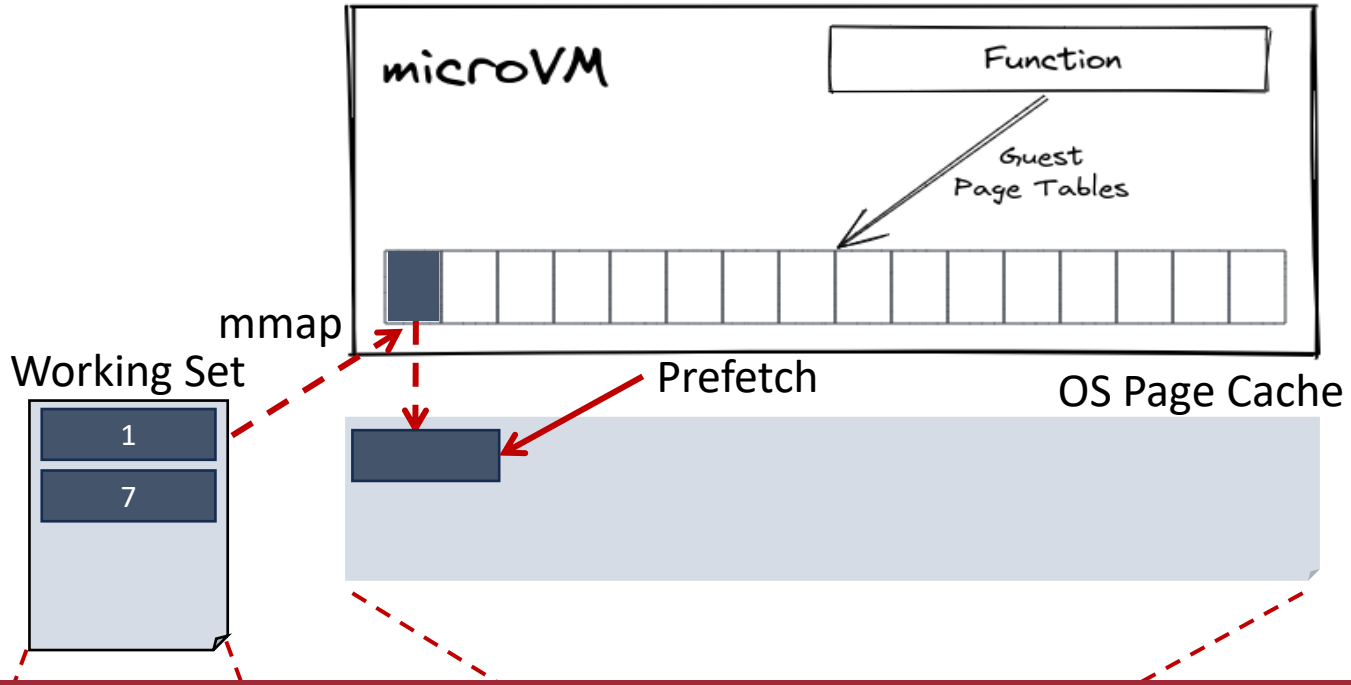
Working Set Prefetching: FaaSnap



Working Set Prefetching: FaaSnap

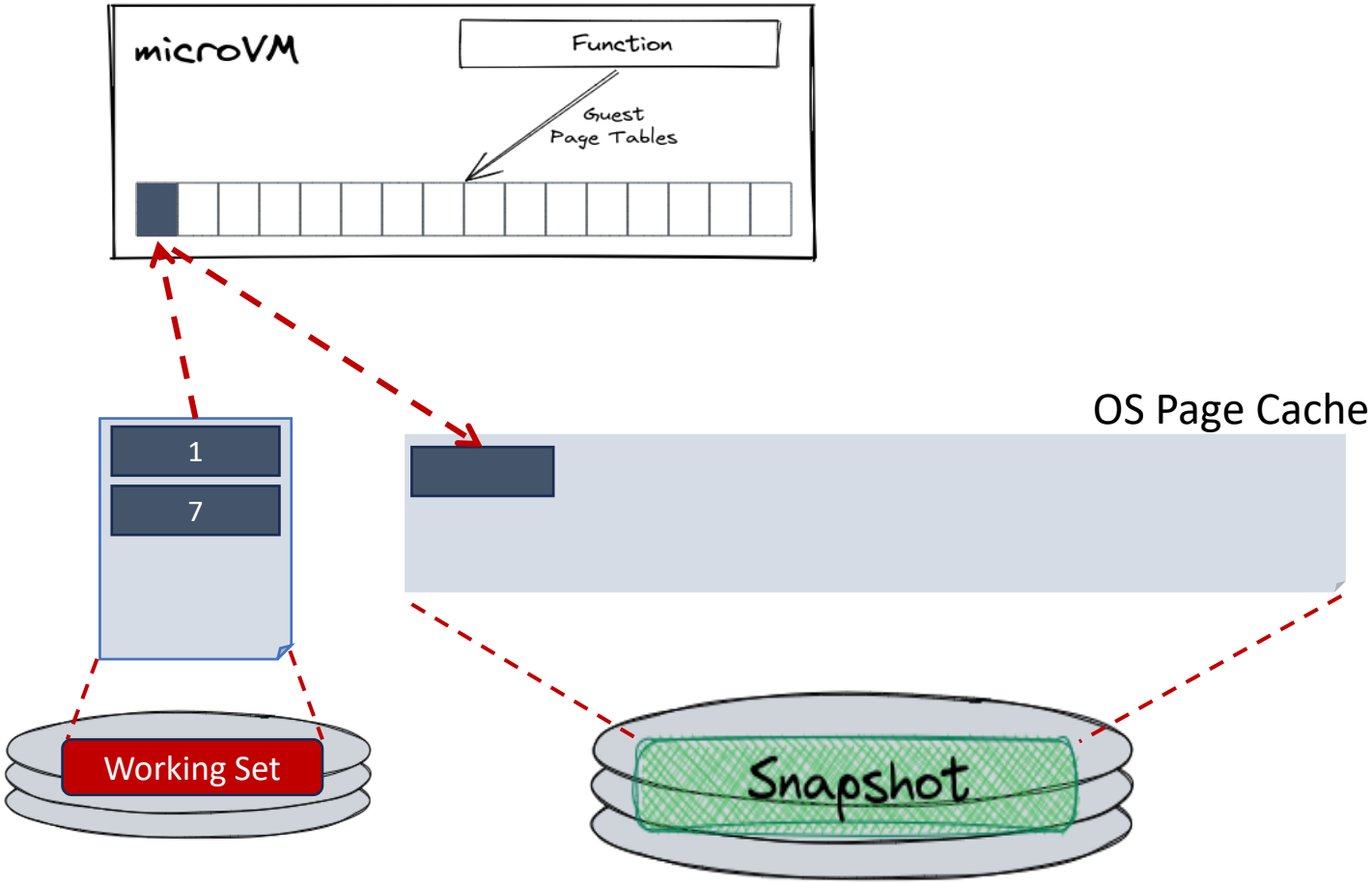


Working Set Prefetching: FaaSnap

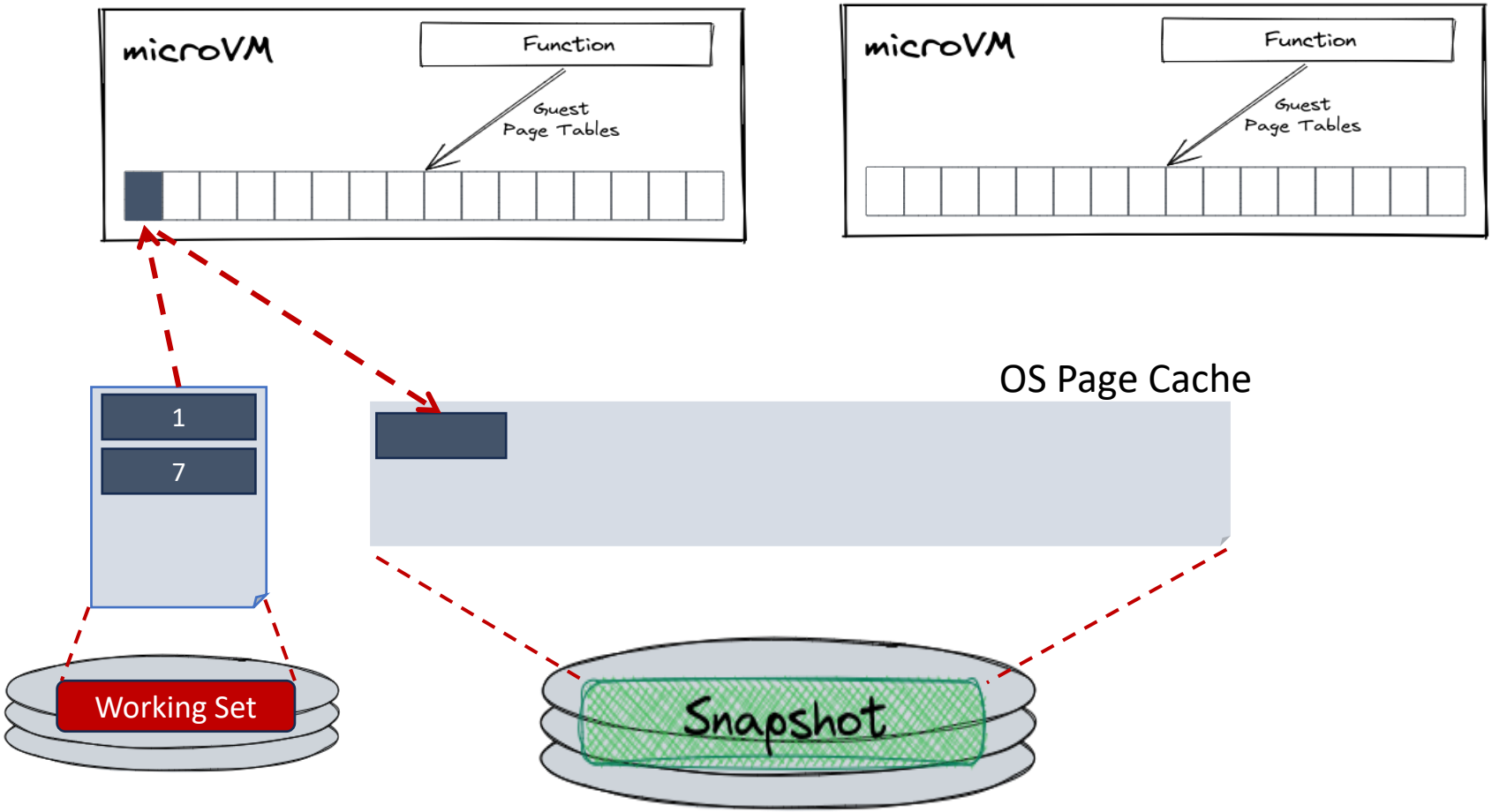


Fragmented working sets incur mmap() syscall overhead

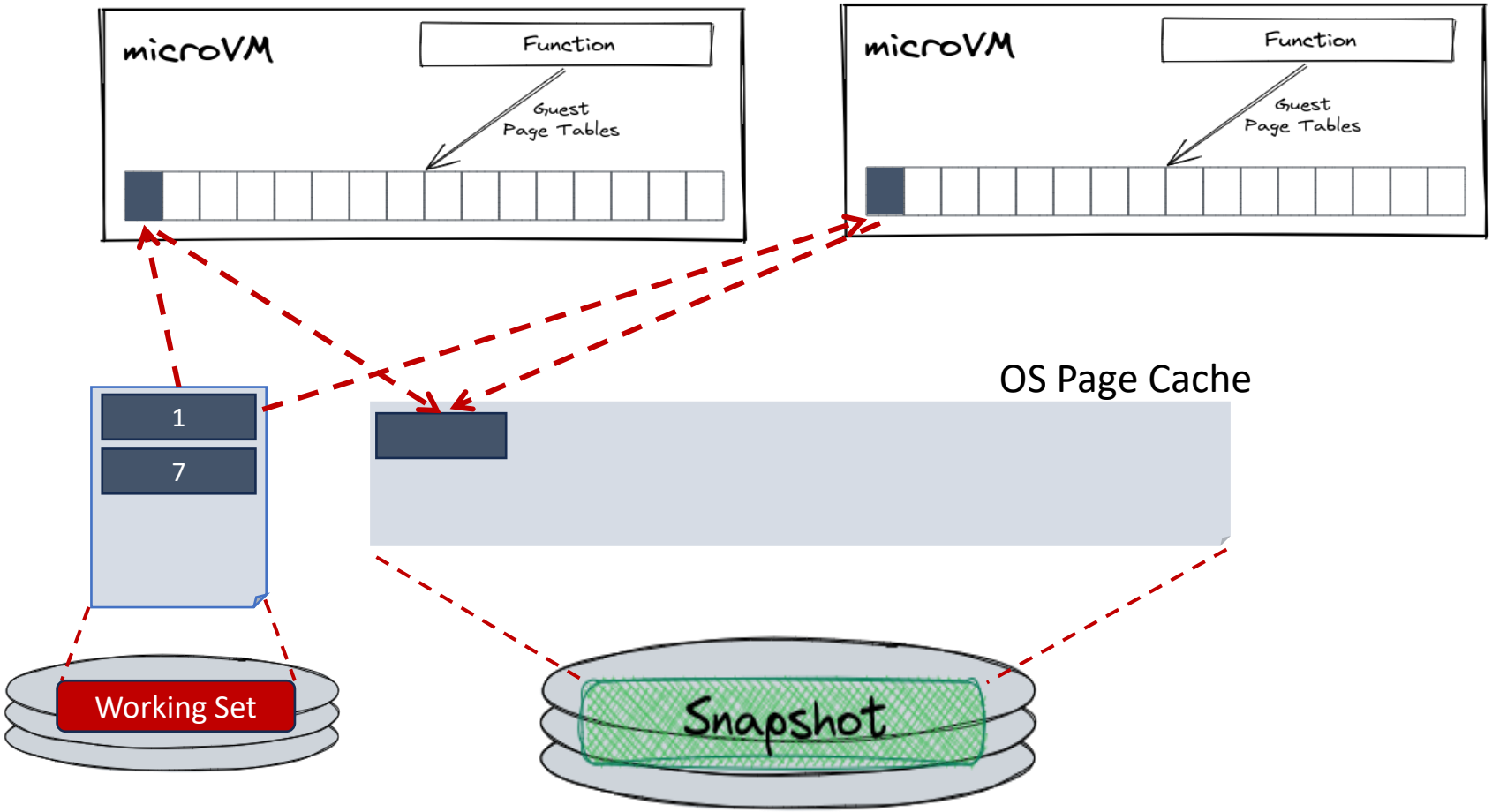
Working Set Prefetching: FaaSnap



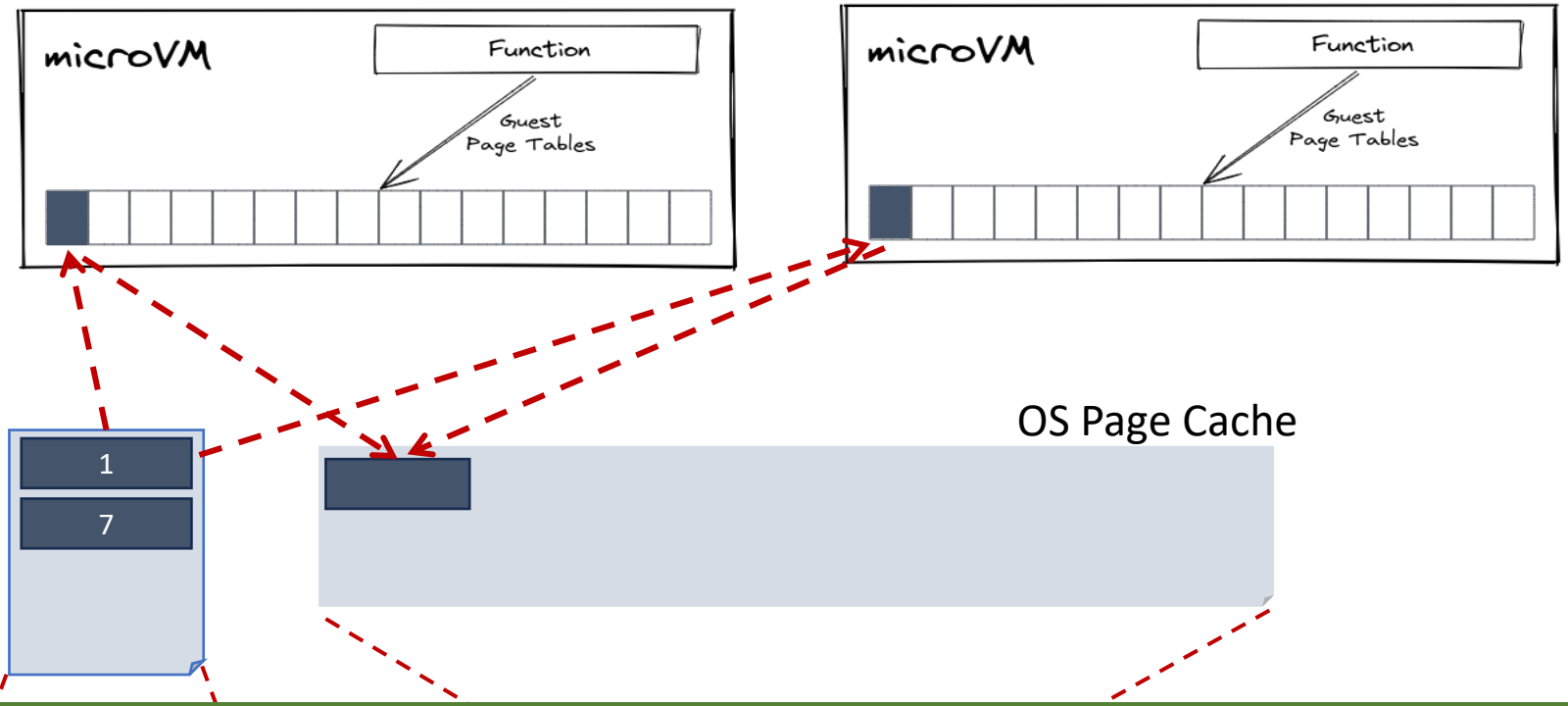
Working Set Prefetching: FaaSnap



Working Set Prefetching: FaaSnap

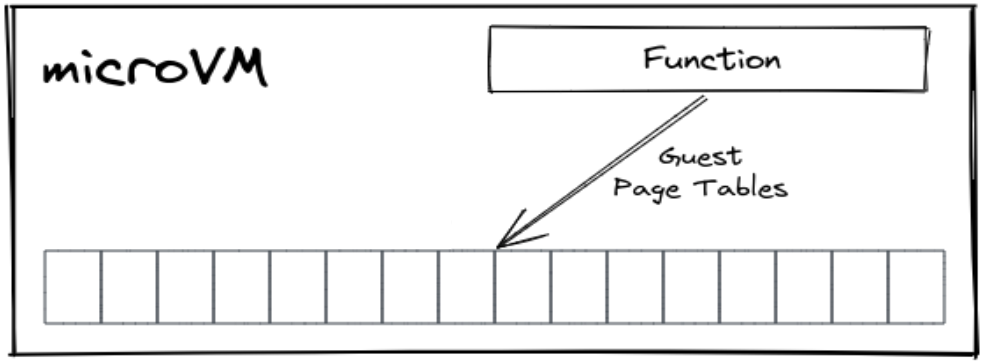


Working Set Prefetching: FaaSnap

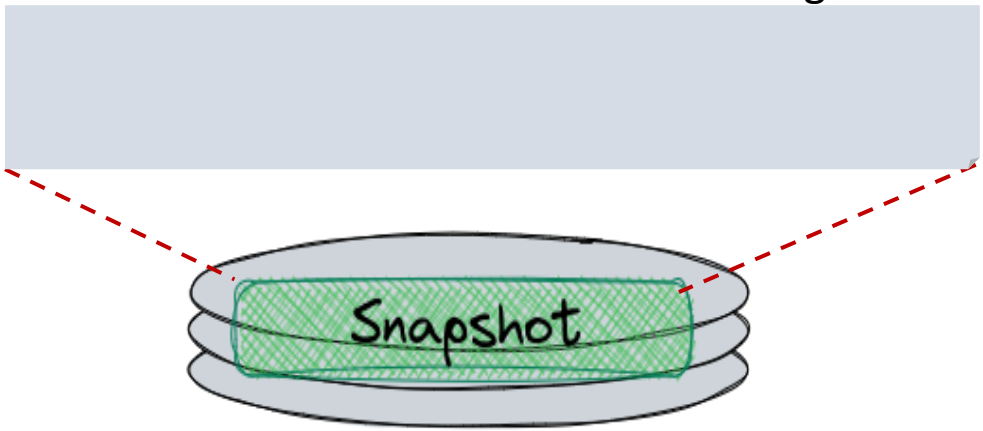


Deduplication via the page cache

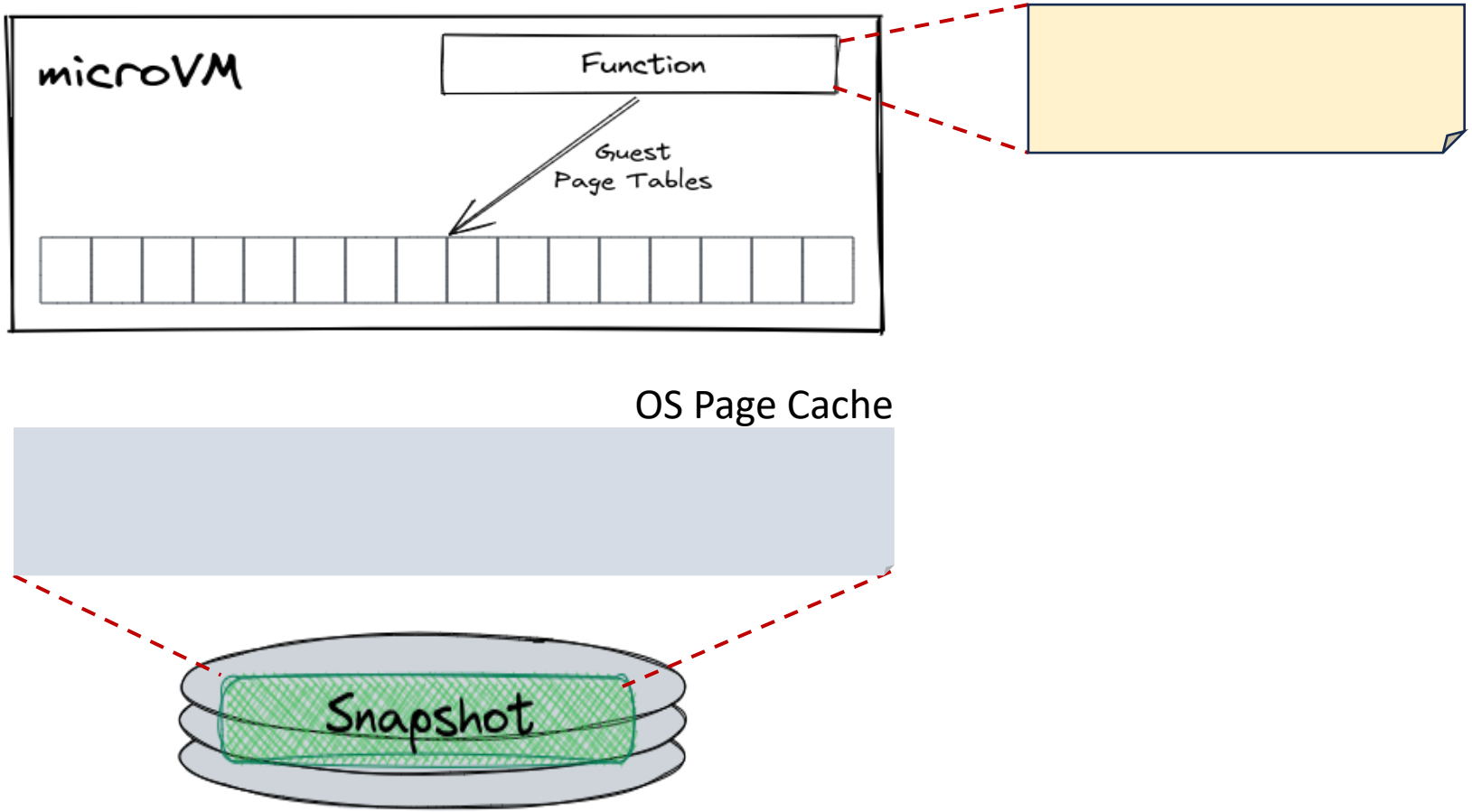
The overhead of free pages



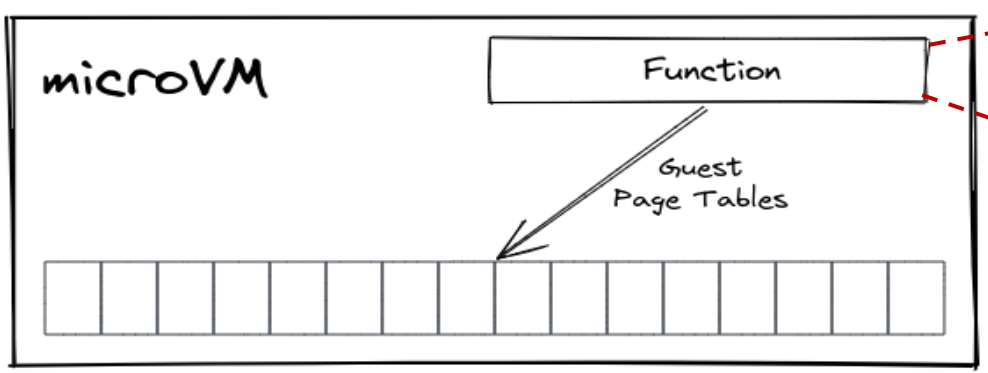
OS Page Cache



The overhead of free pages

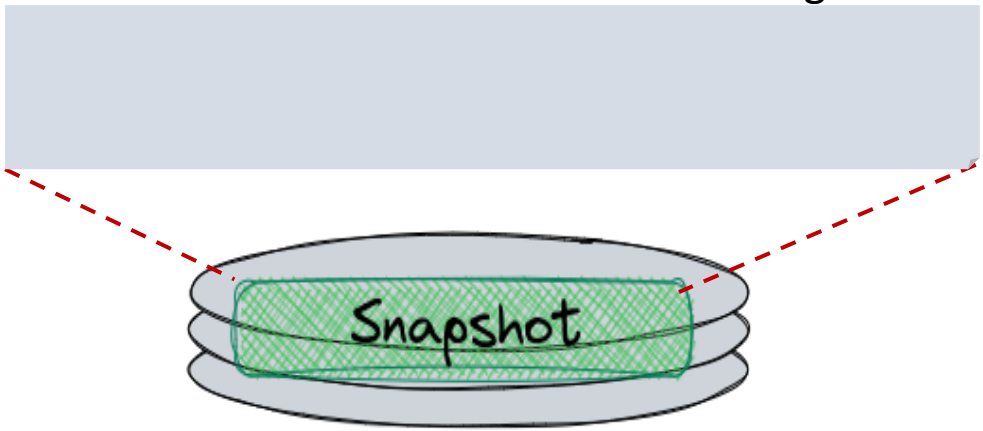


The overhead of free pages

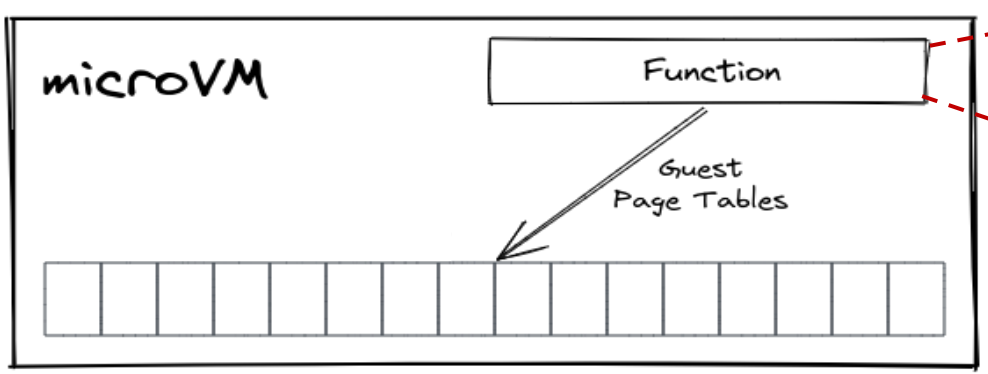



```
fn(input):
    output = malloc()
    return compute(input, &output)
```

OS Page Cache

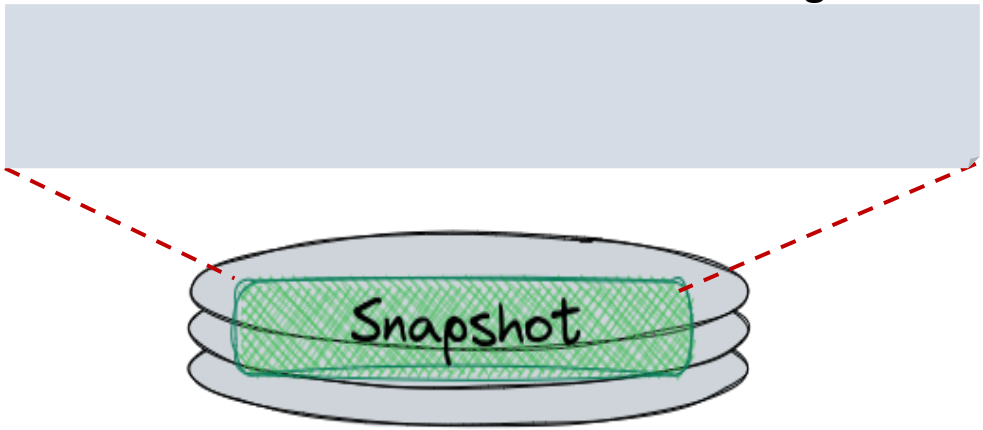


The overhead of free pages

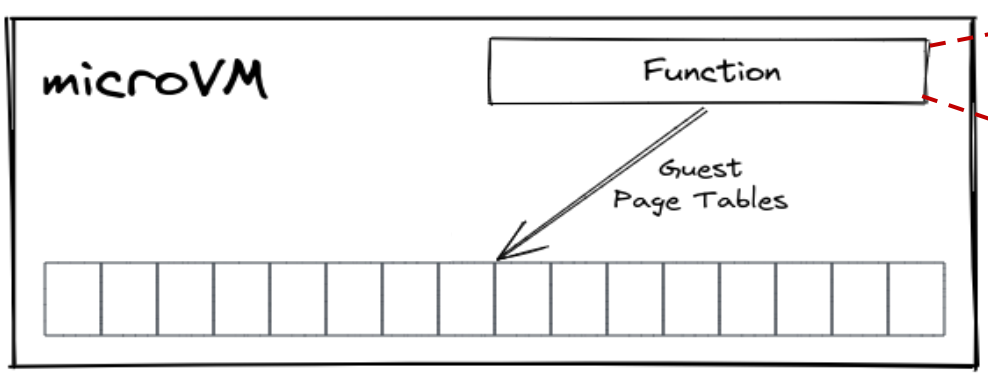



```
fn(input):
  output = malloc()
  return compute(input, &output)
```

OS Page Cache

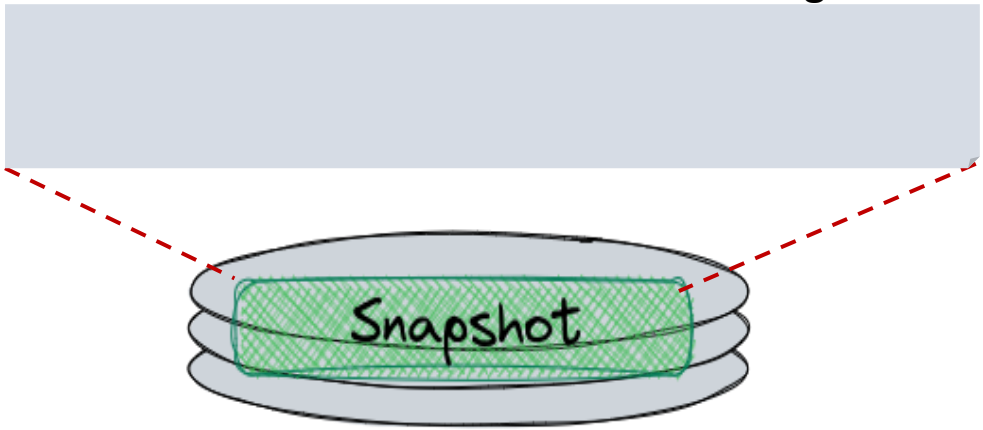


The overhead of free pages

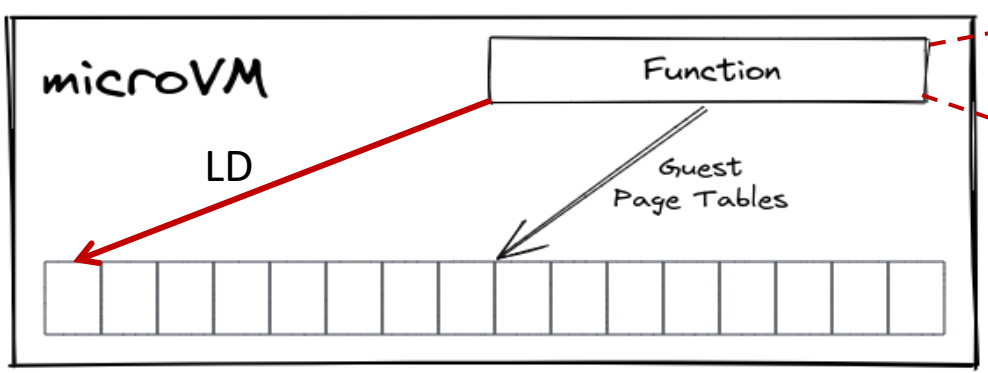



```
fn(input):
    output = malloc()
    return compute(input, &output)
```

OS Page Cache

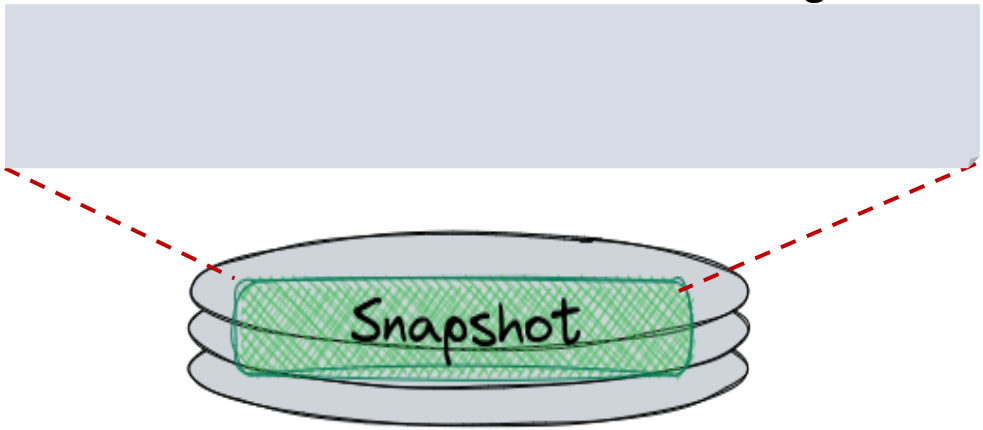


The overhead of free pages

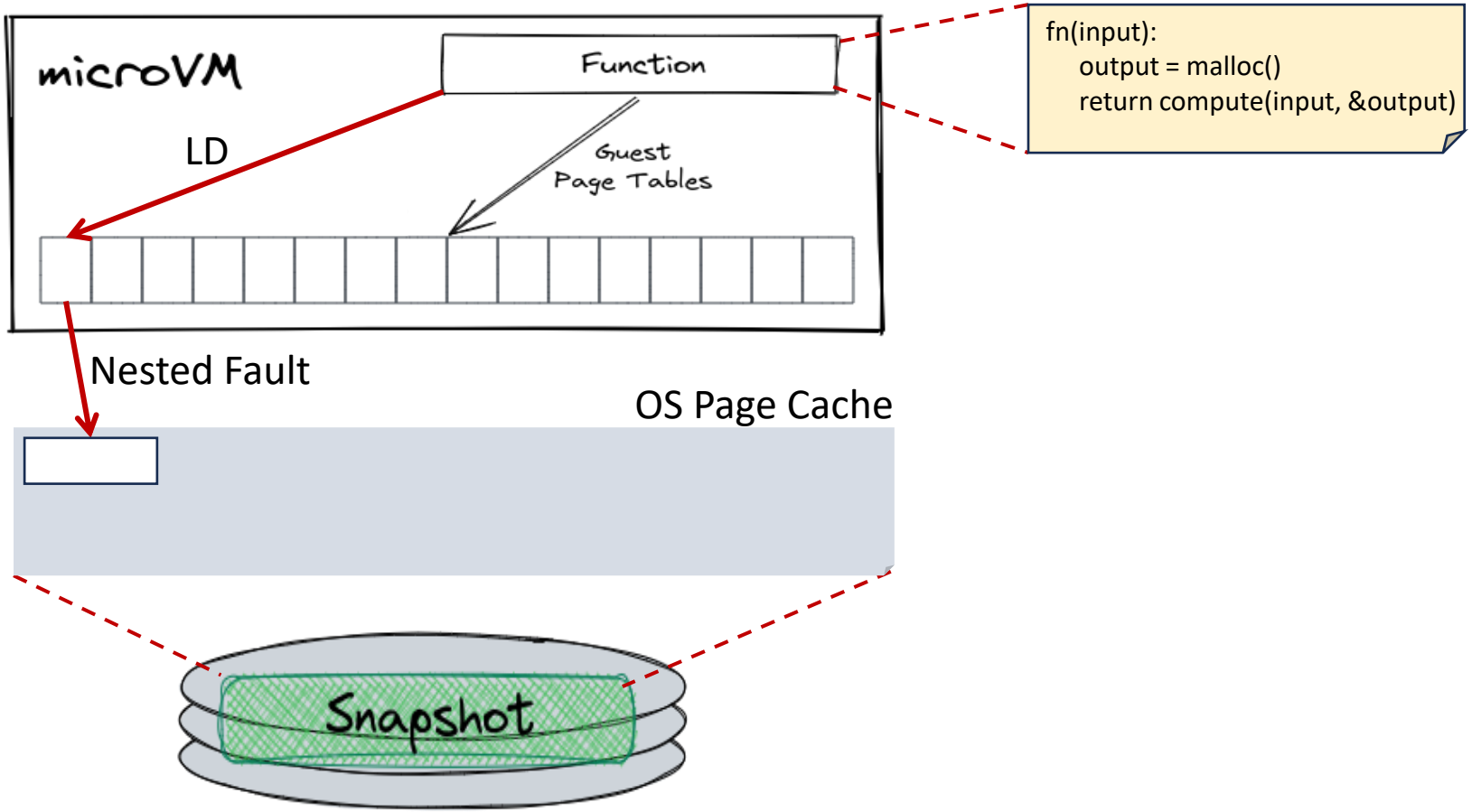



```
fn(input):
    output = malloc()
    return compute(input, &output)
```

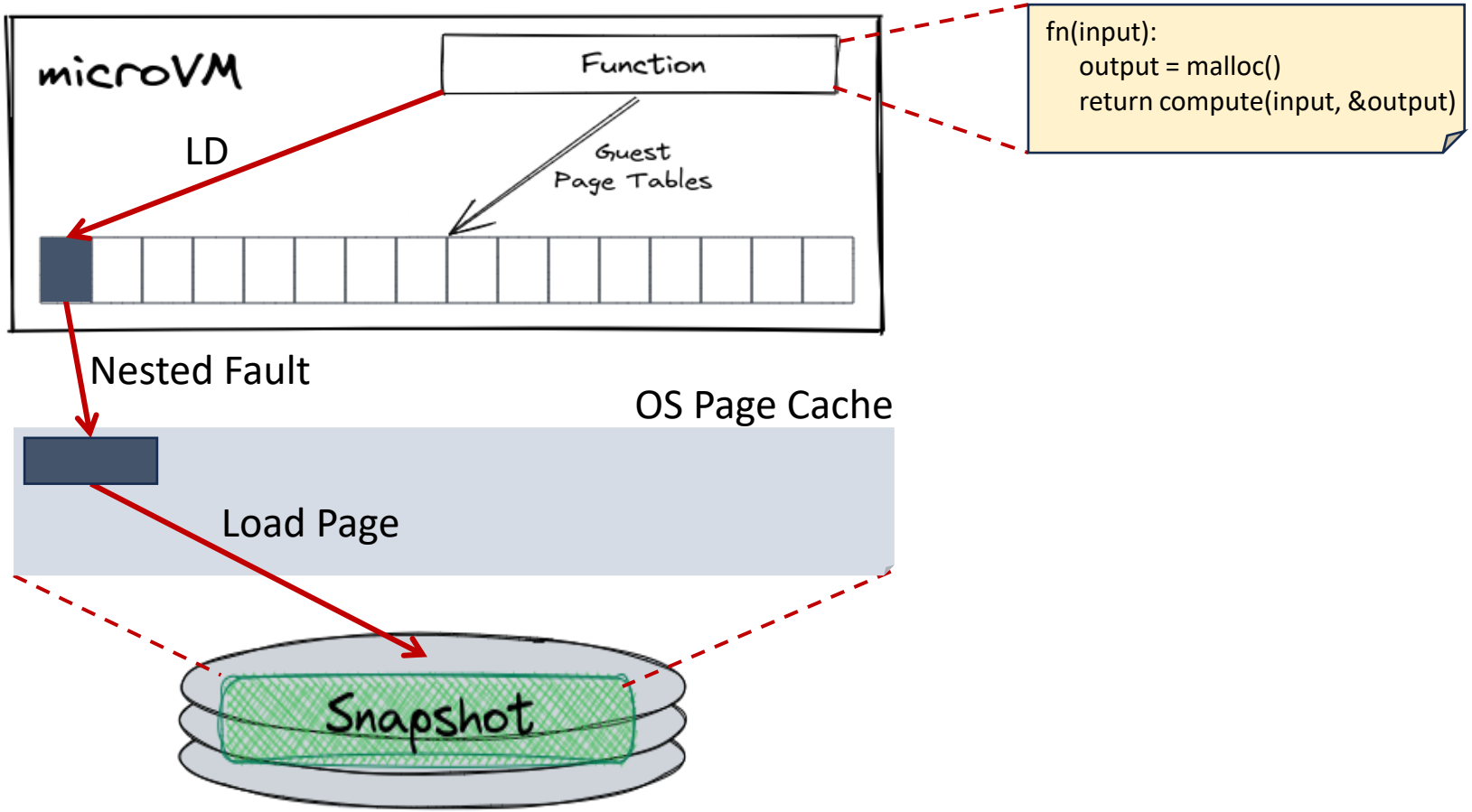
OS Page Cache



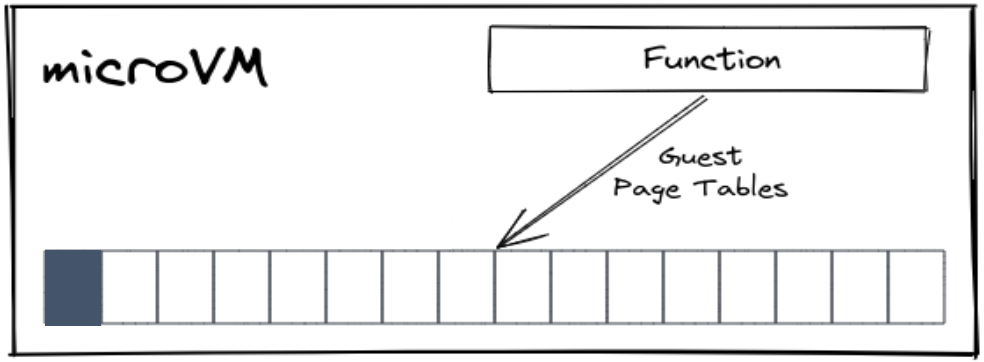
The overhead of free pages



The overhead of free pages

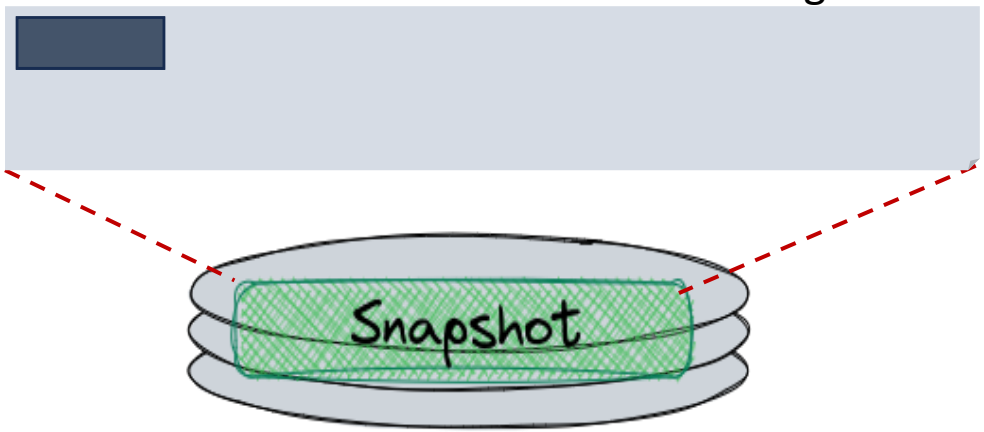


The overhead of free pages

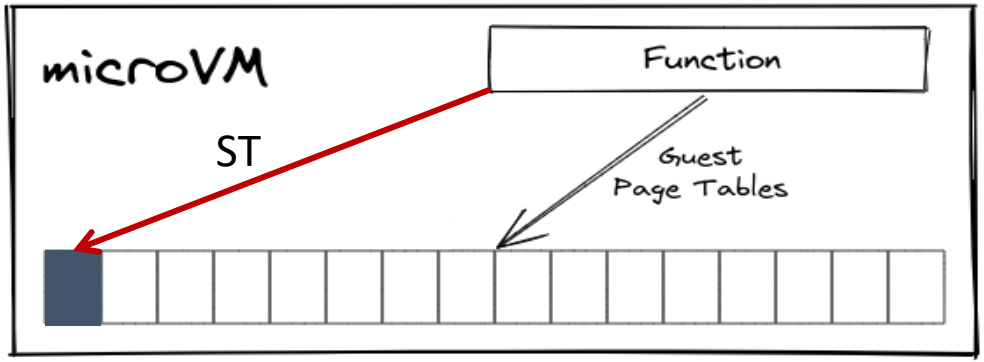


OS Page Cache

Anonymous Memory

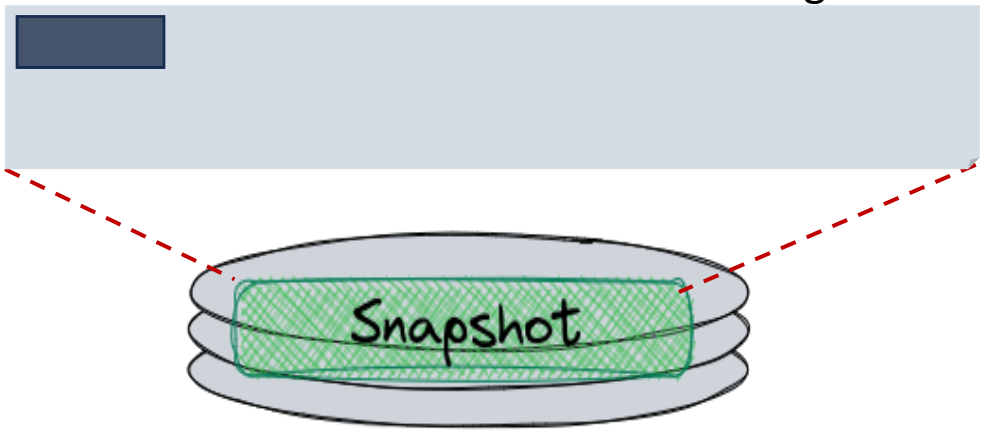


The overhead of free pages

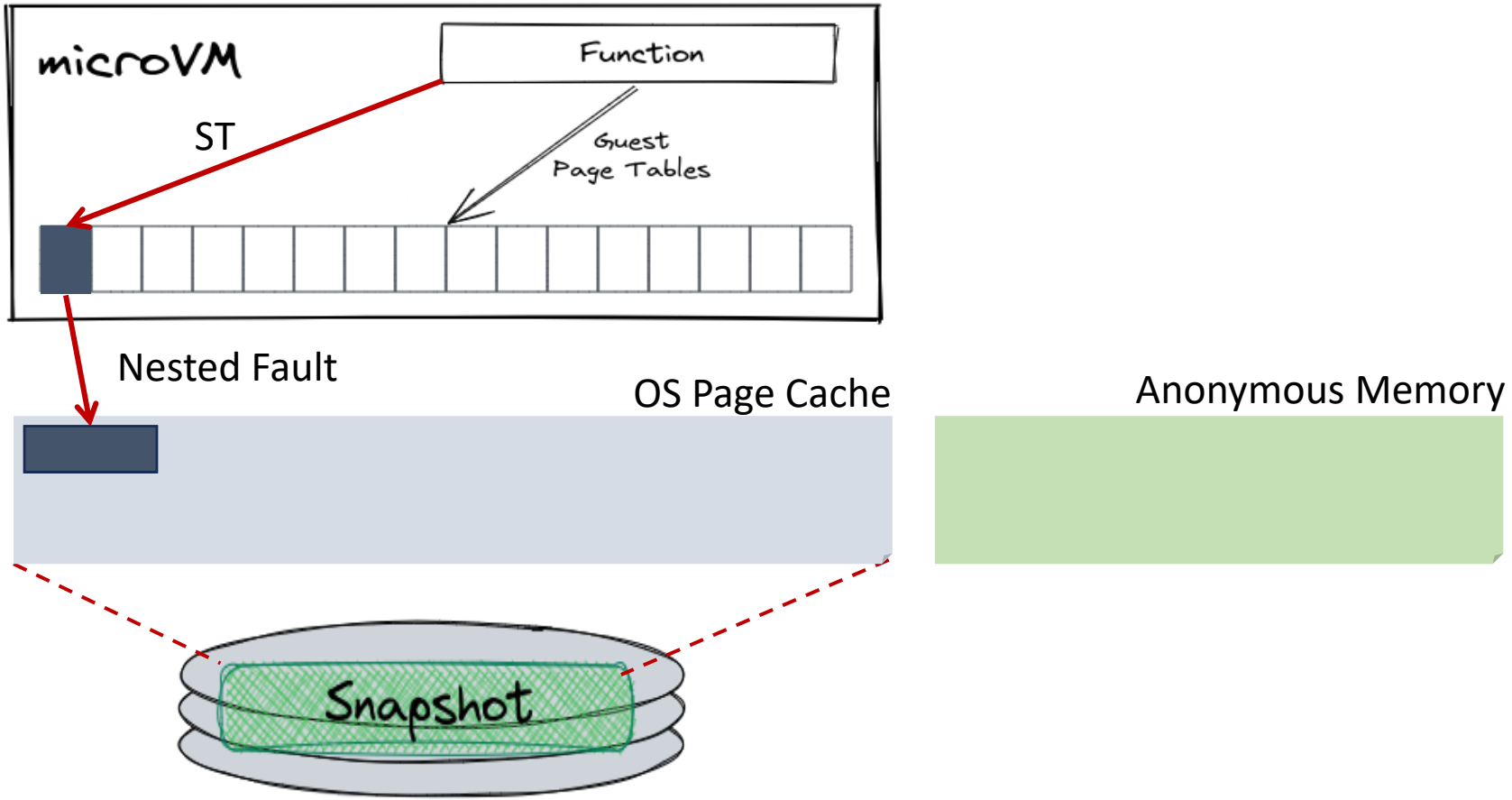


OS Page Cache

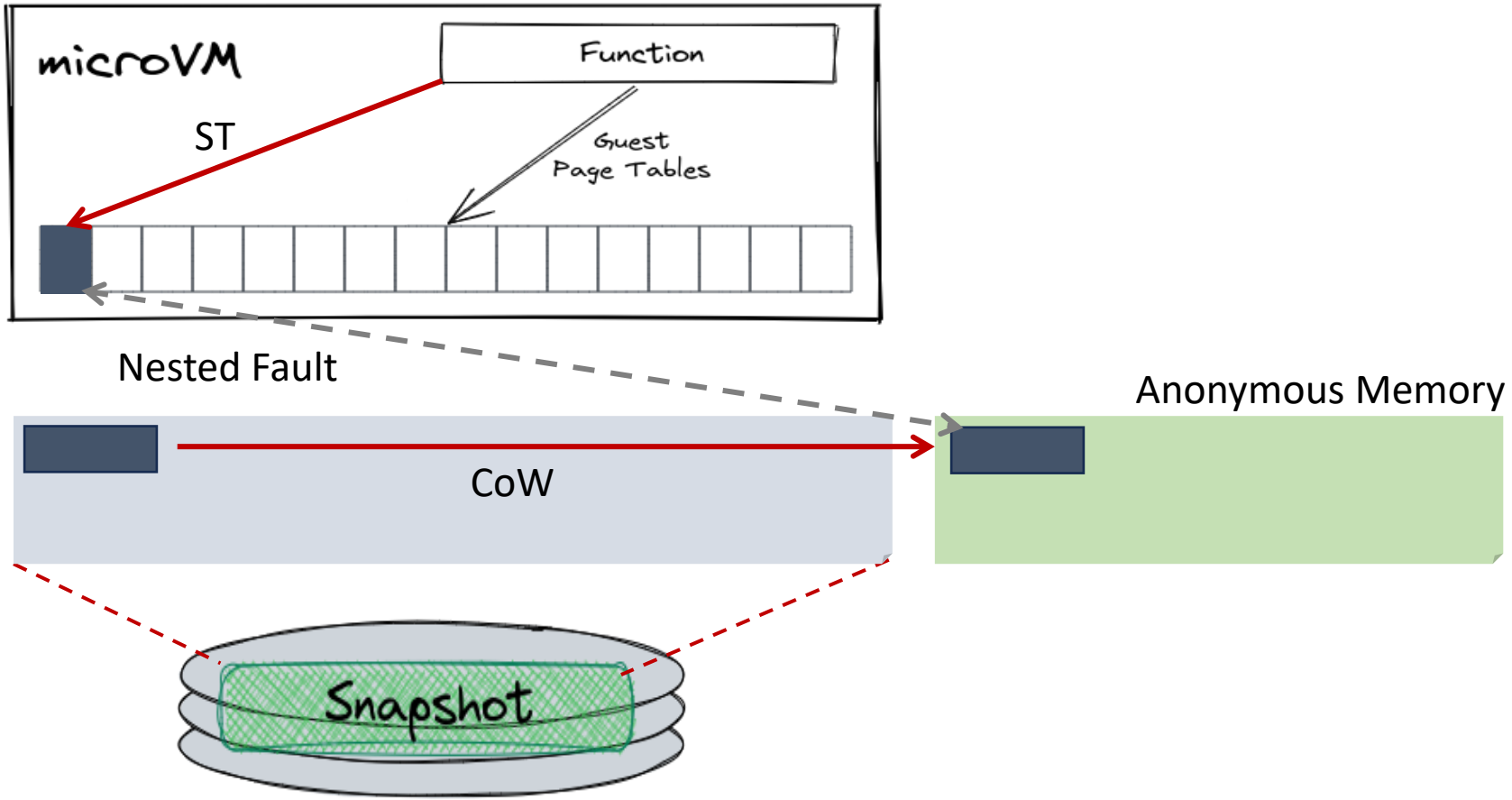
Anonymous Memory



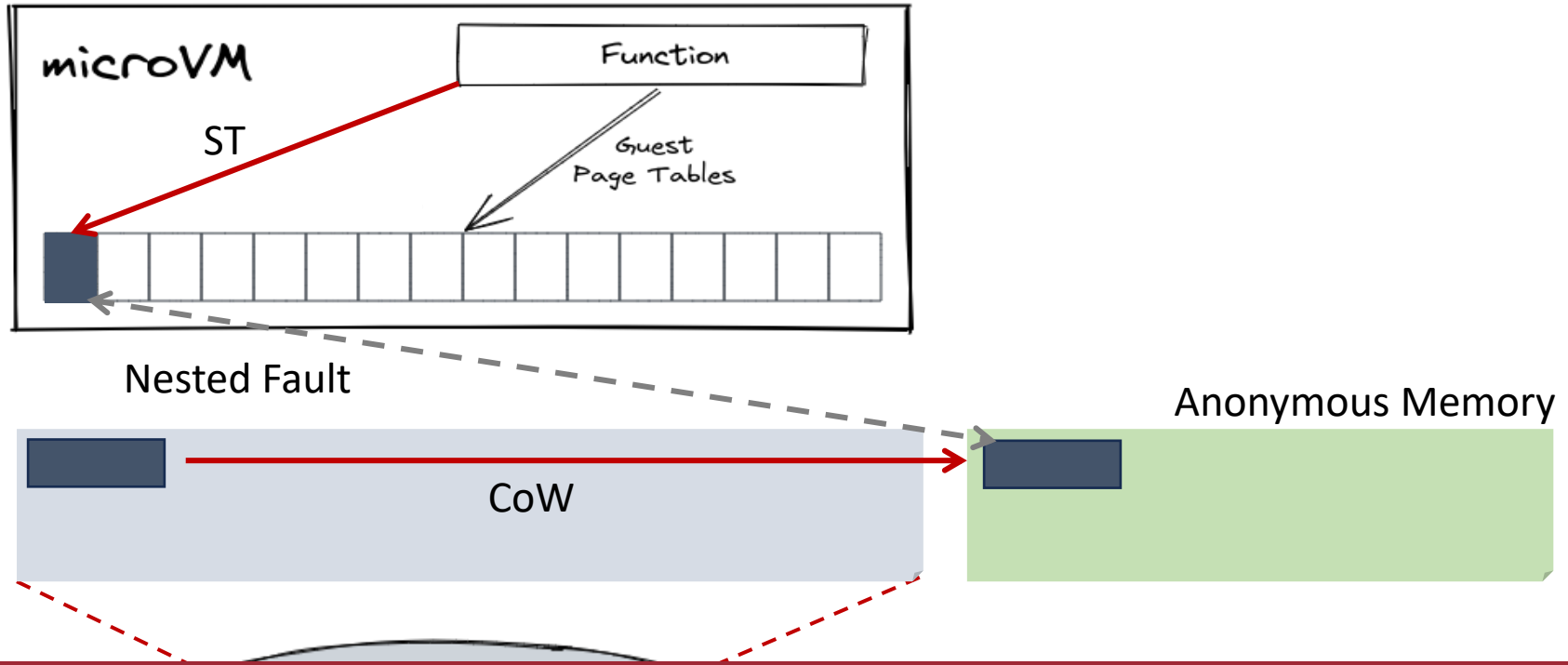
The overhead of free pages



The overhead of free pages

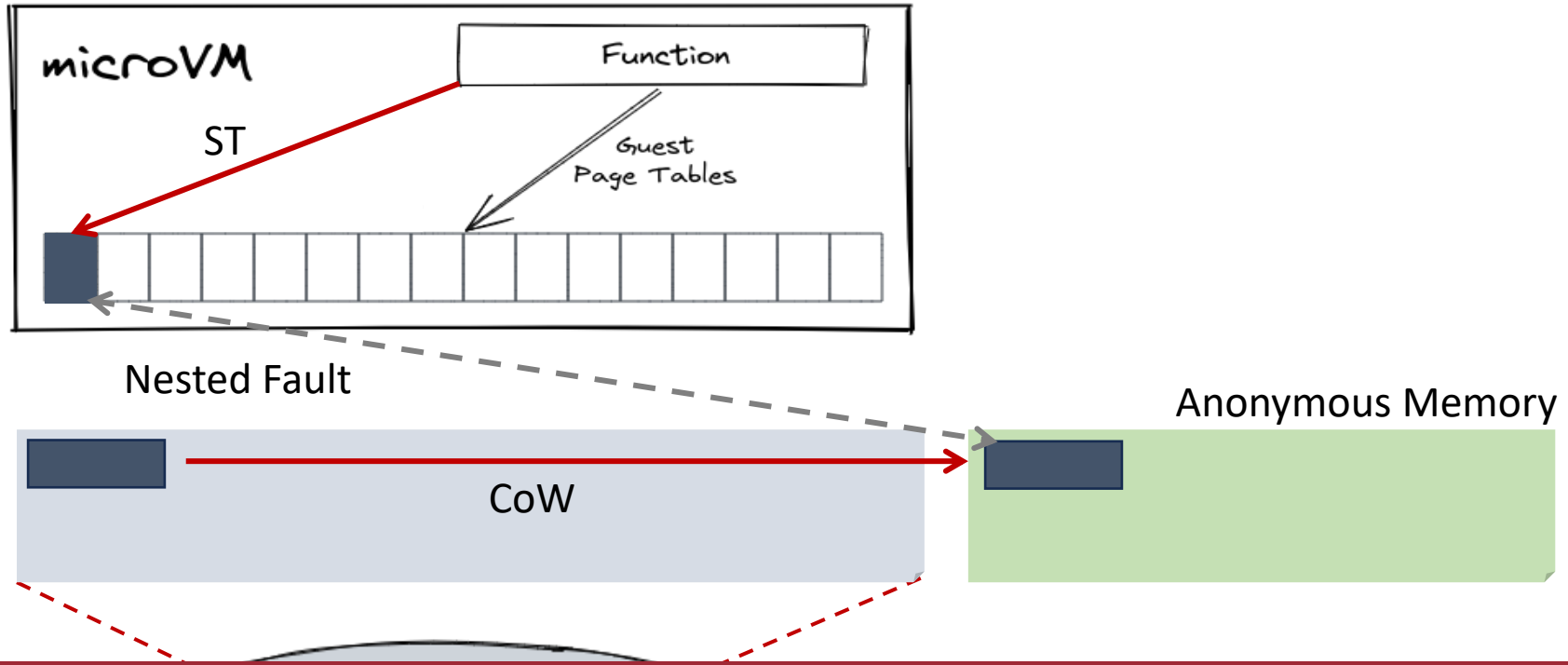


The overhead of free pages



Free pages contain no valid contents == Wasted I/O

The overhead of free pages



Prior art filters free pages from loading via snapshot scanning

Outline

→ FaaS Snapshotting

→ Working Set Prefetching

→ SnapBPF

i. eBPF capture and prefetch

ii. PV free page filtering

→ Evaluation

→ Conclusion

Outline

→ FaaS Snapshotting

→ Working Set Prefetching

→ SnapBPF

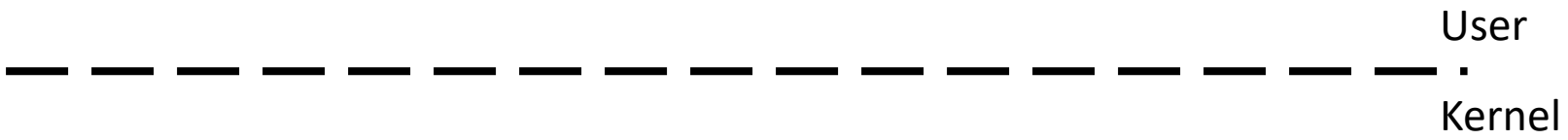
i. eBPF capture and prefetch

ii. PV free page filtering

→ Evaluation

→ Conclusion

eBPF primer



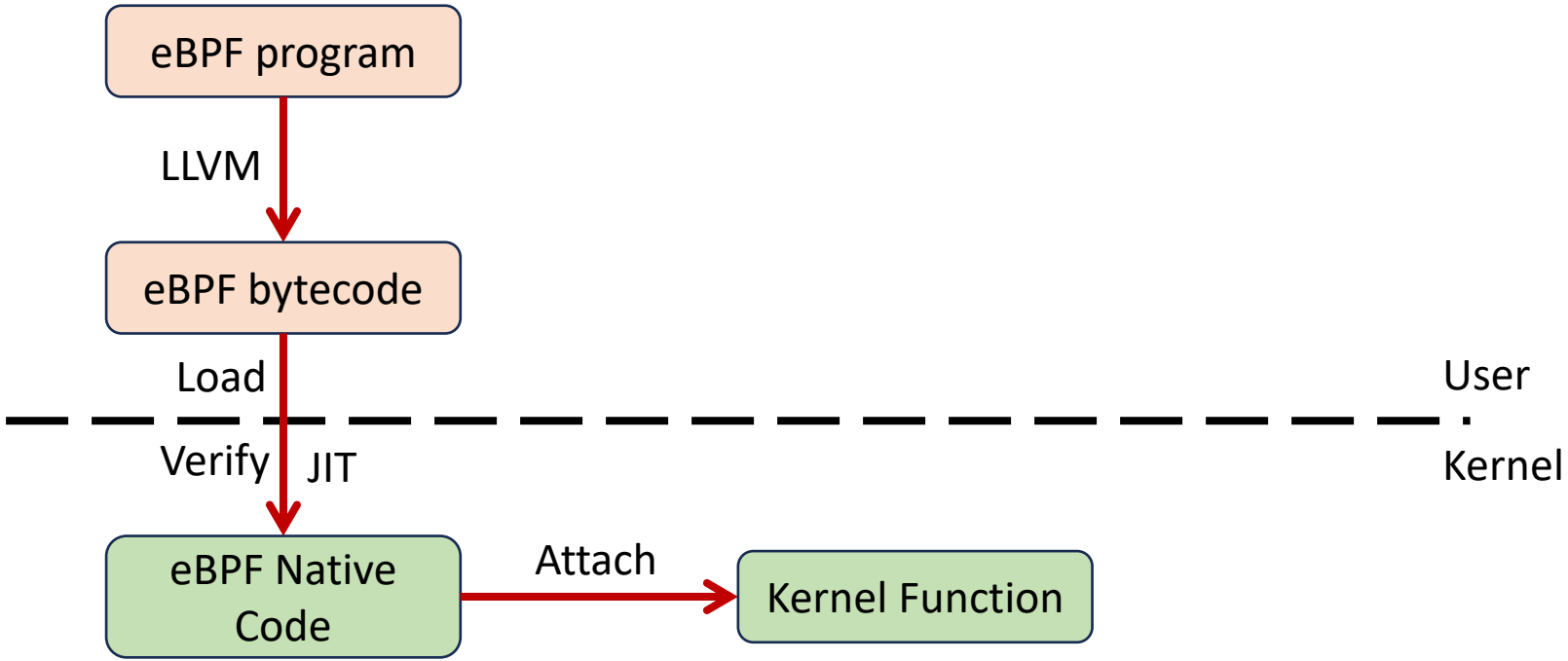
eBPF primer

eBPF primer

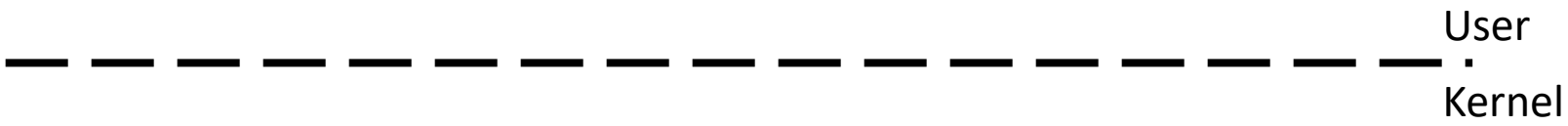
eBPF primer


eBPF program
LLVM
eBPF bytecode
Load
User
Verify
JIT
Kernel
eBPF Native Code

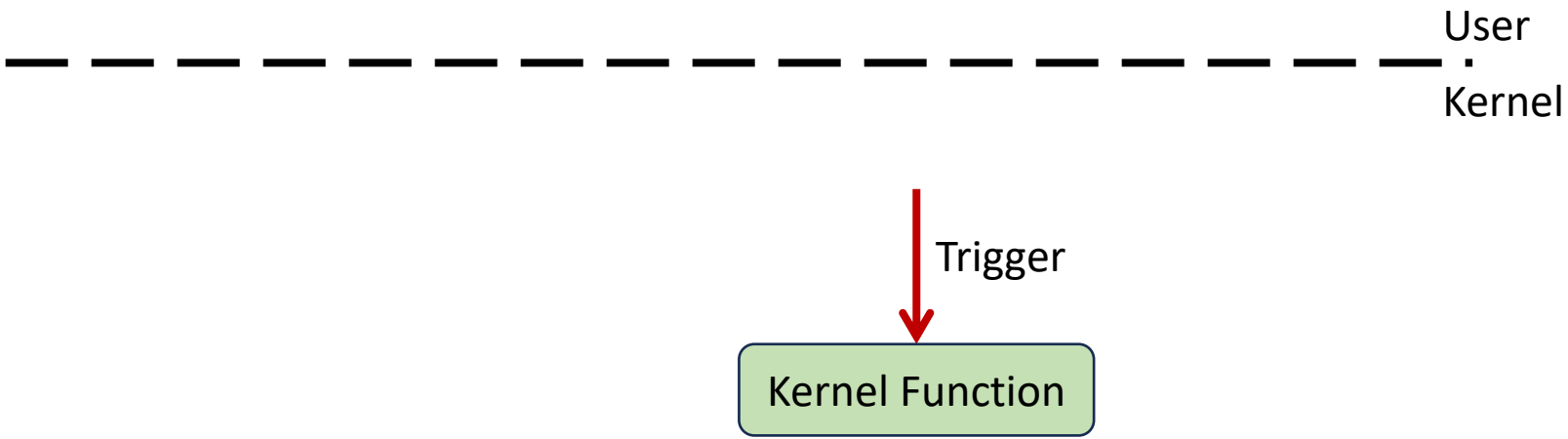
eBPF primer


eBPF program
LLVM
eBPF bytecode
Load
User
Verify
JIT
Kernel
eBPF Native Code
Attach
Kernel Function

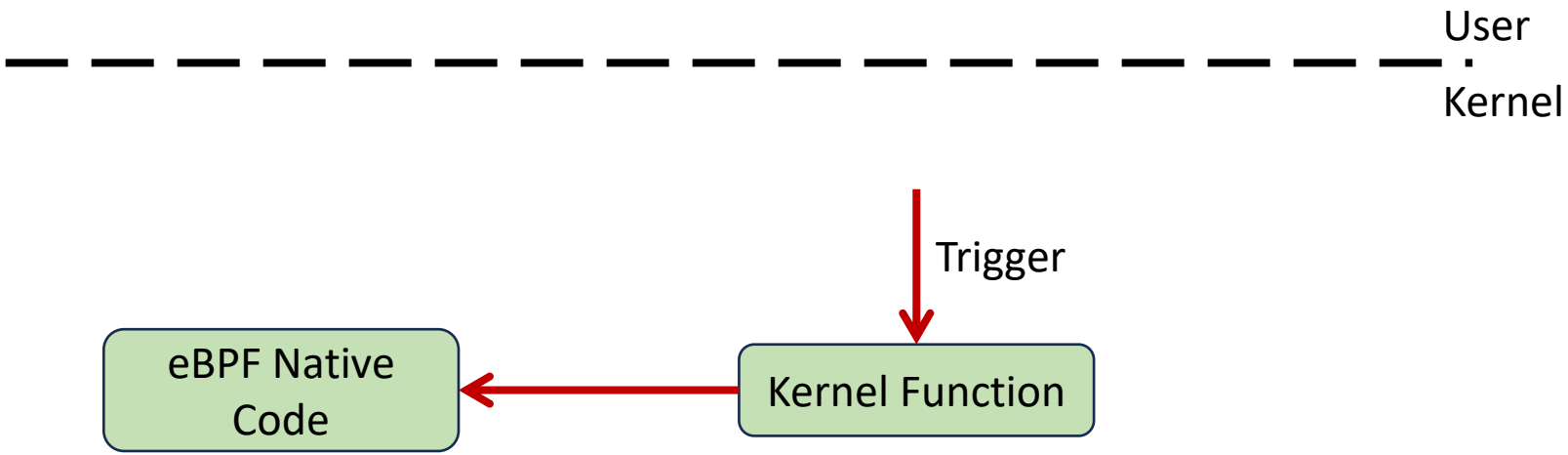
eBPF primer



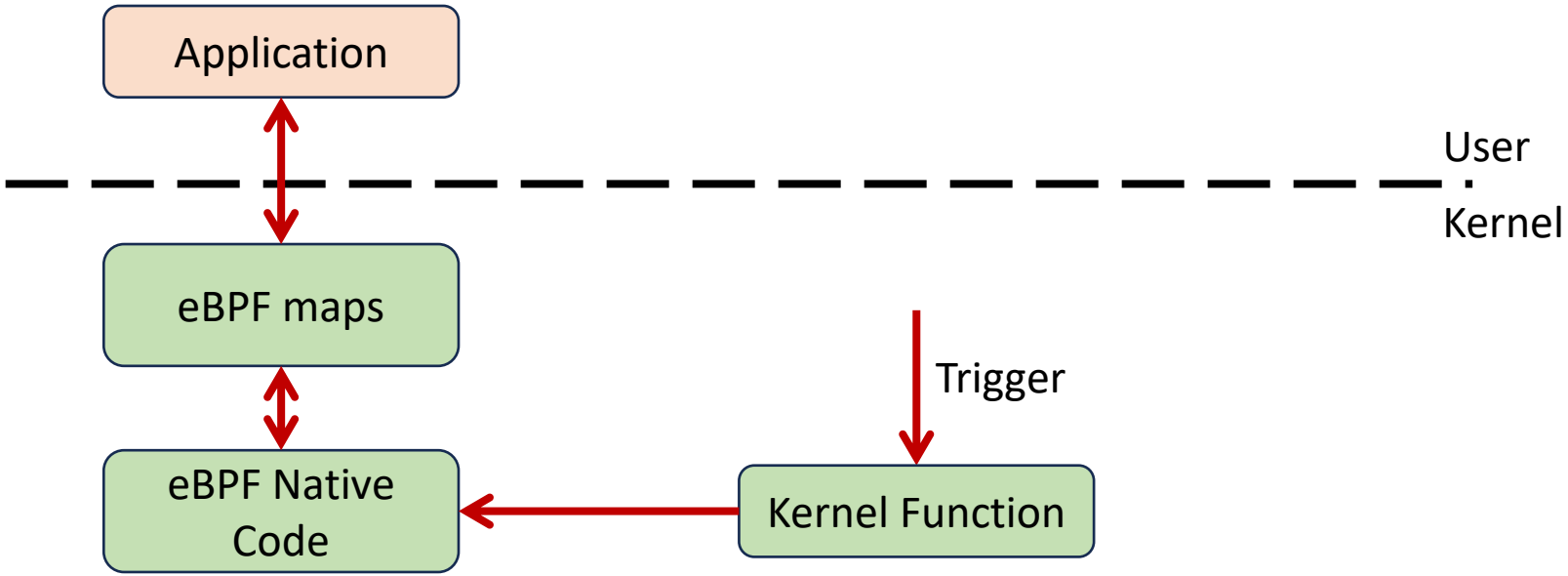
eBPF primer



eBPF primer



eBPF primer

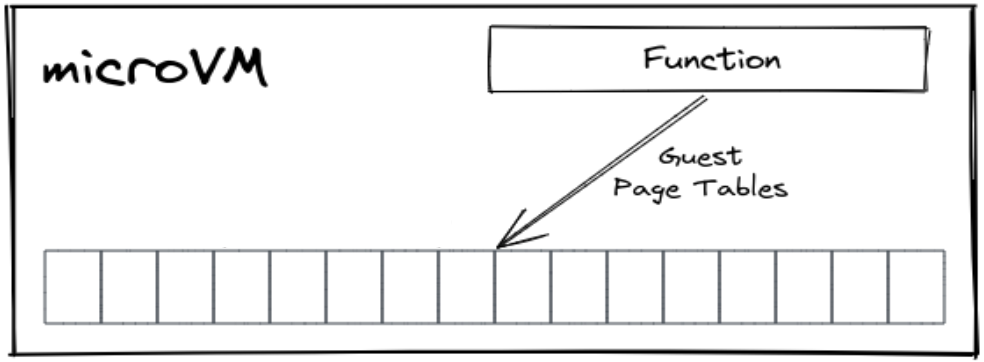

Application
User
Kernel
eBPF maps
Trigger
eBPF Native Code
Kernel Function

Insight:

Insight:

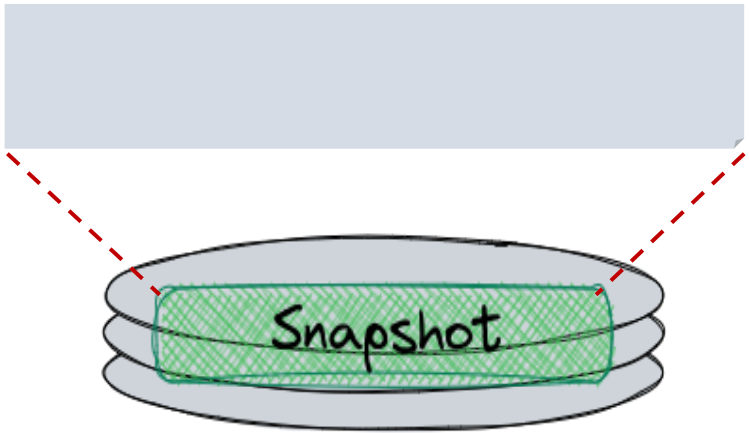
Use eBPF for kernelspace working set prefetching

eBPF Capture

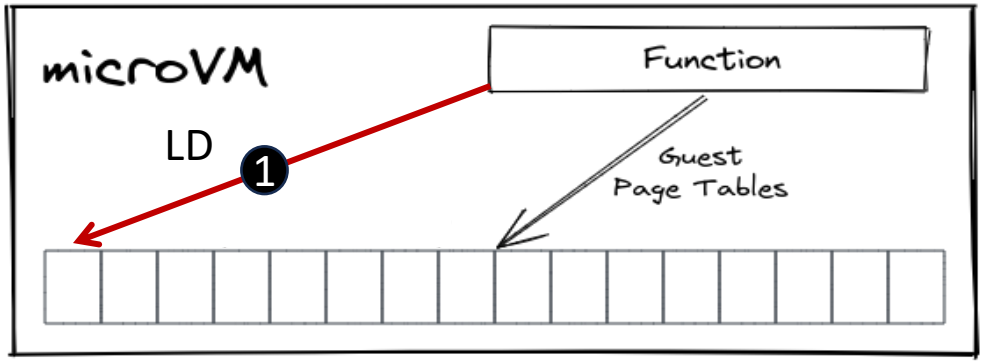


Host Kernel

OS Page Cache

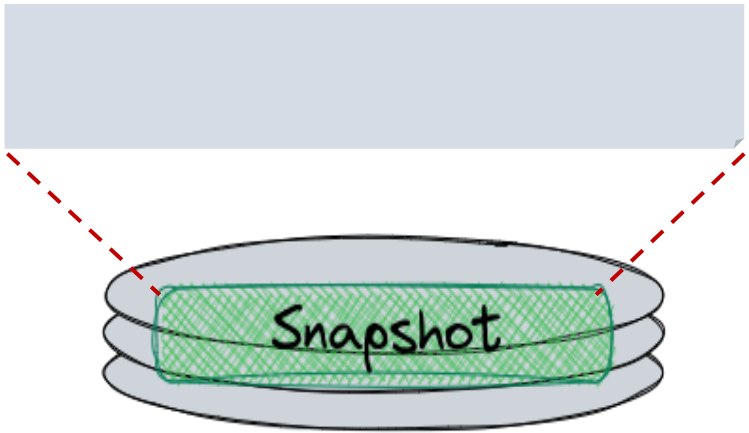


eBPF Capture

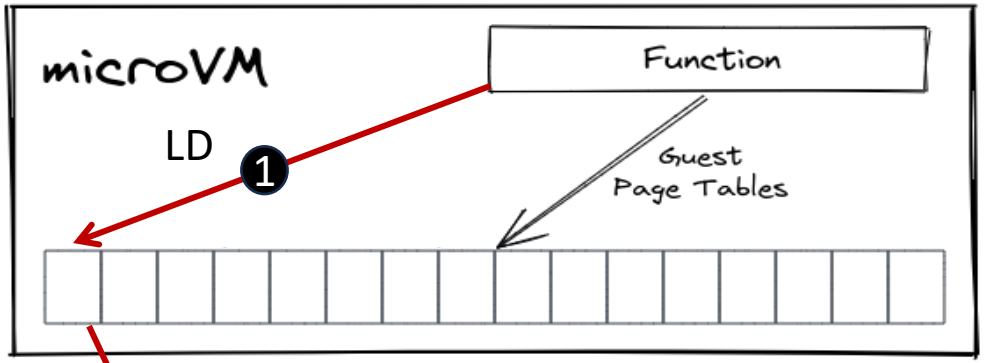


Host Kernel

OS Page Cache



eBPF Capture

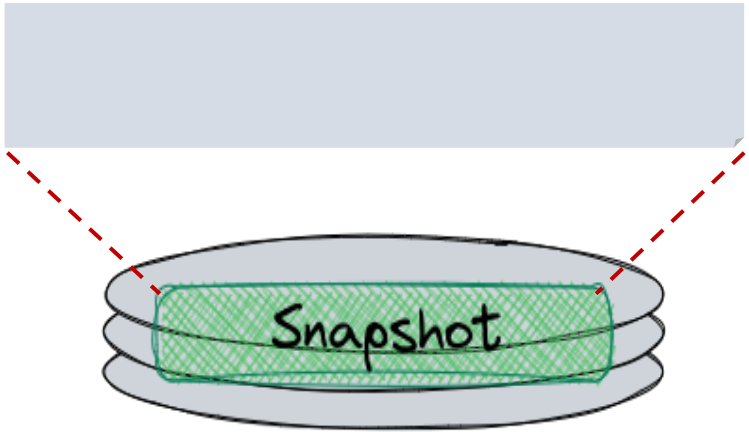


Nested Fault 2

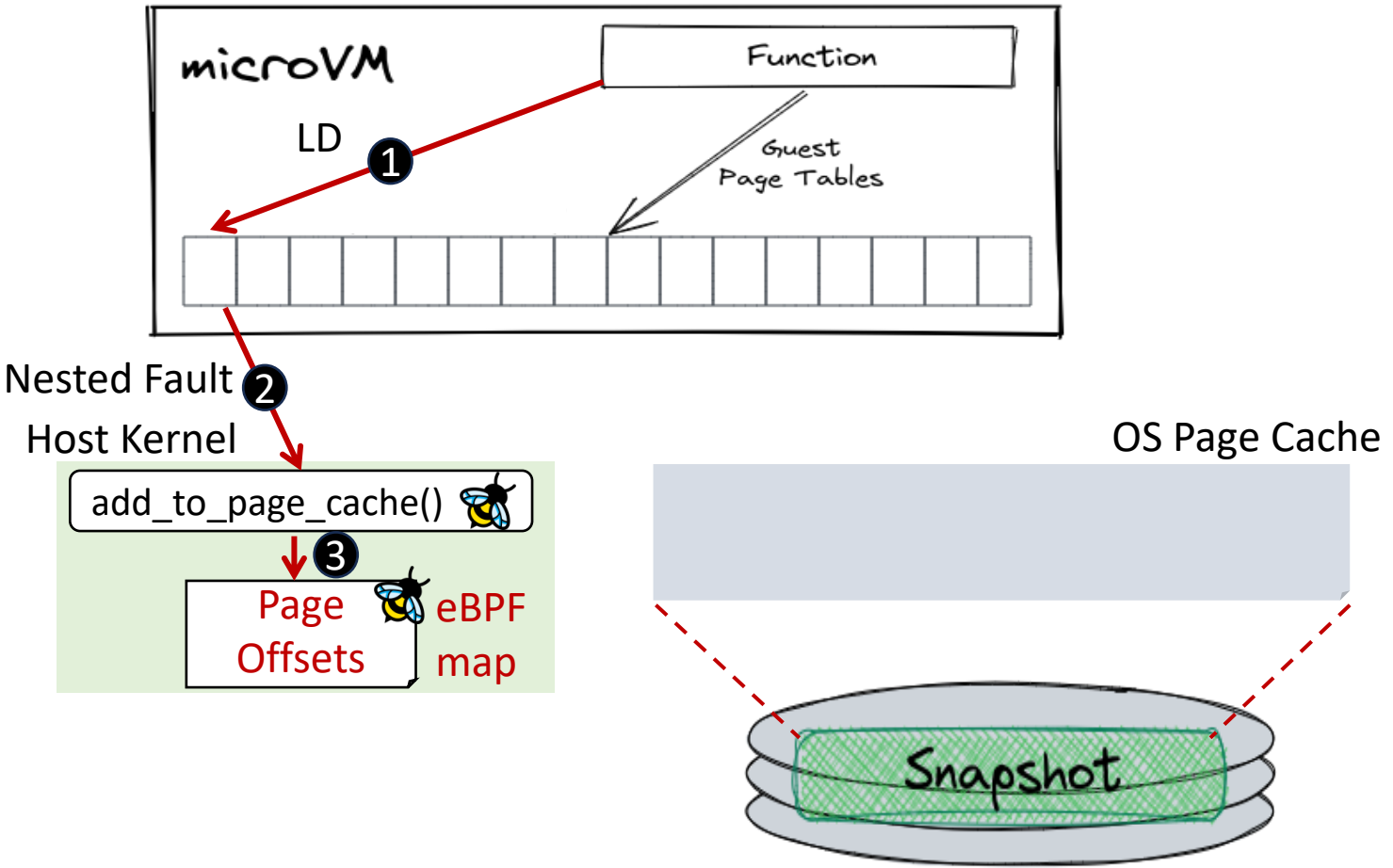
Host Kernel

add_to_page_cache()

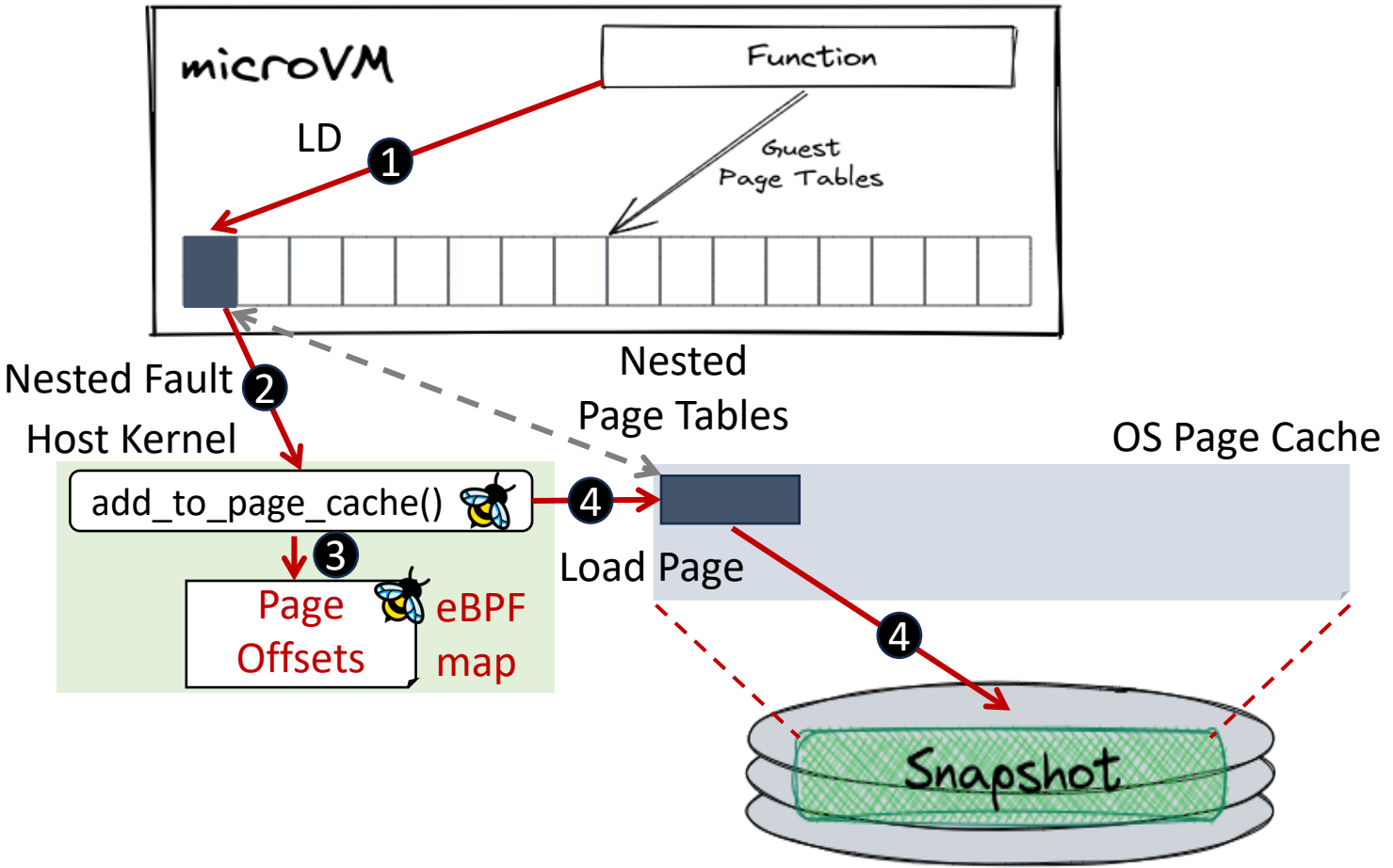
OS Page Cache



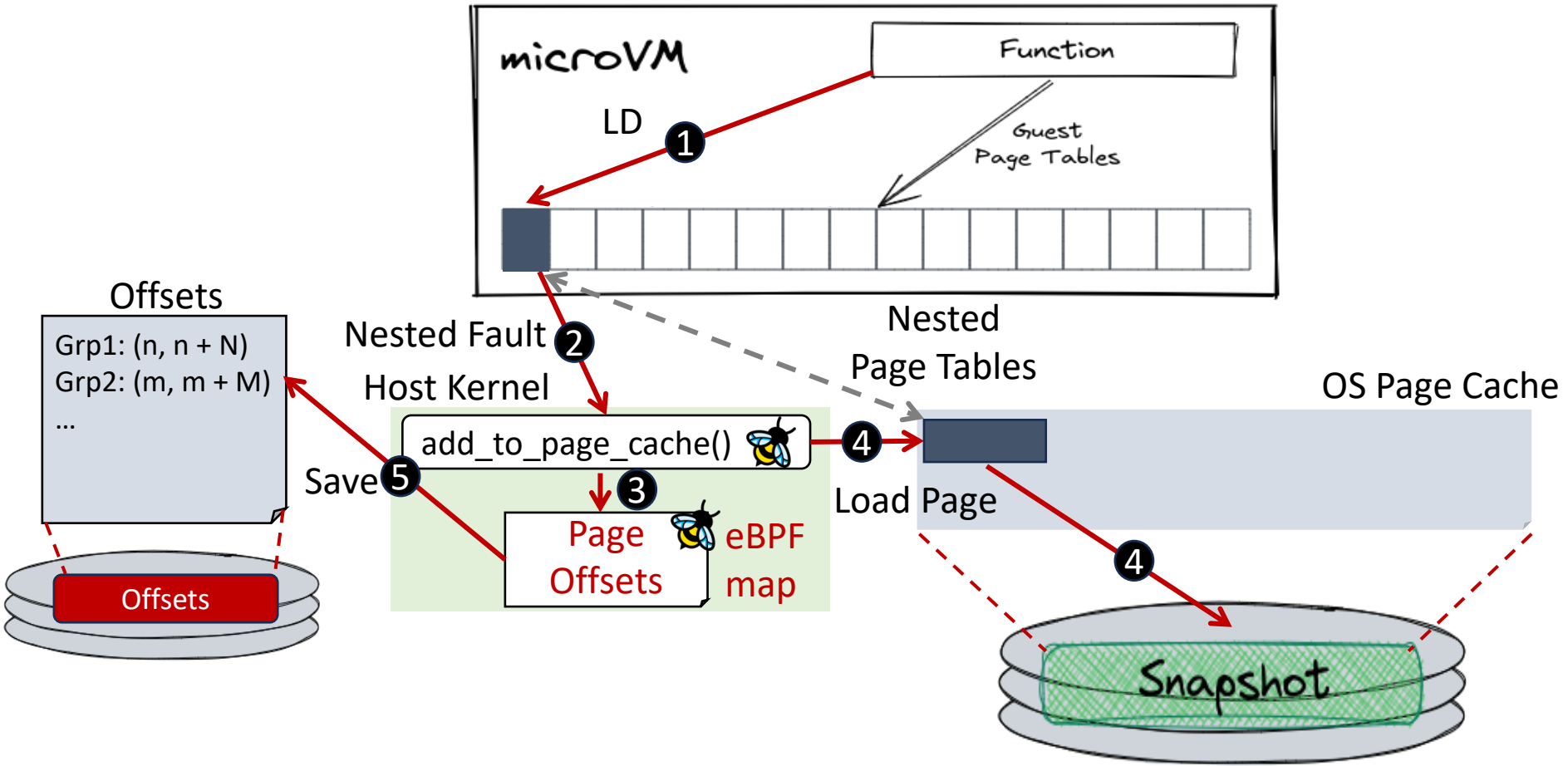
eBPF Capture



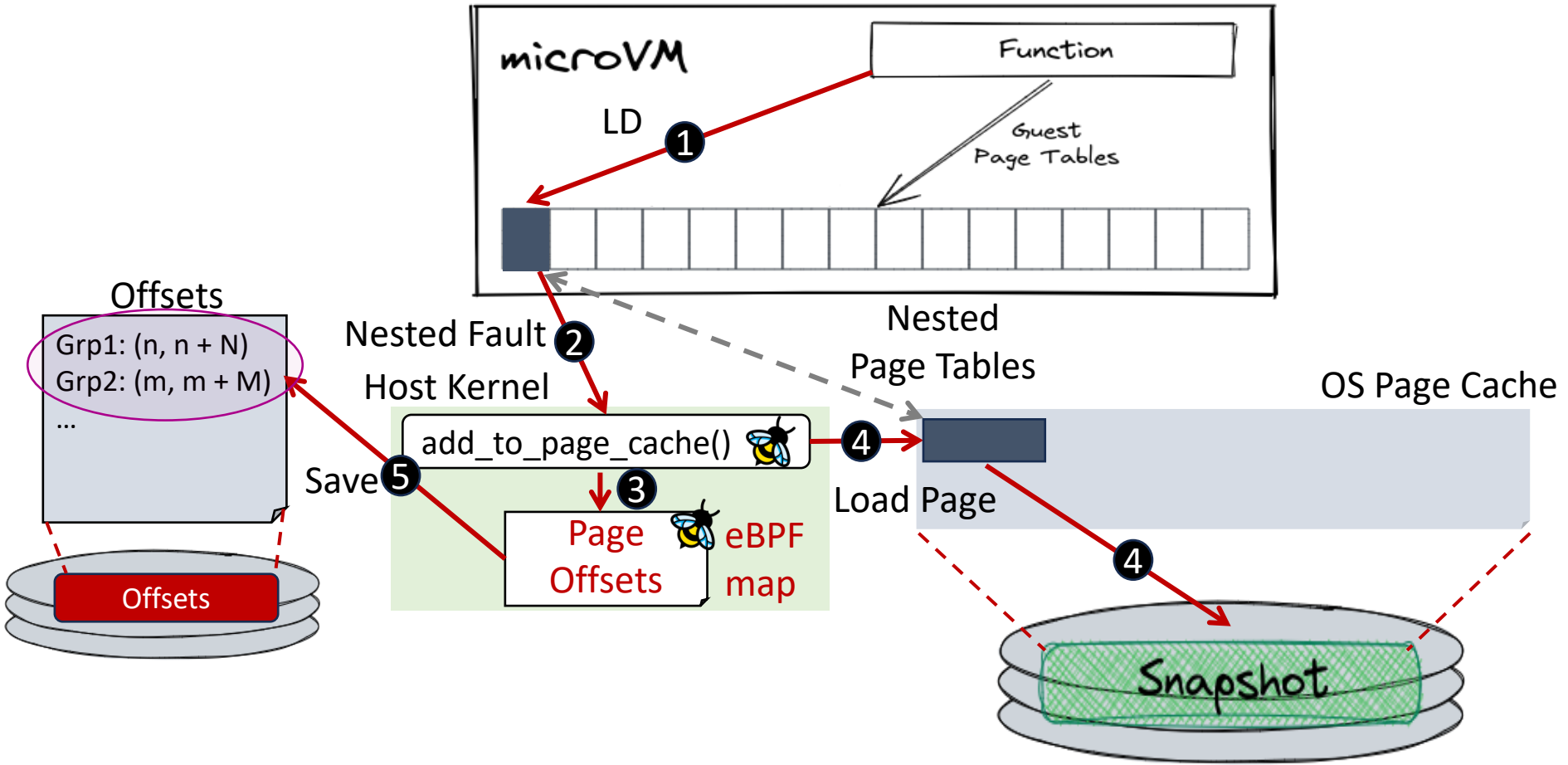
eBPF Capture



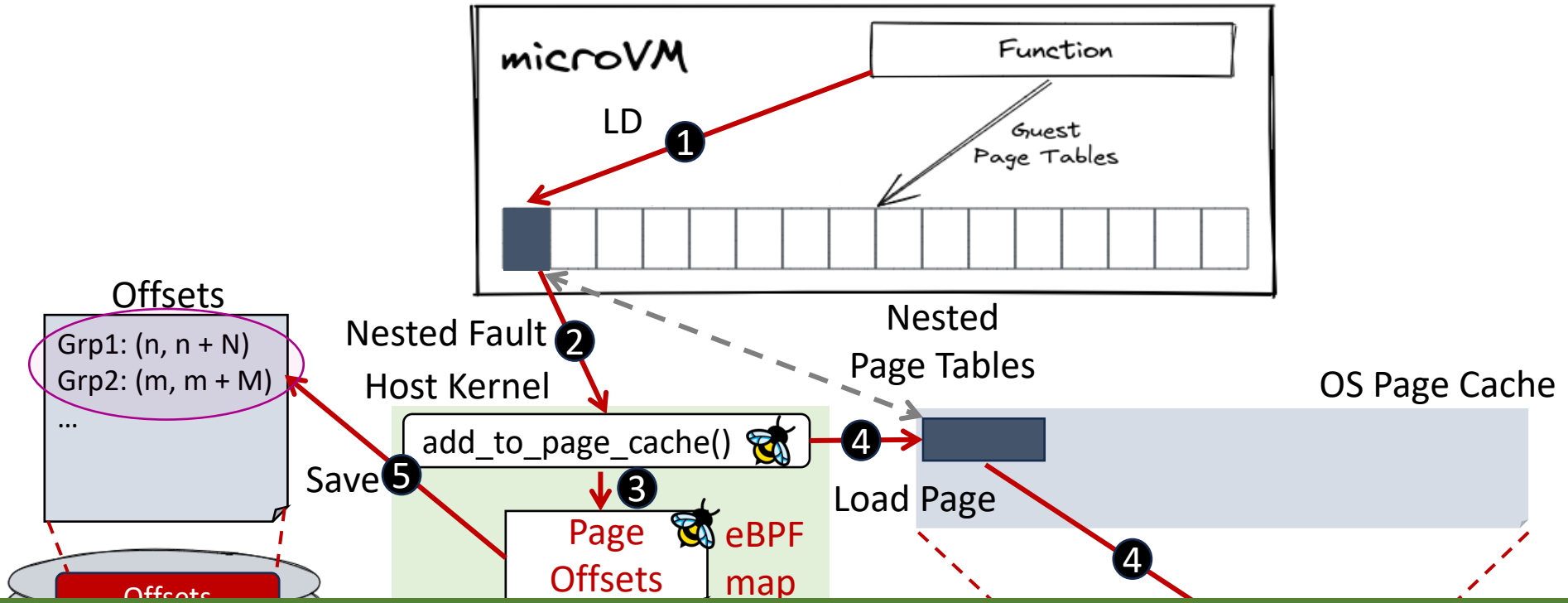
eBPF Capture



eBPF Capture

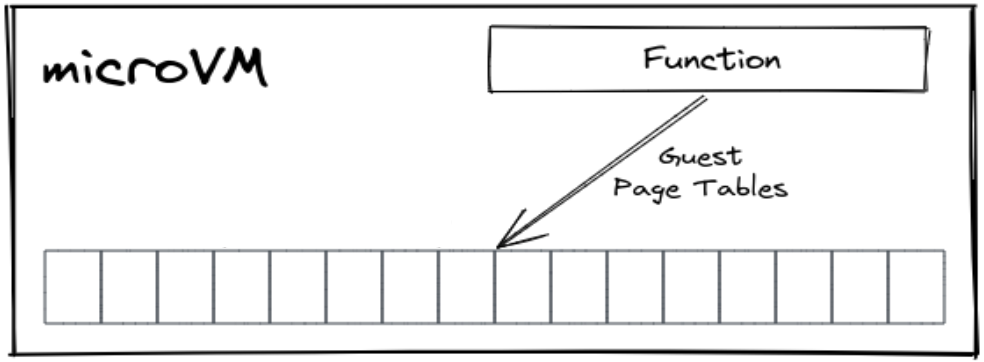


eBPF Capture



Precise and efficient working set capture in kernelspace

eBPF Prefetch

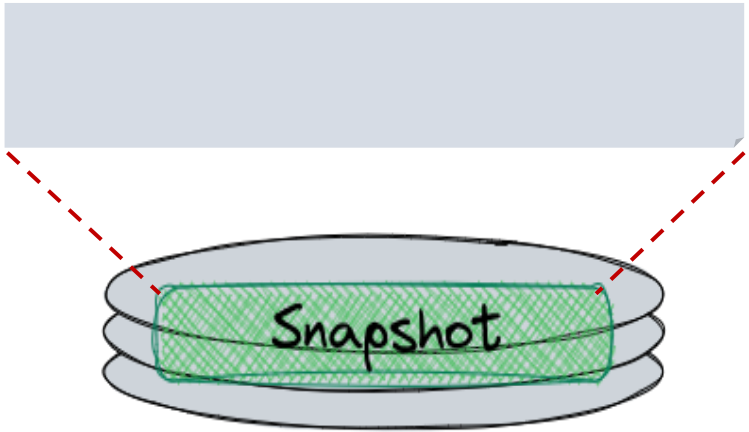


Offsets

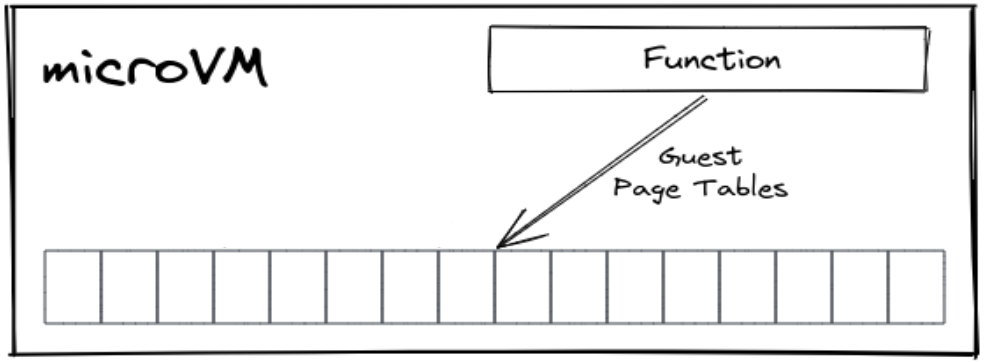
Grp1: (n, n + N)
Grp2: (m, m + M)
…

Host Kernel

OS Page Cache



eBPF Prefetch



snapbpf_load()

1

Offsets

Grp1: (n, n + N)
Grp2: (m, m + M)
…

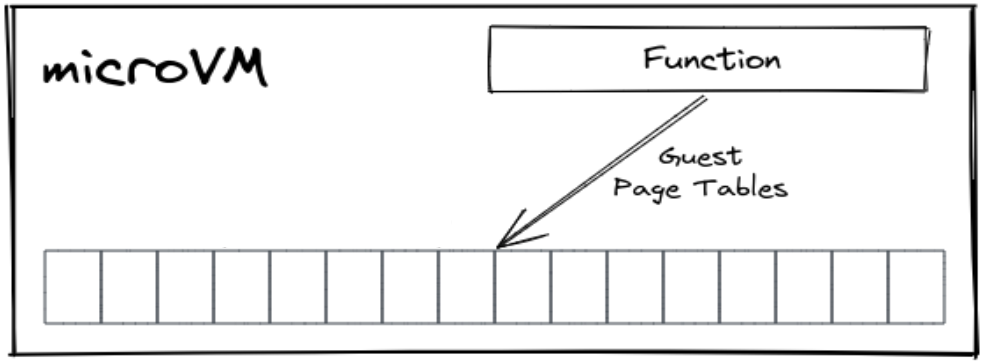
Offsets

Host Kernel

OS Page Cache

Snapshot

eBPF Prefetch



snapbpf_load()

1

Offsets

Grp1: (n, n + N)
Grp2: (m, m + M)
…

Host Kernel

OS Page Cache

Offsets

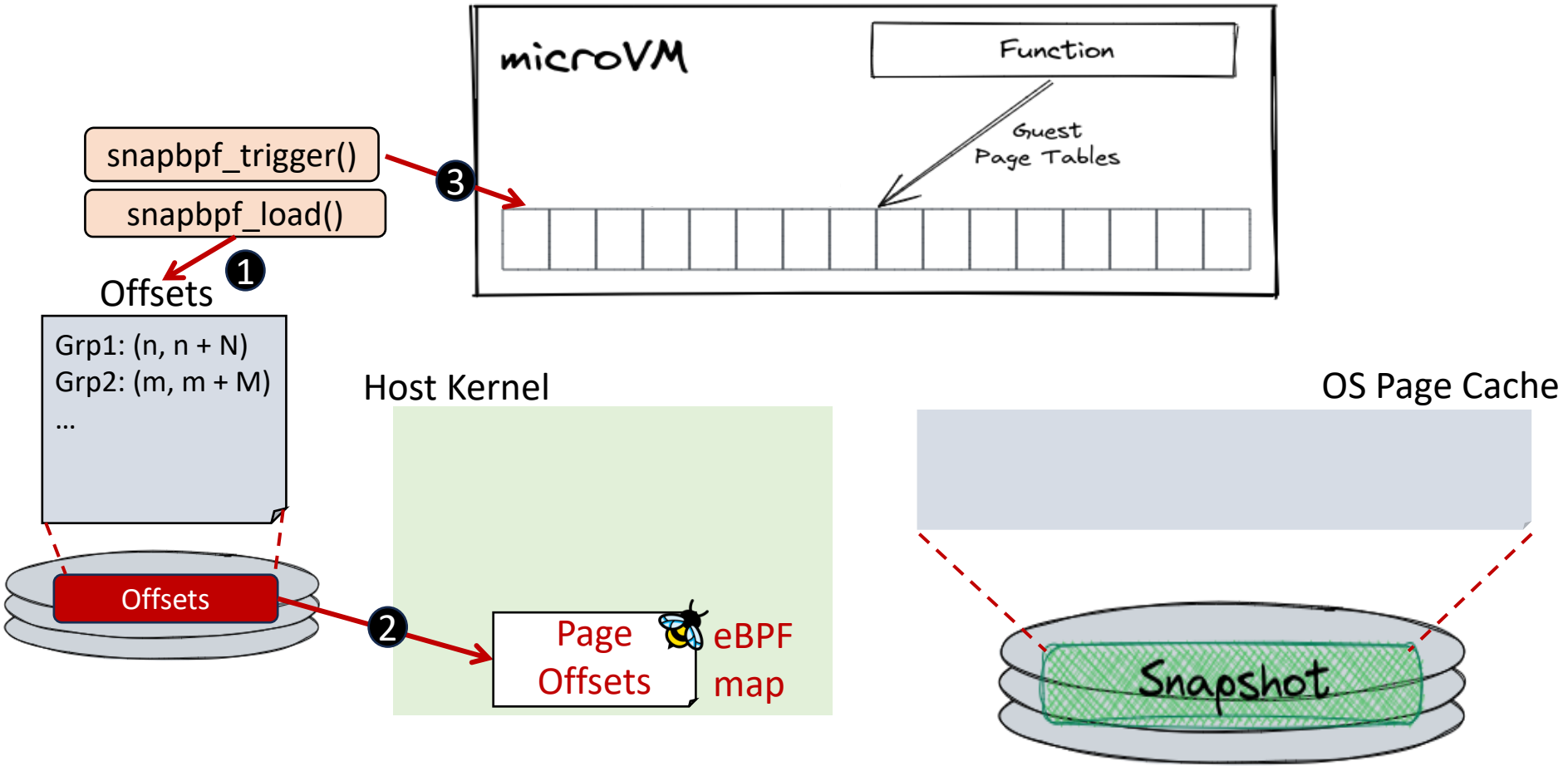
2

Page Offsets

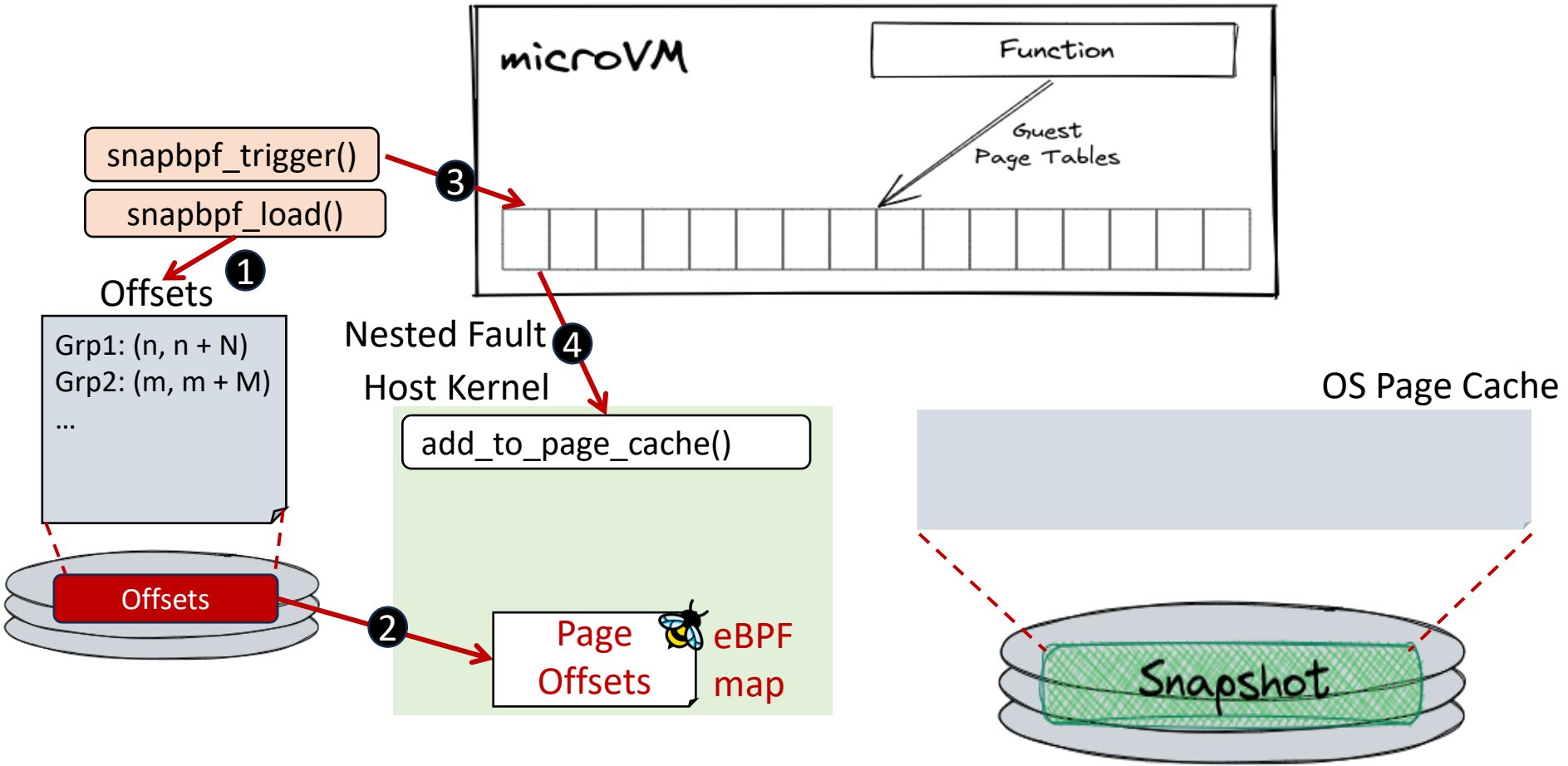
eBPF map

Snapshot

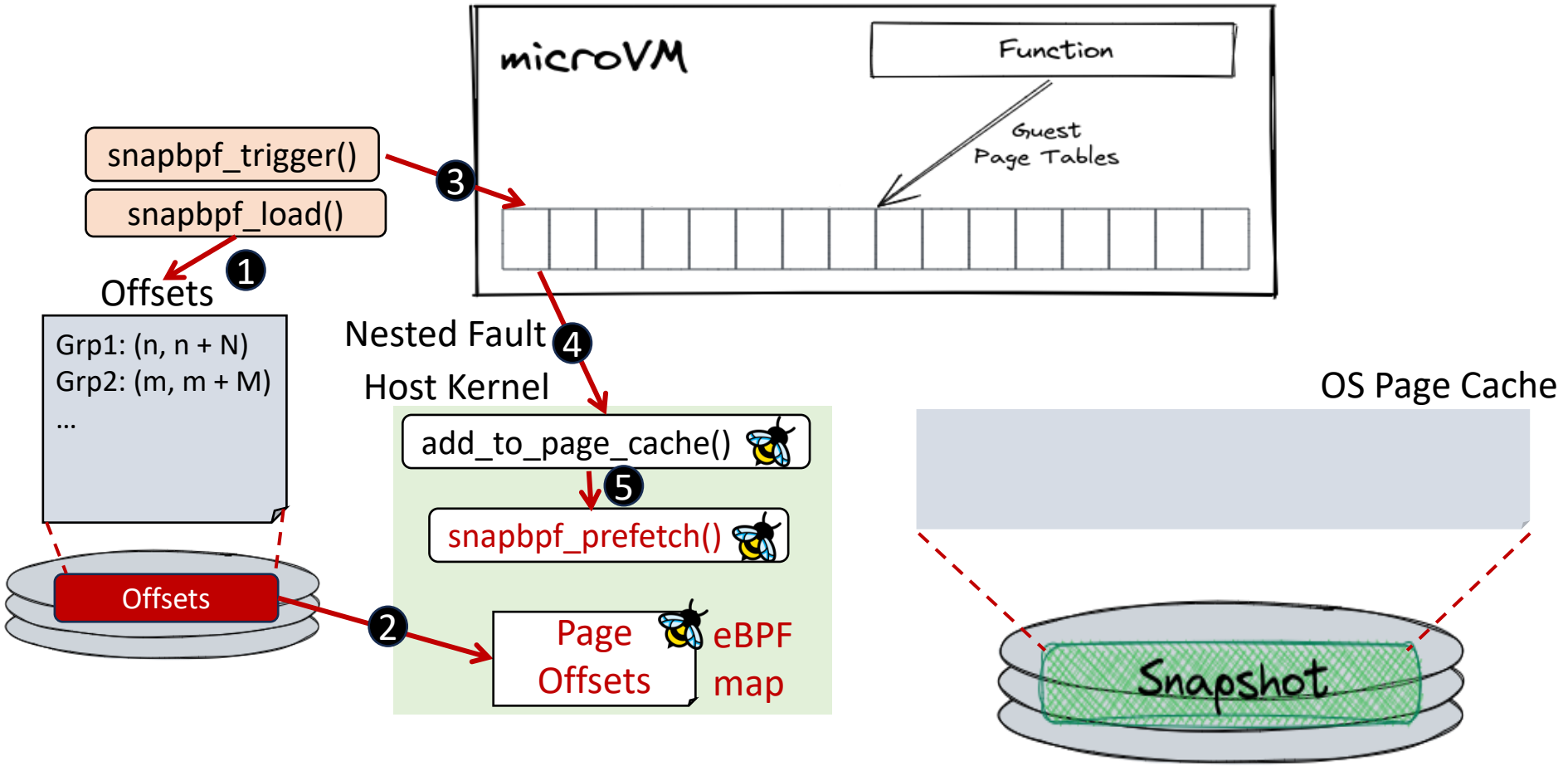
eBPF Prefetch



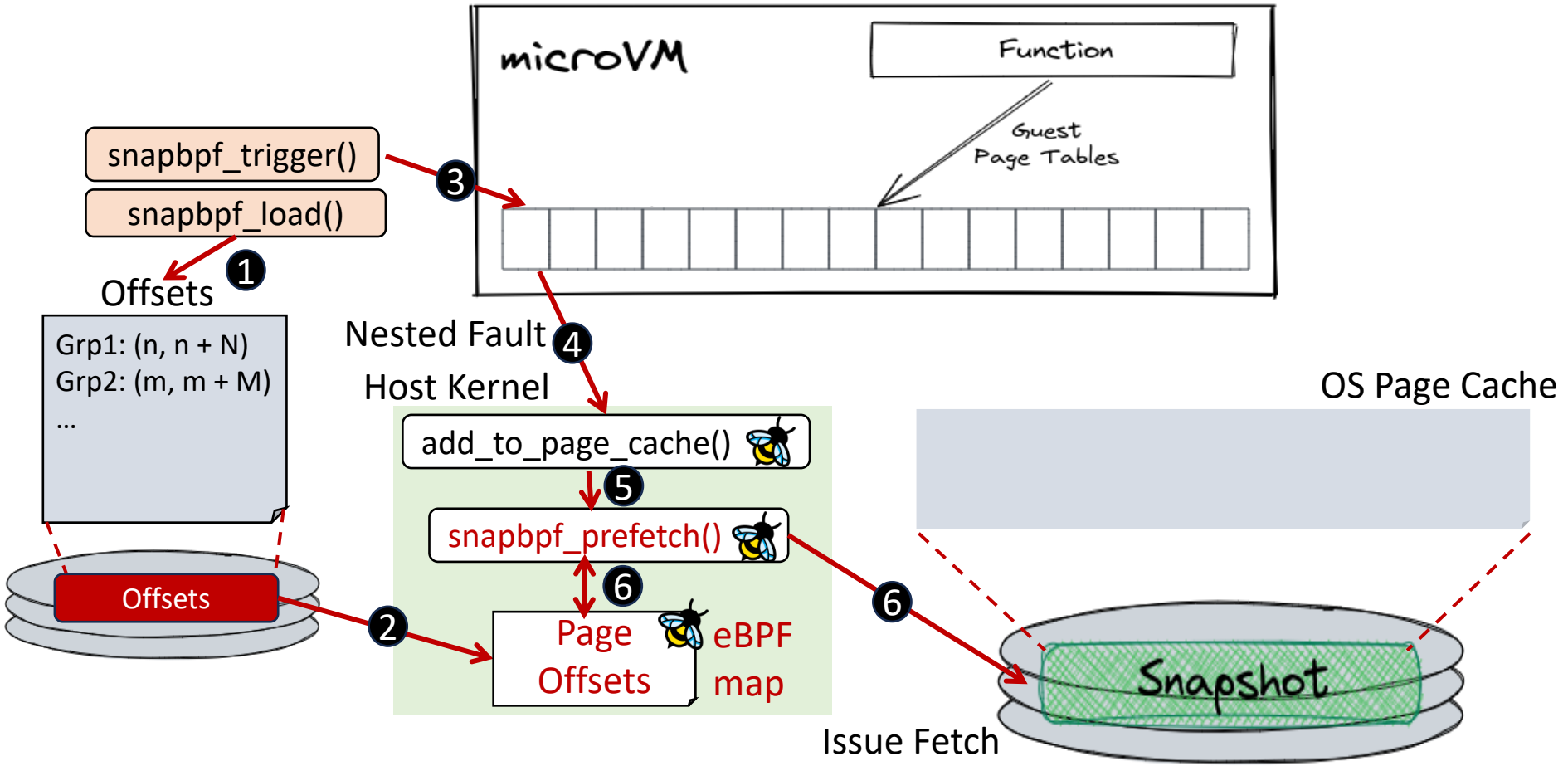
eBPF Prefetch



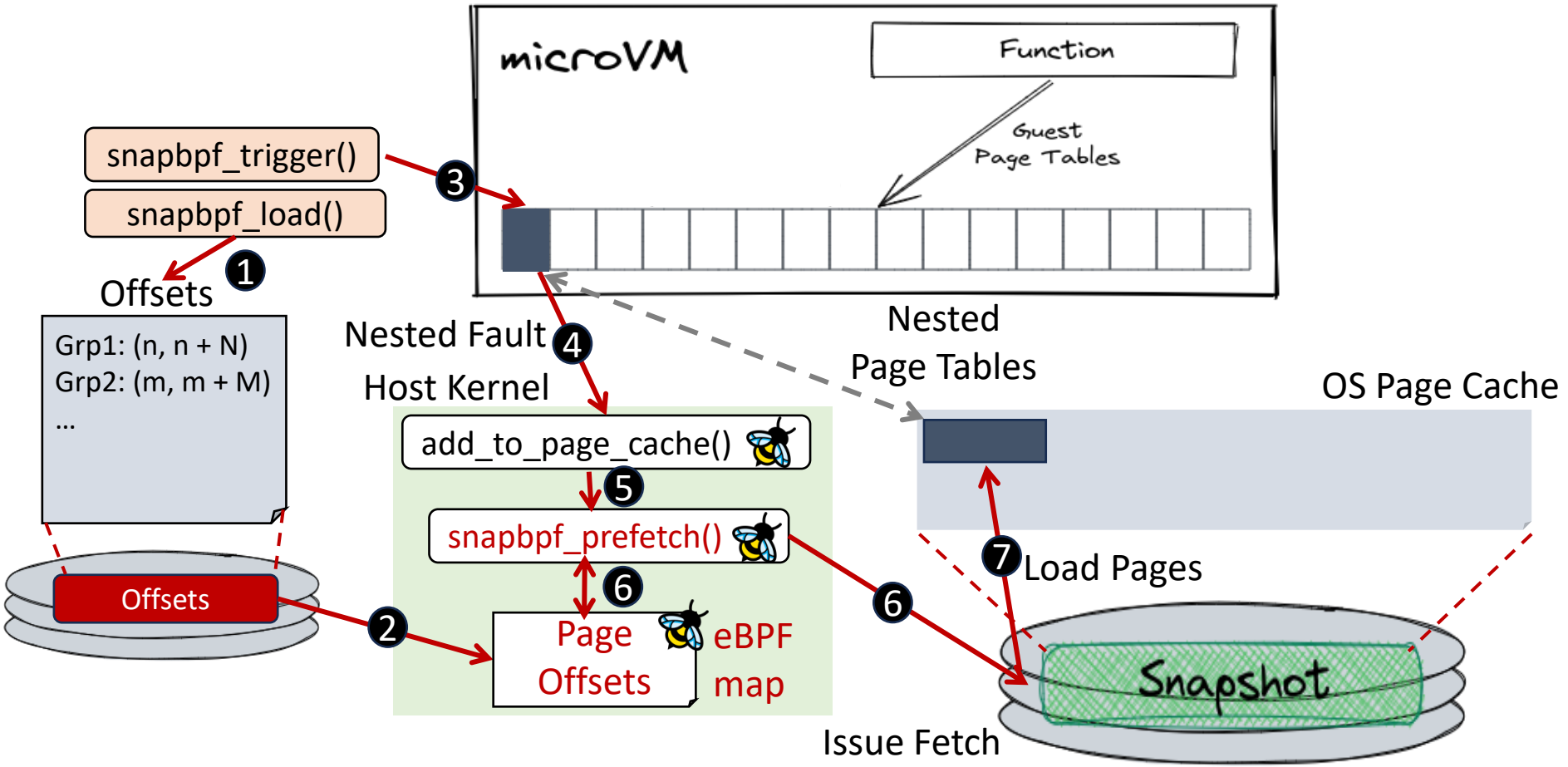
eBPF Prefetch



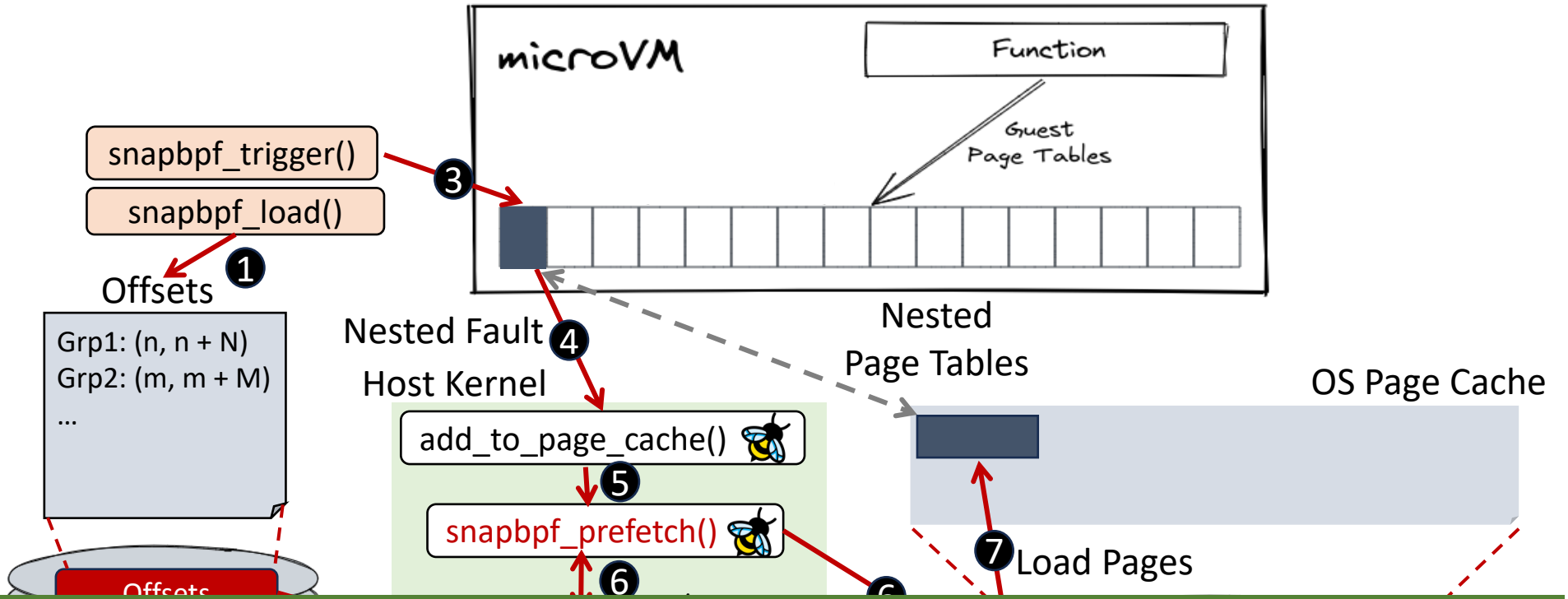
eBPF Prefetch



eBPF Prefetch



eBPF Prefetch



Fetch requests are asynchronous and overlap with microVM setup

Outline

→ FaaS Snapshotting

→ Working Set Prefetching

→ SnapBPF

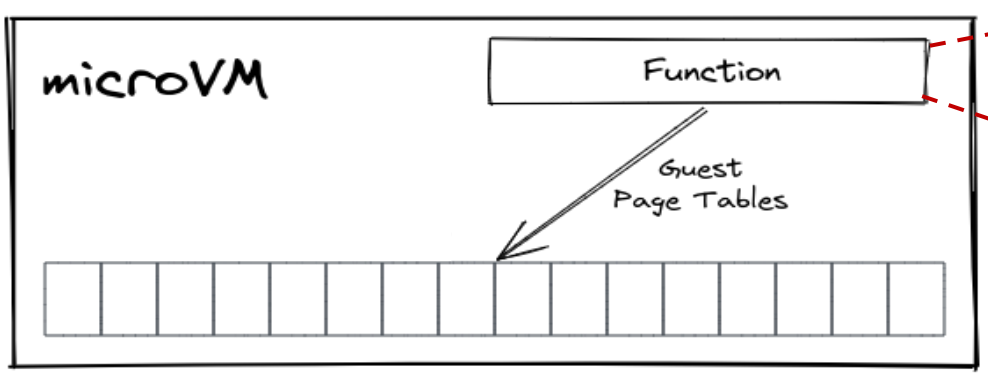
i. eBPF capture and prefetch

ii. PV free page filtering

→ Evaluation

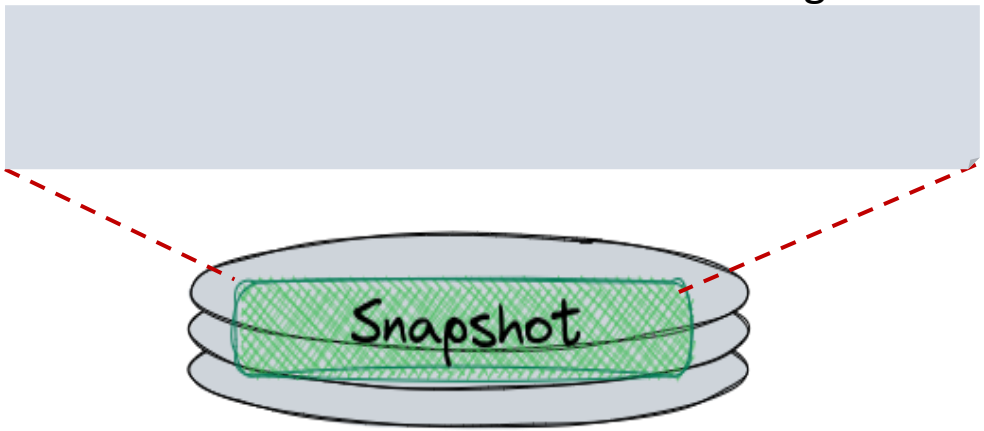
→ Conclusion

Free page filtering

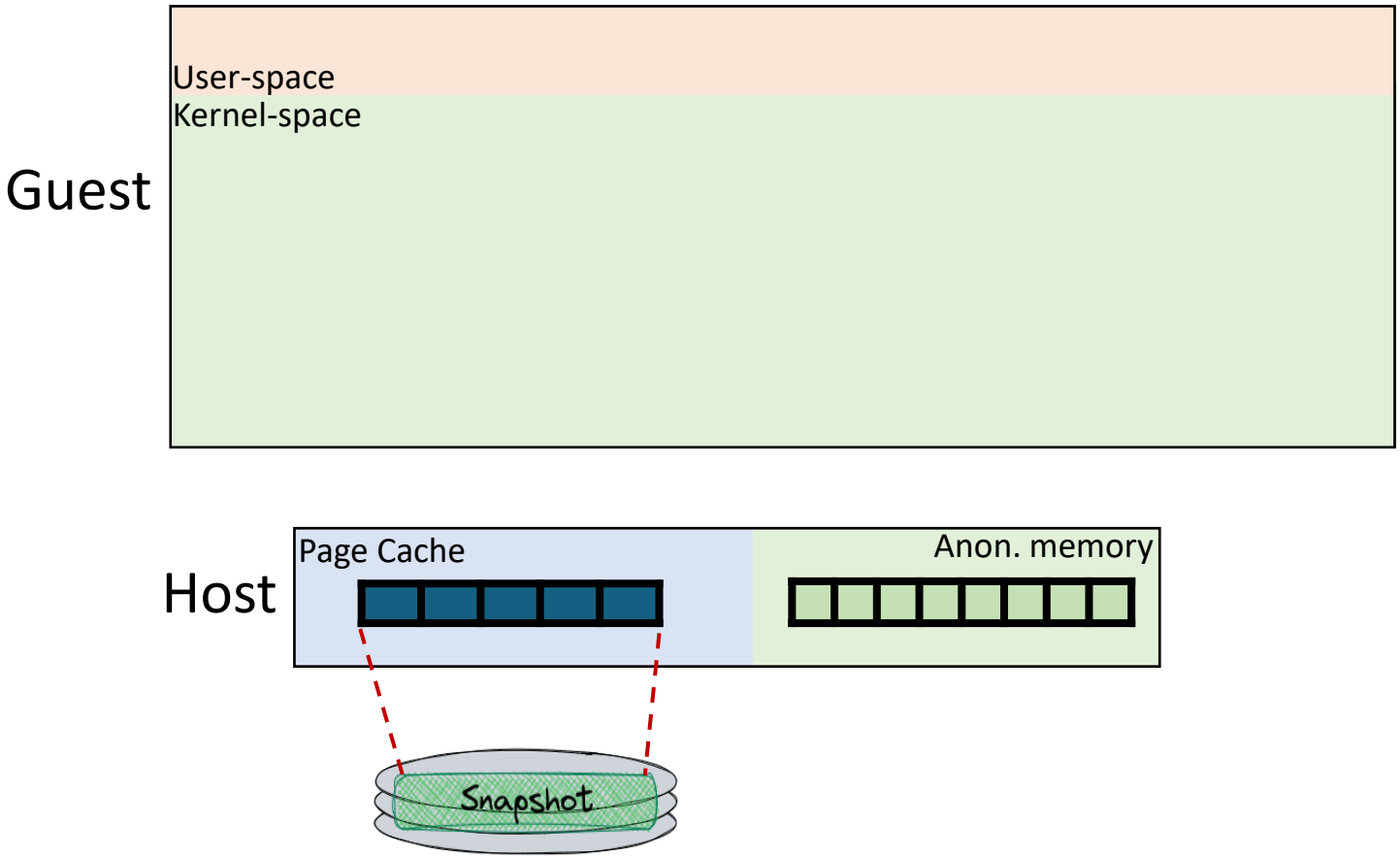



```
fn(input):
  output = malloc()
  return compute(input, &output)
```

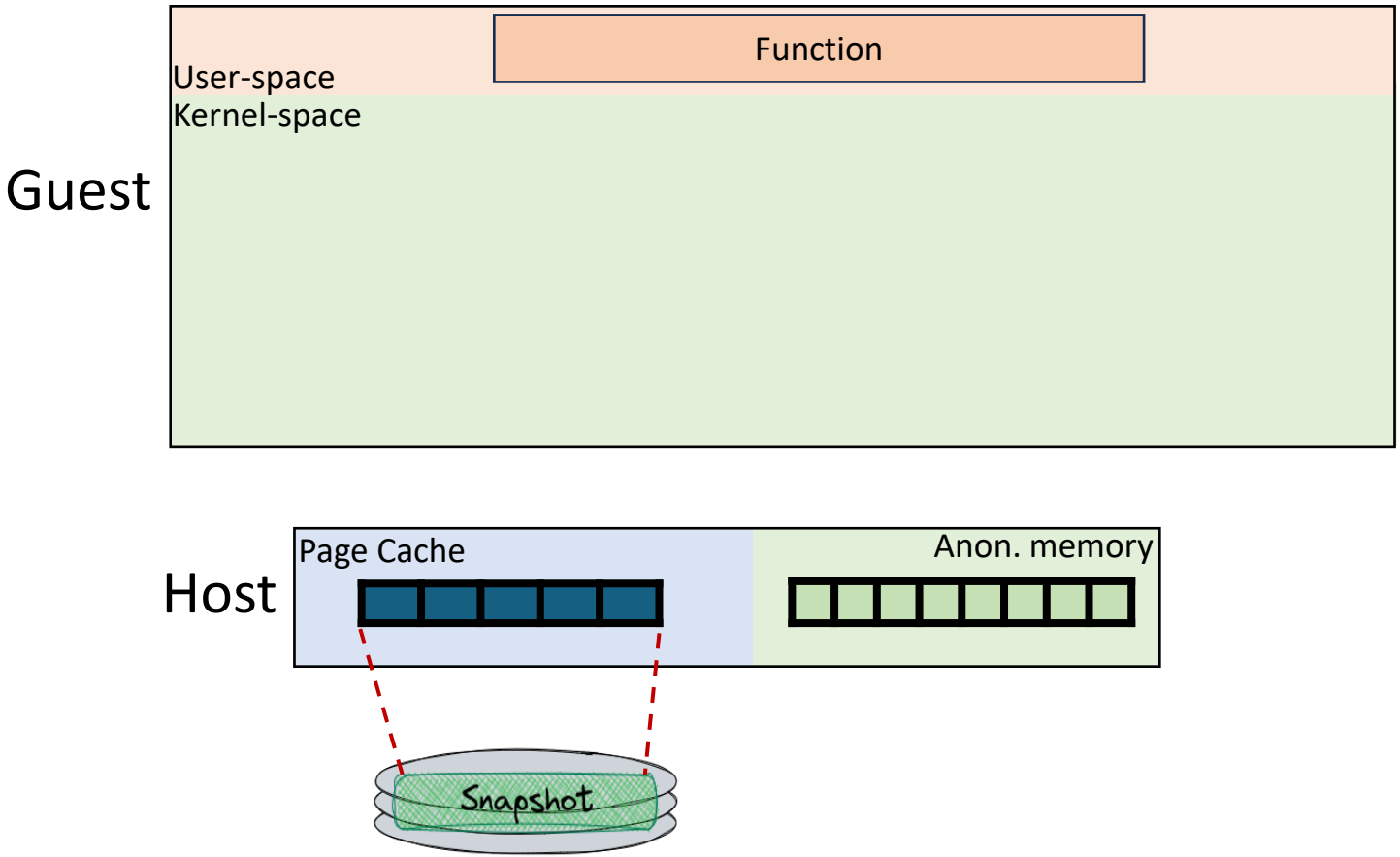
OS Page Cache



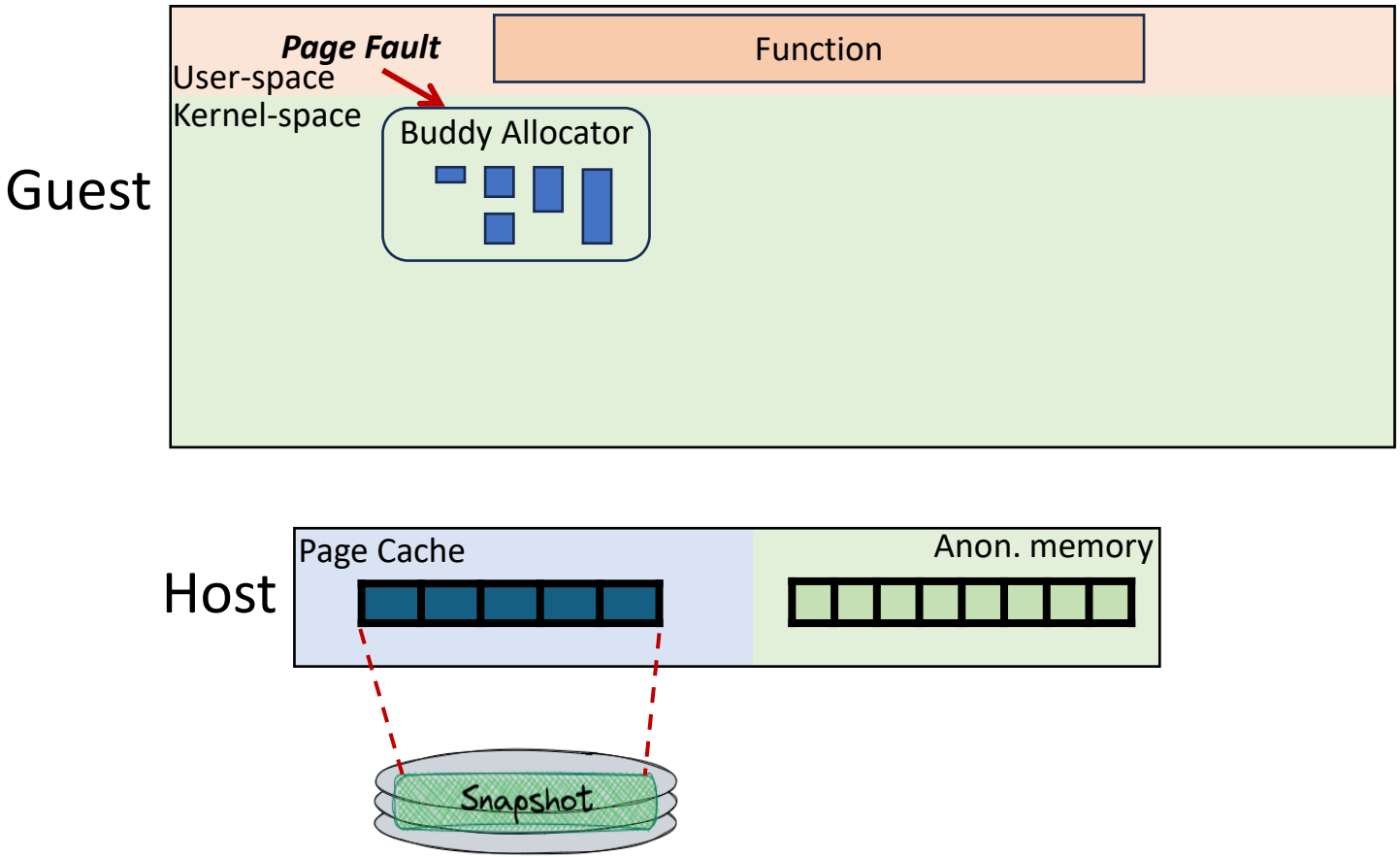
PV PTE marking



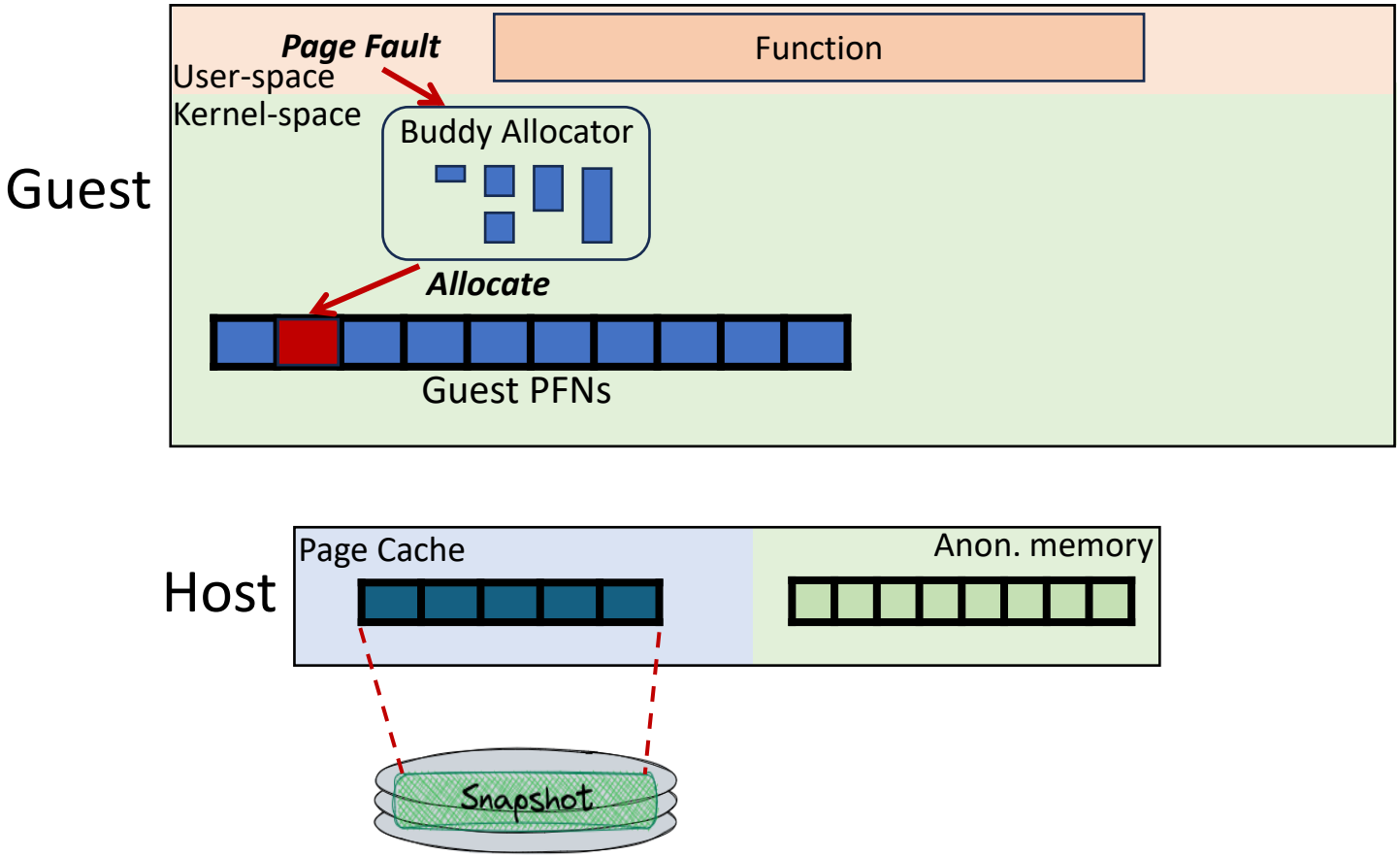
PV PTE marking



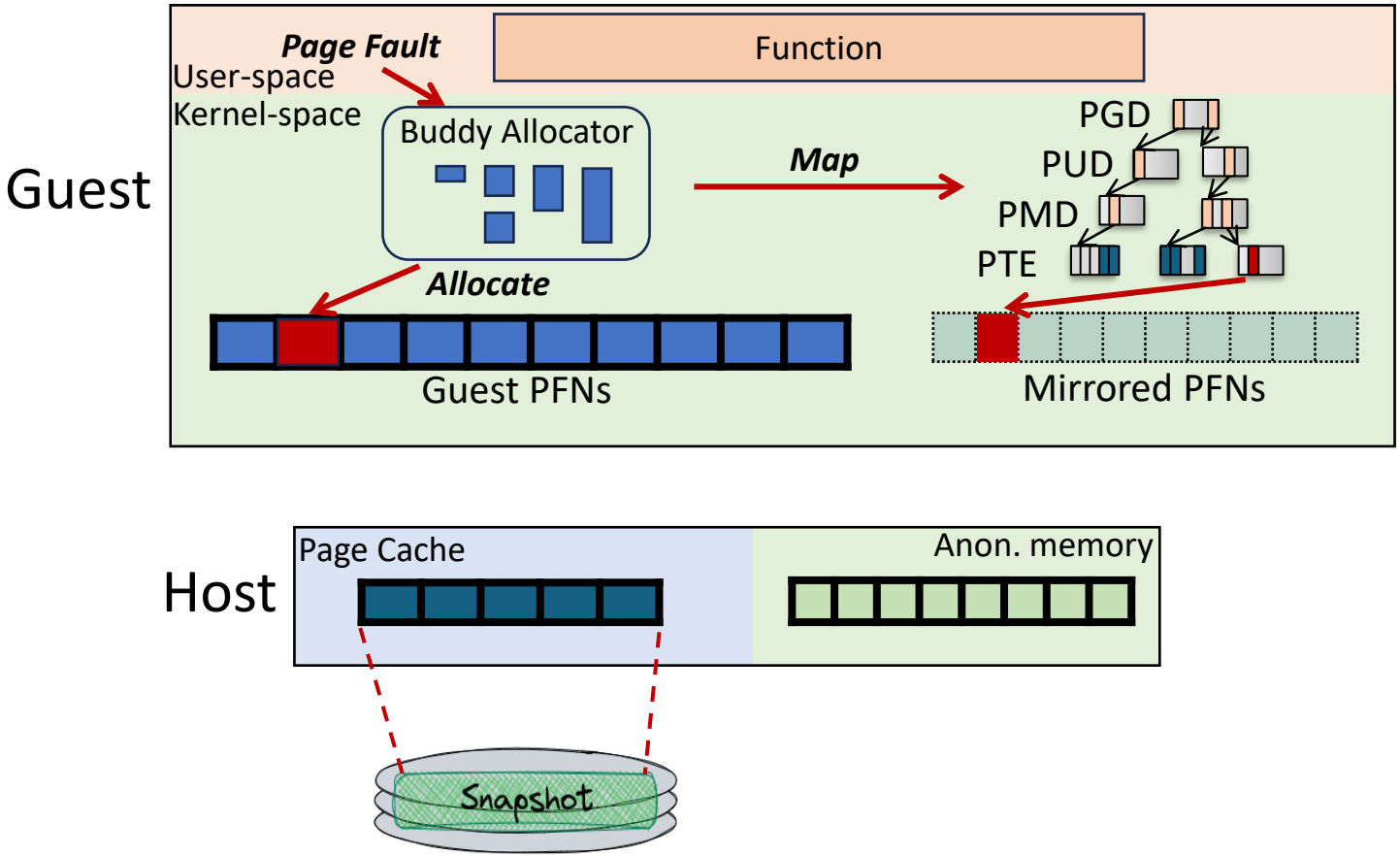
PV PTE marking



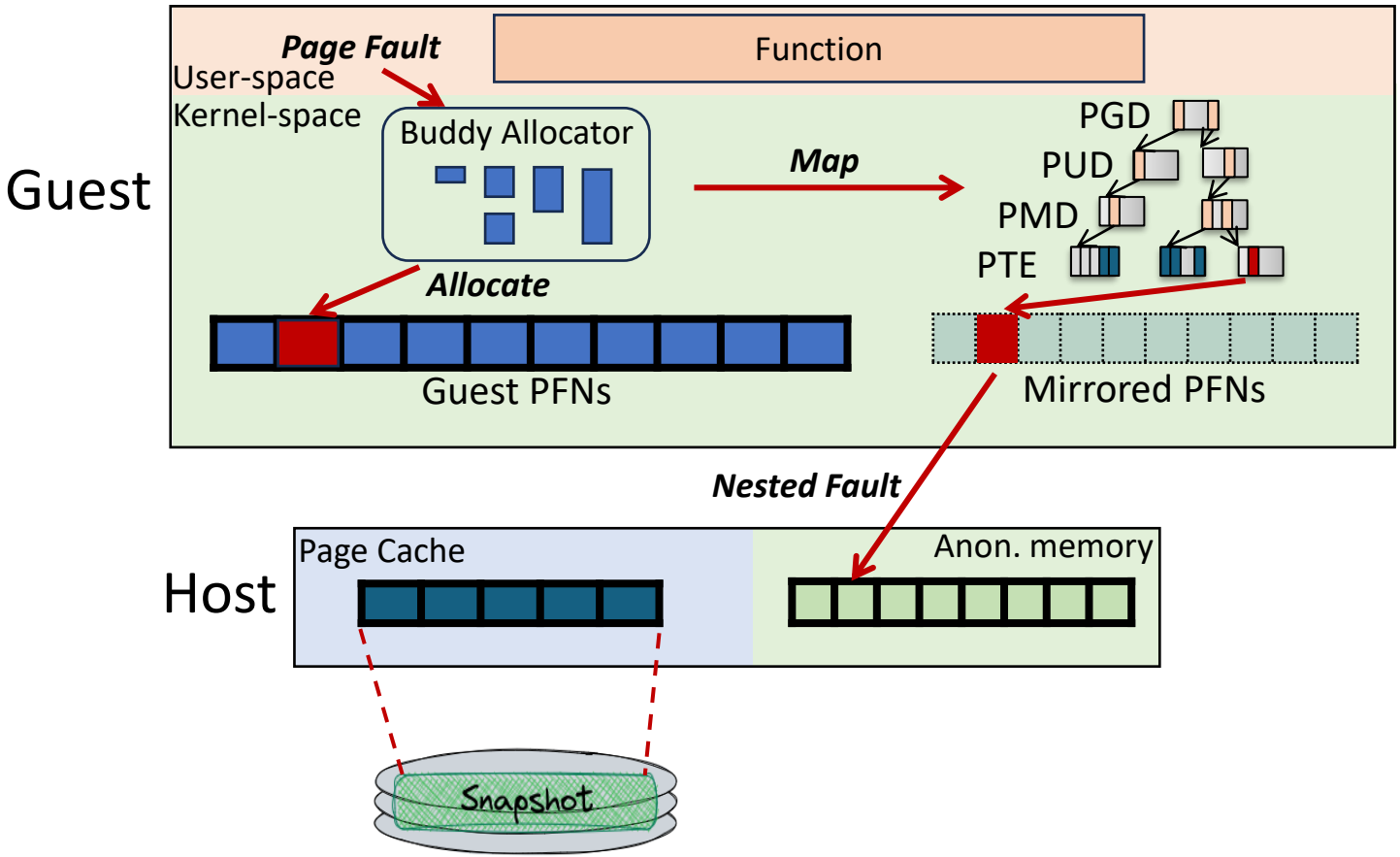
PV PTE marking



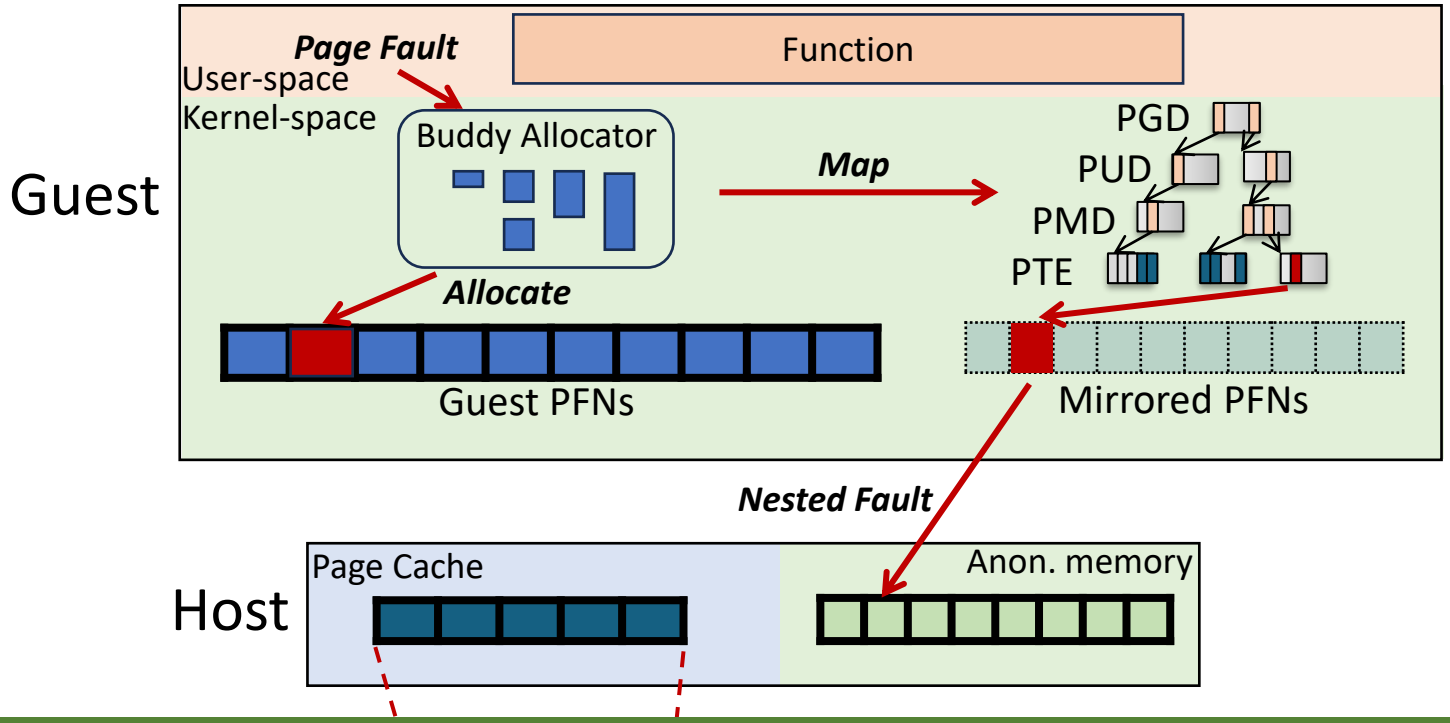
PV PTE marking



PV PTE marking



PV PTE marking



Online free page filtering with minimal overhead

Our proposal: SnapBPF

	Mechanism	Mode	WS serialization	WS deduplication	Free Page Filtering
REAP **Faast**	userfaultfd	userspace	Required	No	None Metadada Scanning
FaaSnap	mincore mmap	userspace	Required	Yes	Memory Scanning
		Our proposal			
SnapBPF	eBPF	kernelspace	None	Yes	PV / Online

Outline

→ FaaS Snapshotting

→ Working Set Prefetching

→ SnapBPF

i. eBPF capture and prefetch

ii. PV free page filtering

→ Evaluation

→ Conclusion

Experimental Setup

Experimental Setup

→ Implemented in Linux v6.3 and Firecracker v1.11

Experimental Setup

→ Implemented in Linux v6.3 and Firecracker v1.11

→ FunctionBench and real-world FaaS workloads

Experimental Setup

→ Implemented in Linux v6.3 and Firecracker v1.11

→ FunctionBench and real-world FaaS workloads

→ AMD Zen2 evaluation testbed (AMD EPYC 7402)

Experimental Setup

- → Implemented in Linux v6.3 and Firecracker v1.11
- → FunctionBench and real-world FaaS workloads
- → AMD Zen2 evaluation testbed (AMD EPYC 7402)
- → Micron 5300 TLC NAND flash SATA SSD
  - → 95K random 4K read IOPS
  - → 540MB / sec 128K sequential read throughput

Evaluation Questions

Evaluation Questions

→ Can SnapBPF match SOTA performance without sequentially serialized working sets?

Evaluation Questions

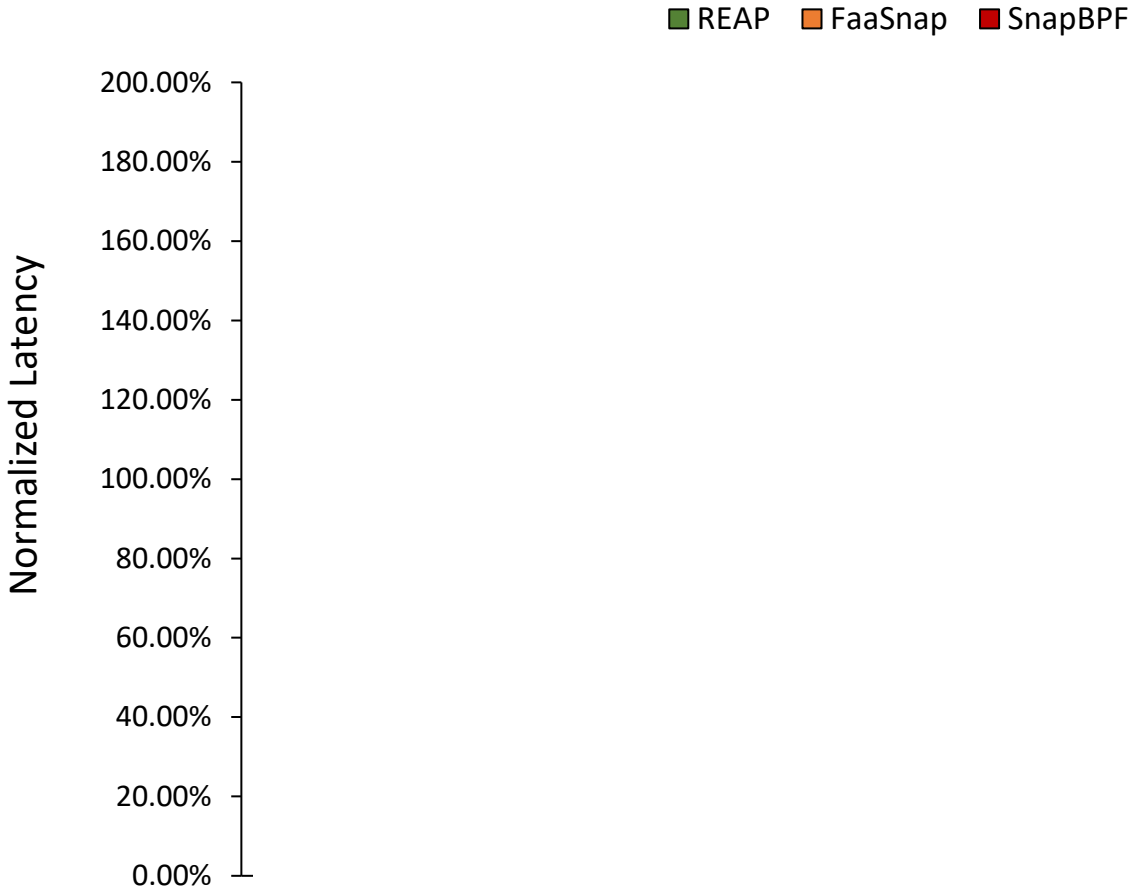
→ Can SnapBPF match SOTA performance without sequentially serialized working sets?

→ Can SnapBPF effectively deduplicate working sets in memory while sustaining performance?

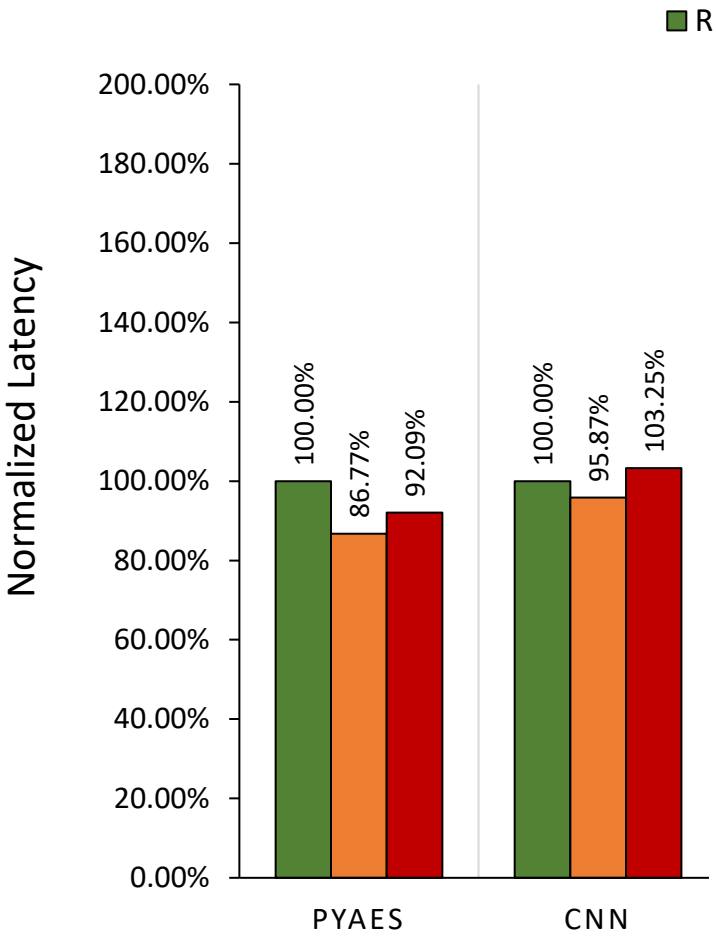
Single function E2E latency

Normalized Latency

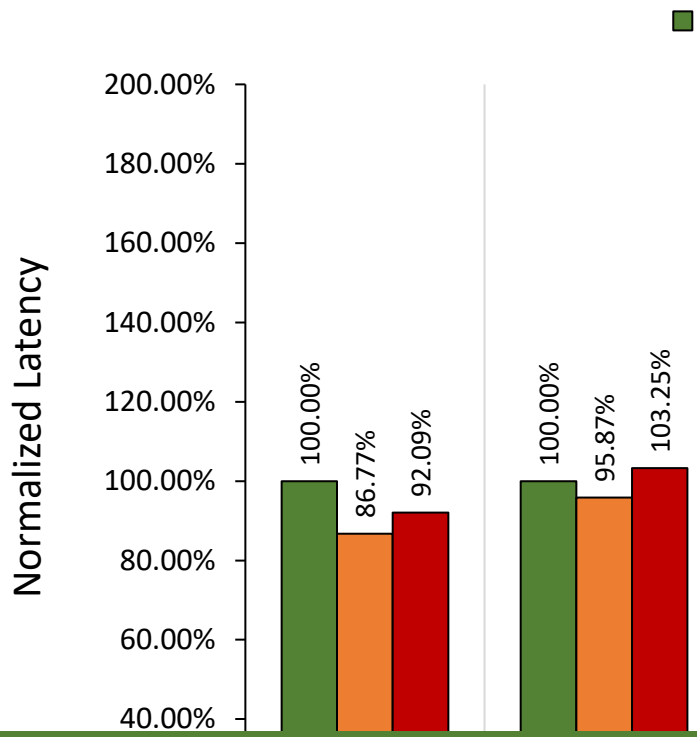
Single function E2E latency



Single function E2E latency

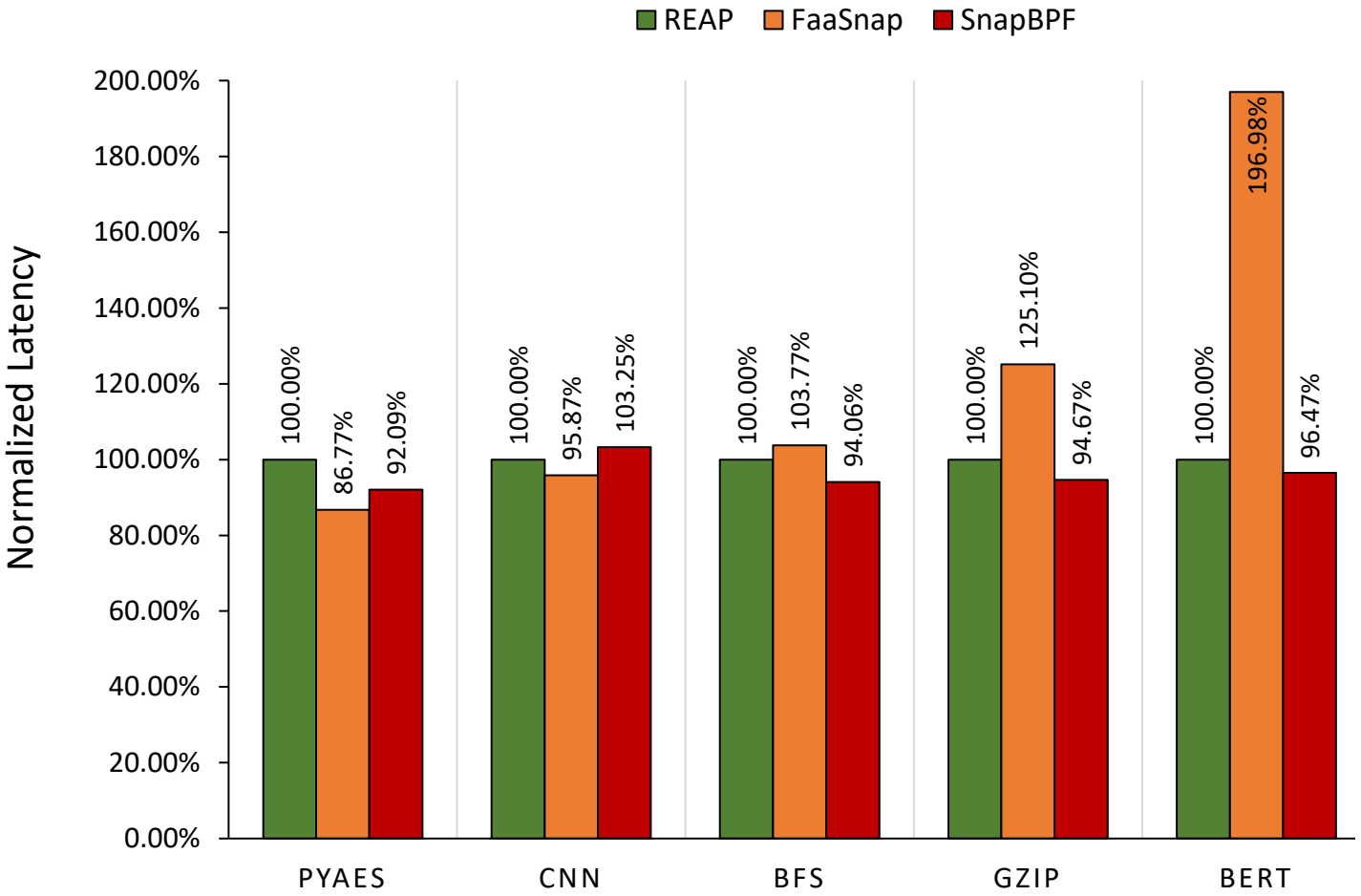


Single function E2E latency

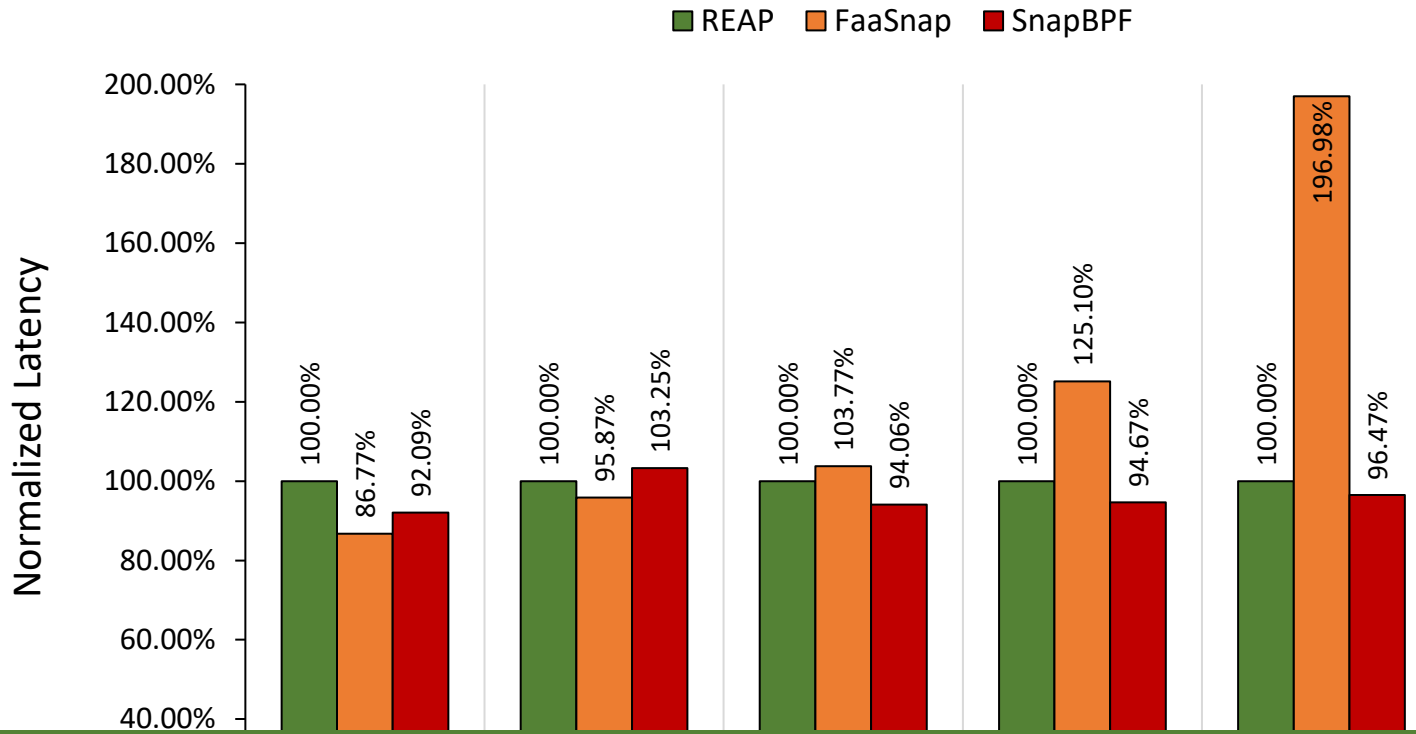


SnapBPF matches SOTA performance for small functions

Single function E2E latency

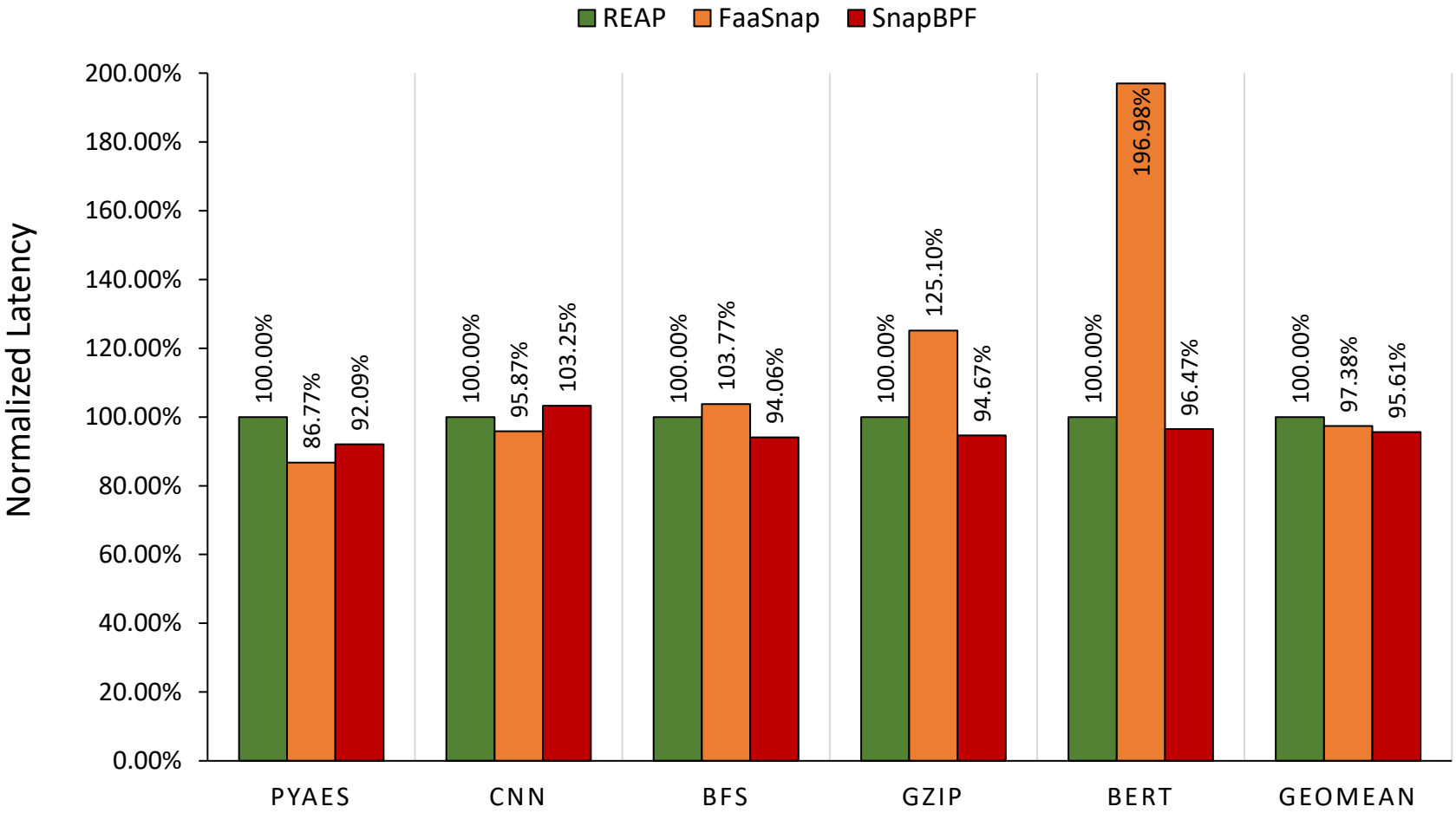


Single function E2E latency

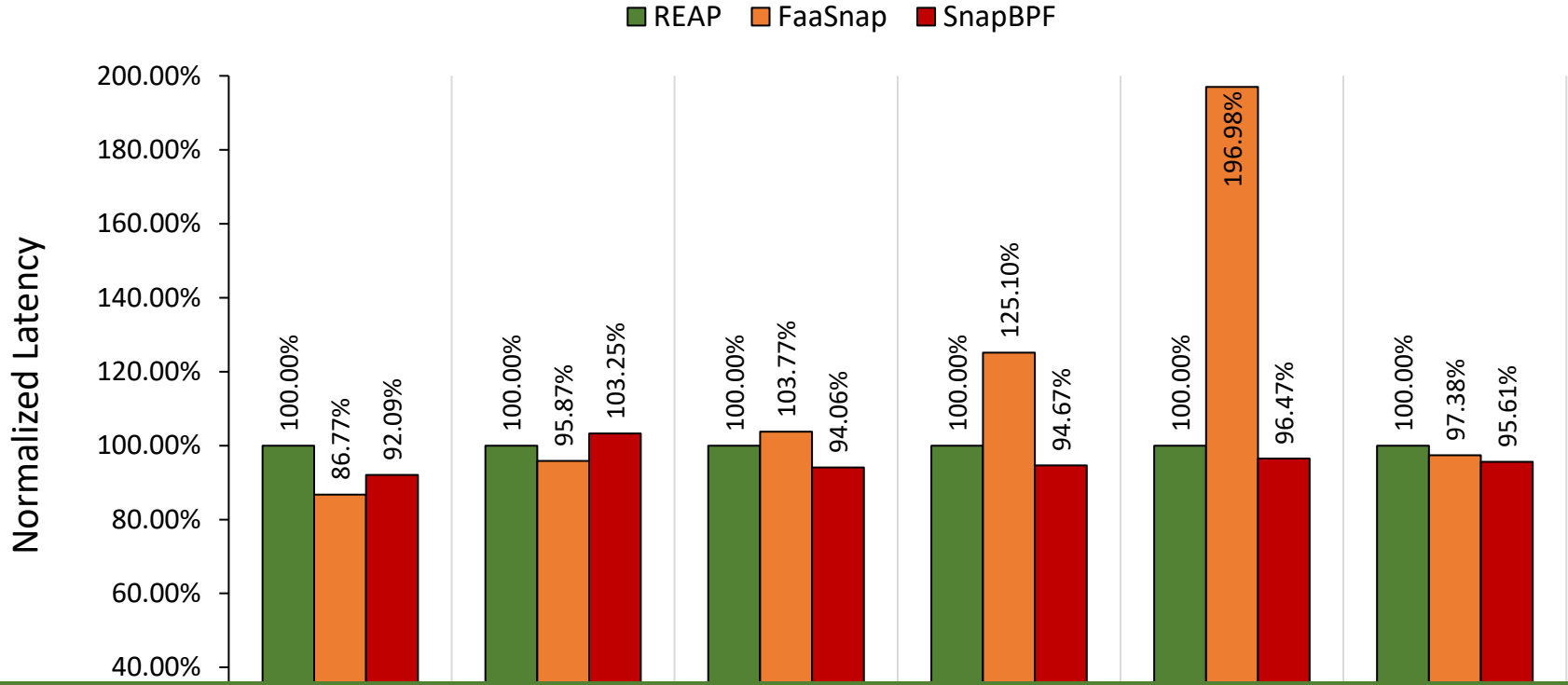


SnapBPF is resilient to working set fragmentation

Single function E2E latency



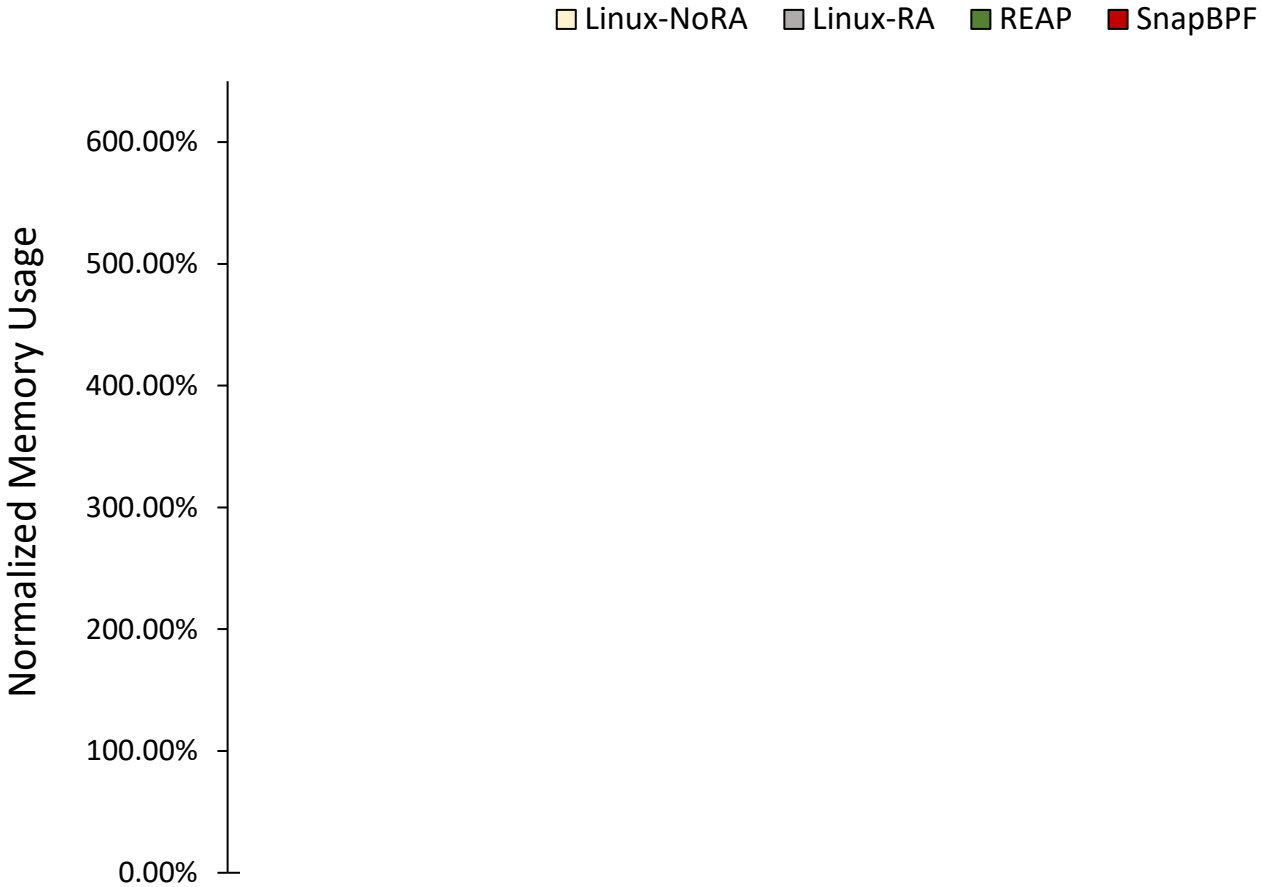
Single function E2E latency



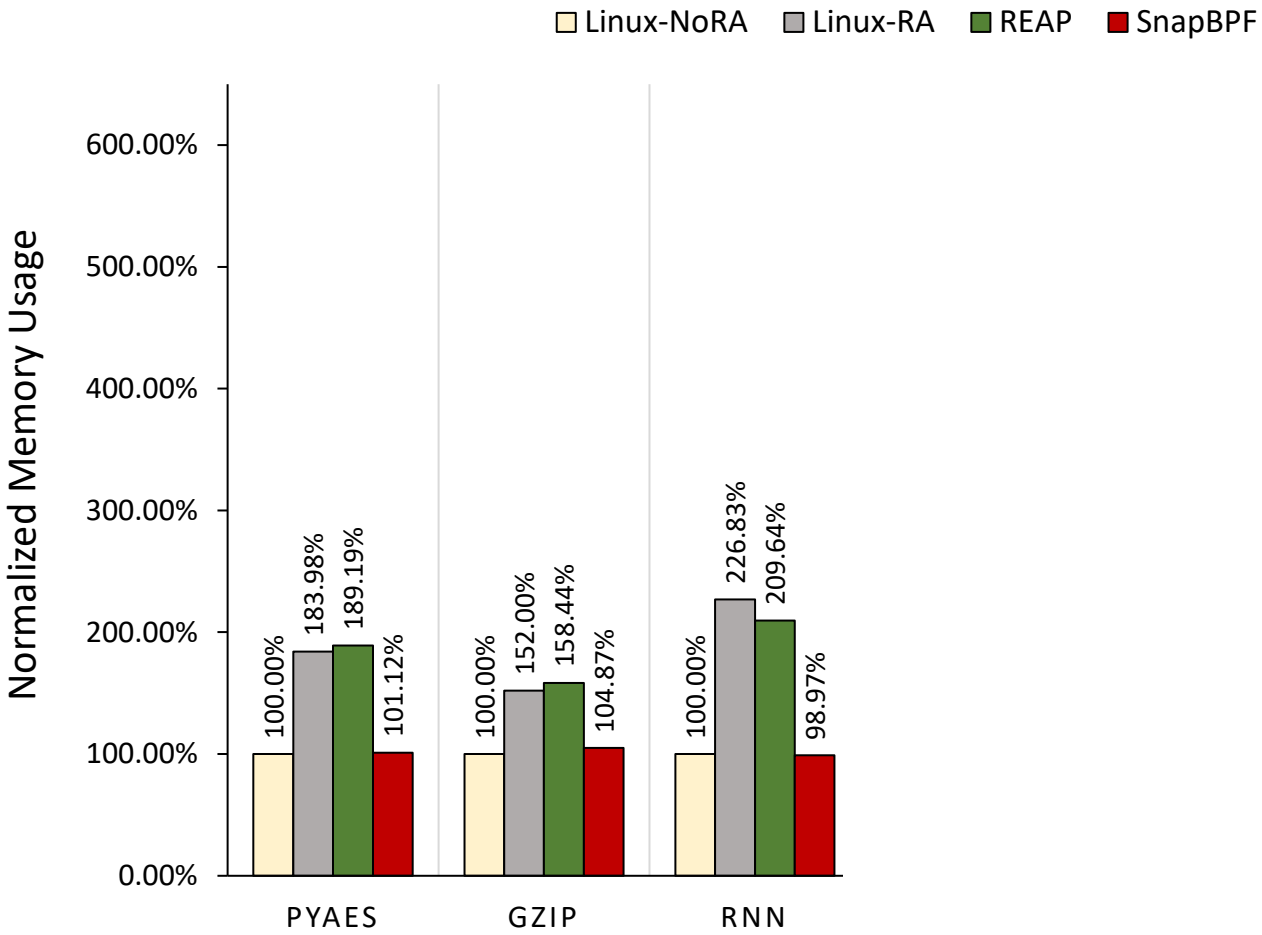
SnapBPF matches and outperforms SOTA on average without separately serialized working sets

Concurrent Execution: Memory

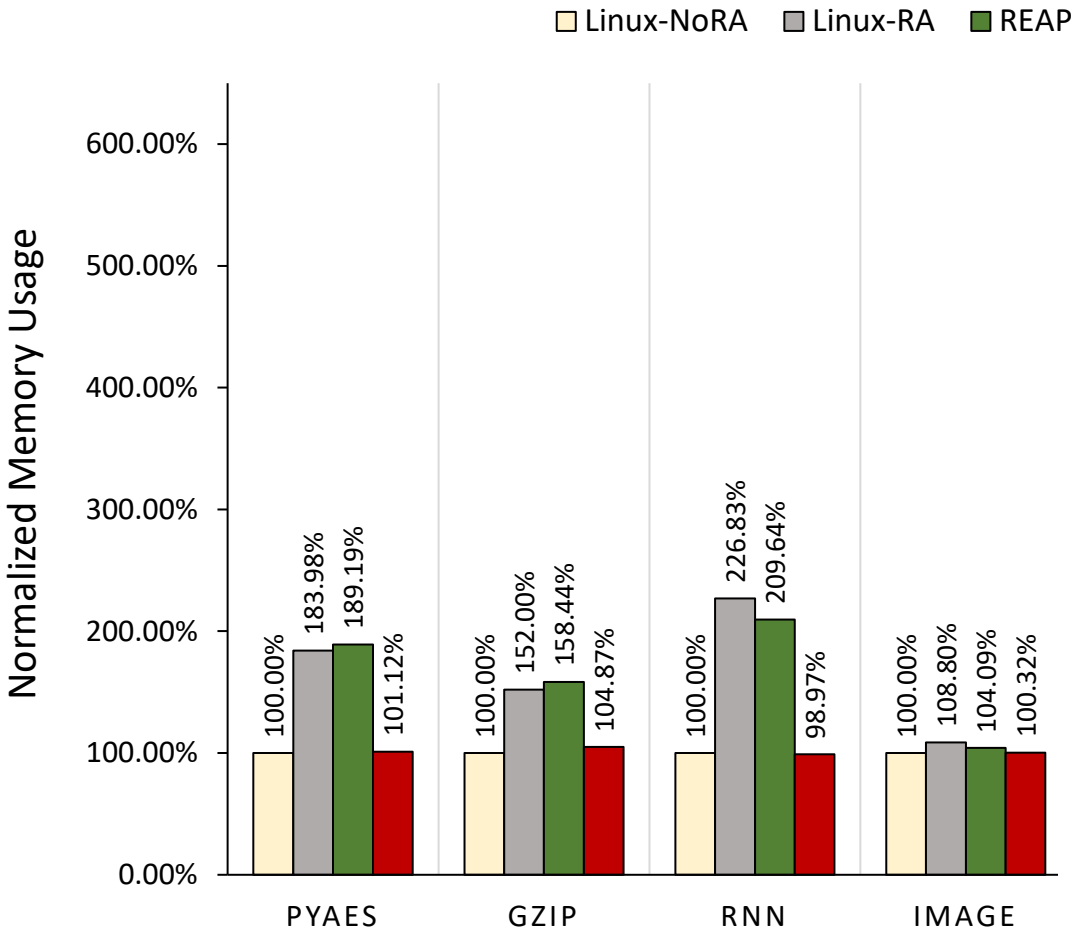
Concurrent Execution: Memory



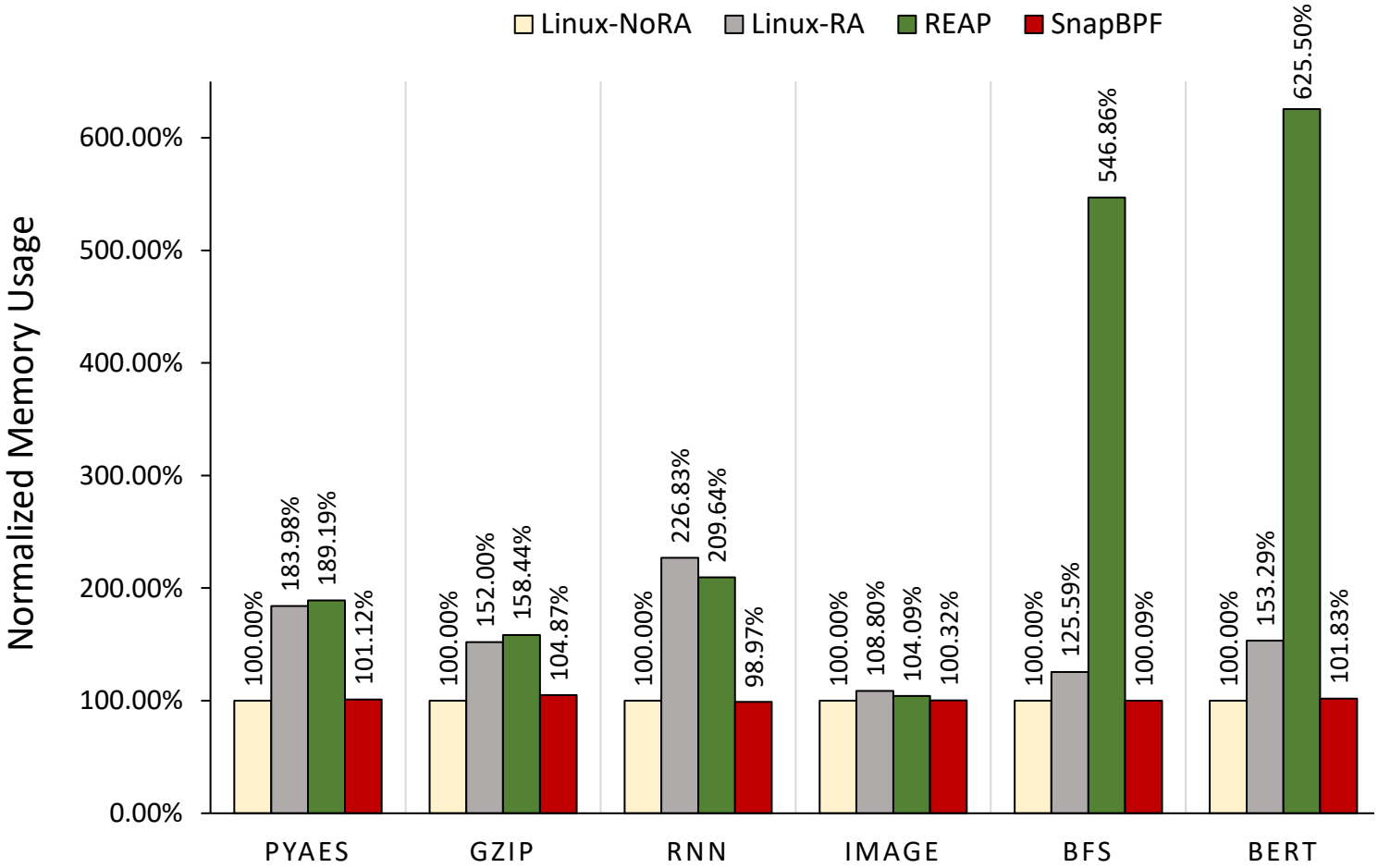
Concurrent Execution: Memory



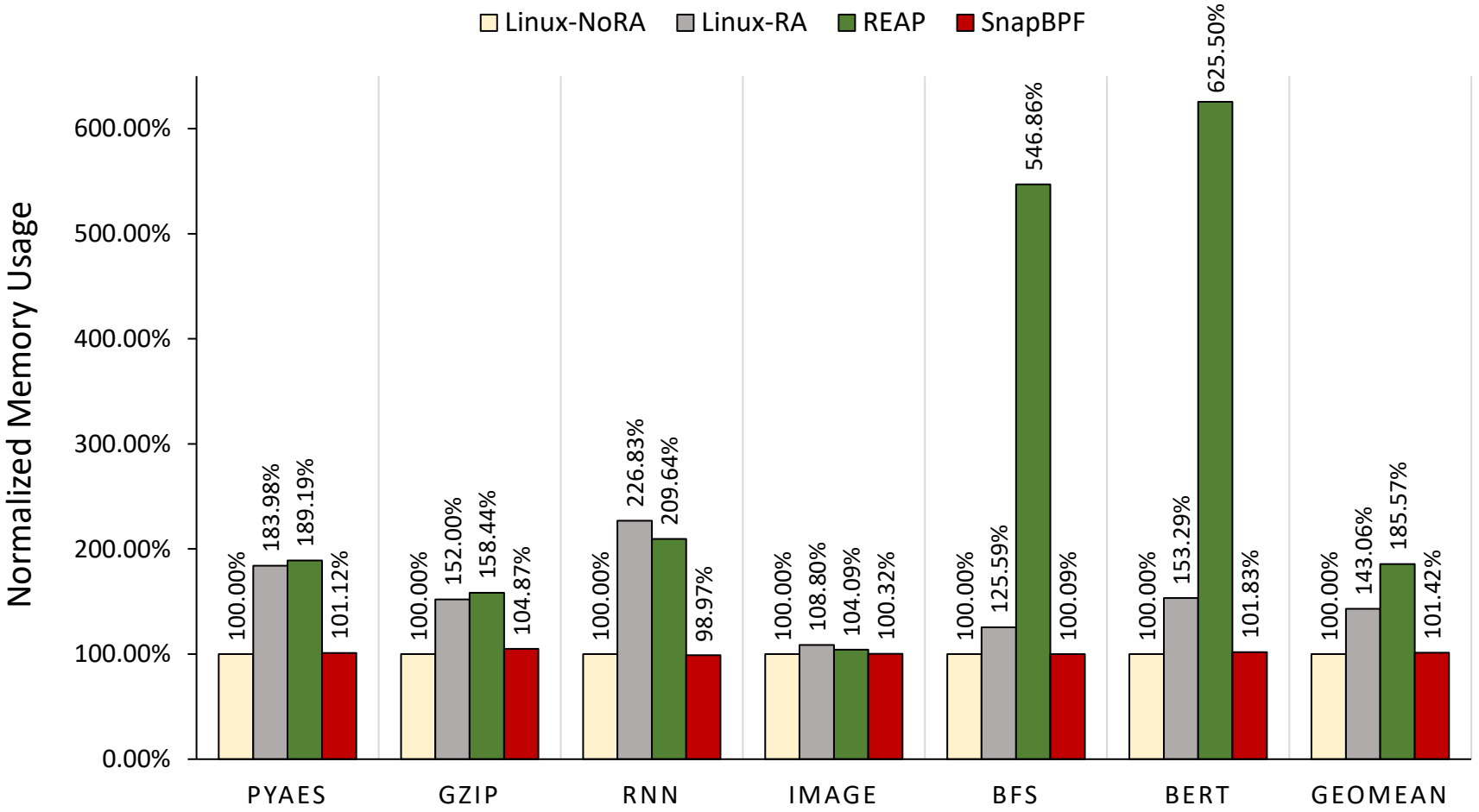
Concurrent Execution: Memory



Concurrent Execution: Memory

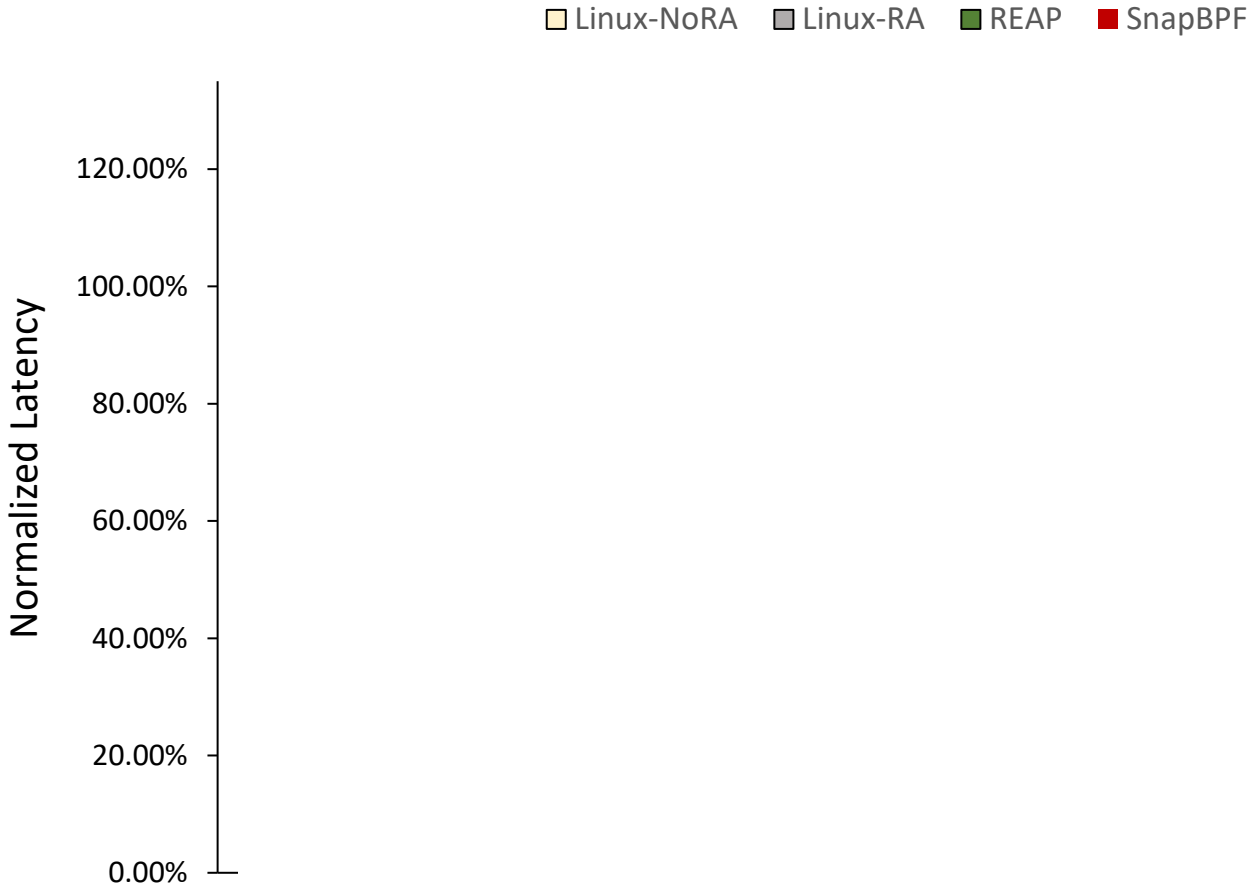


Concurrent Execution: Memory

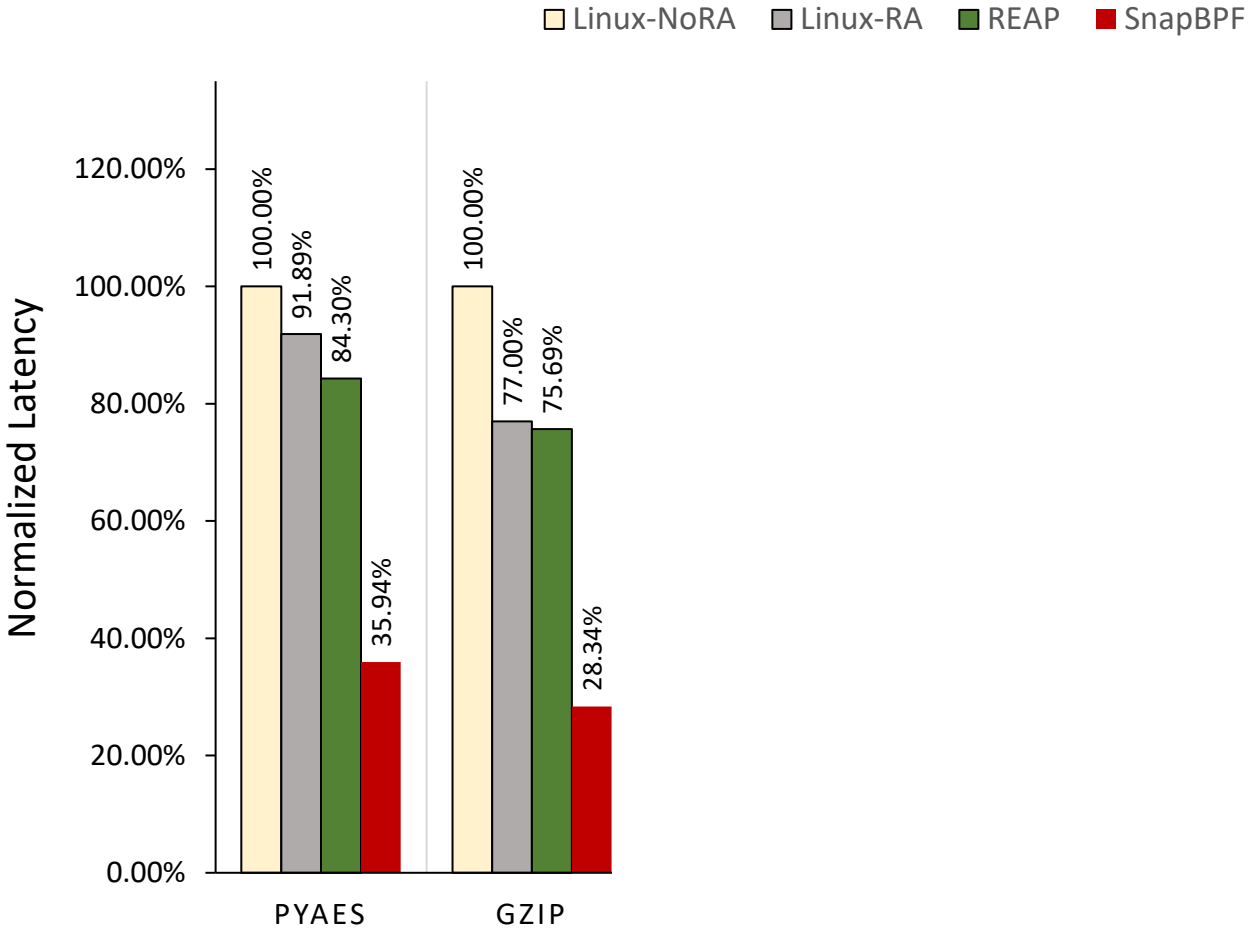


Concurrent Execution: Latency

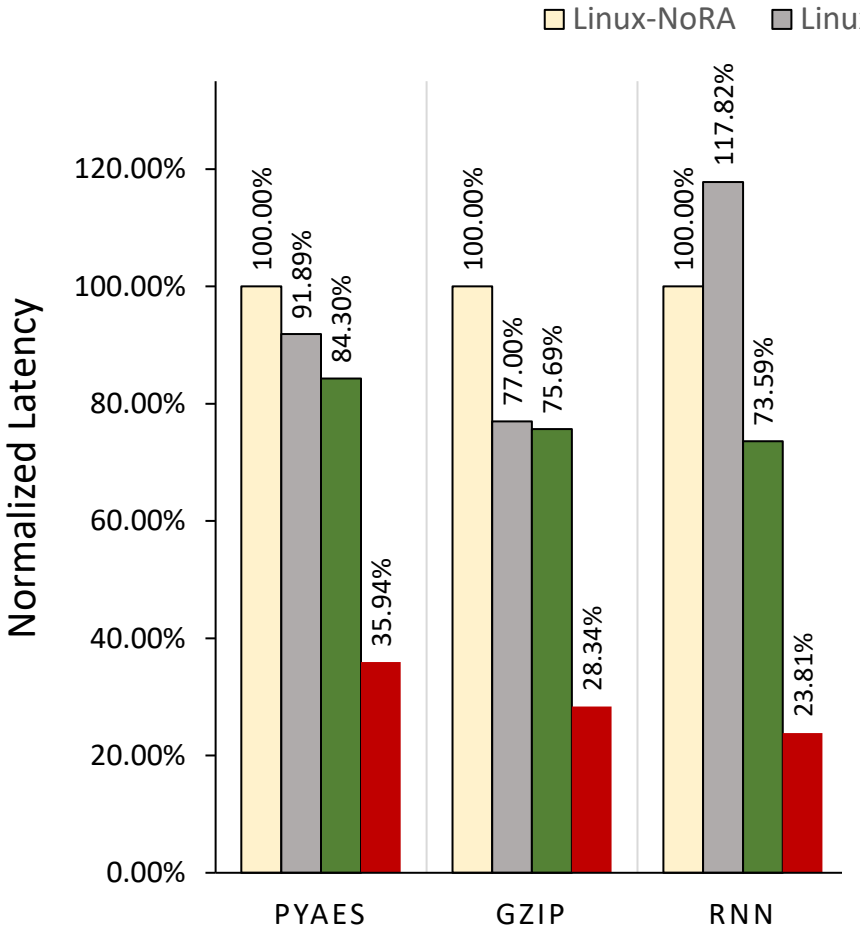
Concurrent Execution: Latency



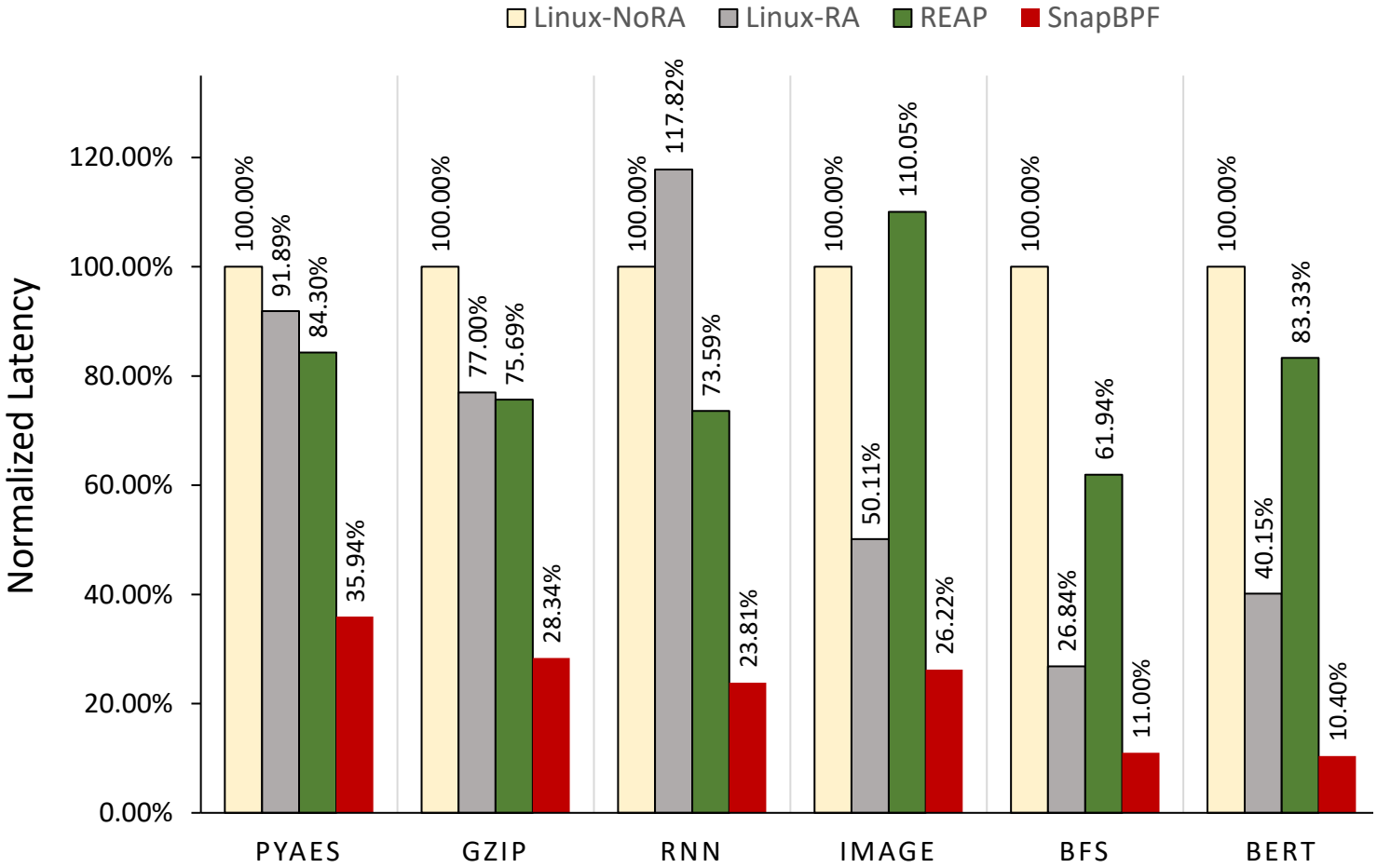
Concurrent Execution: Latency



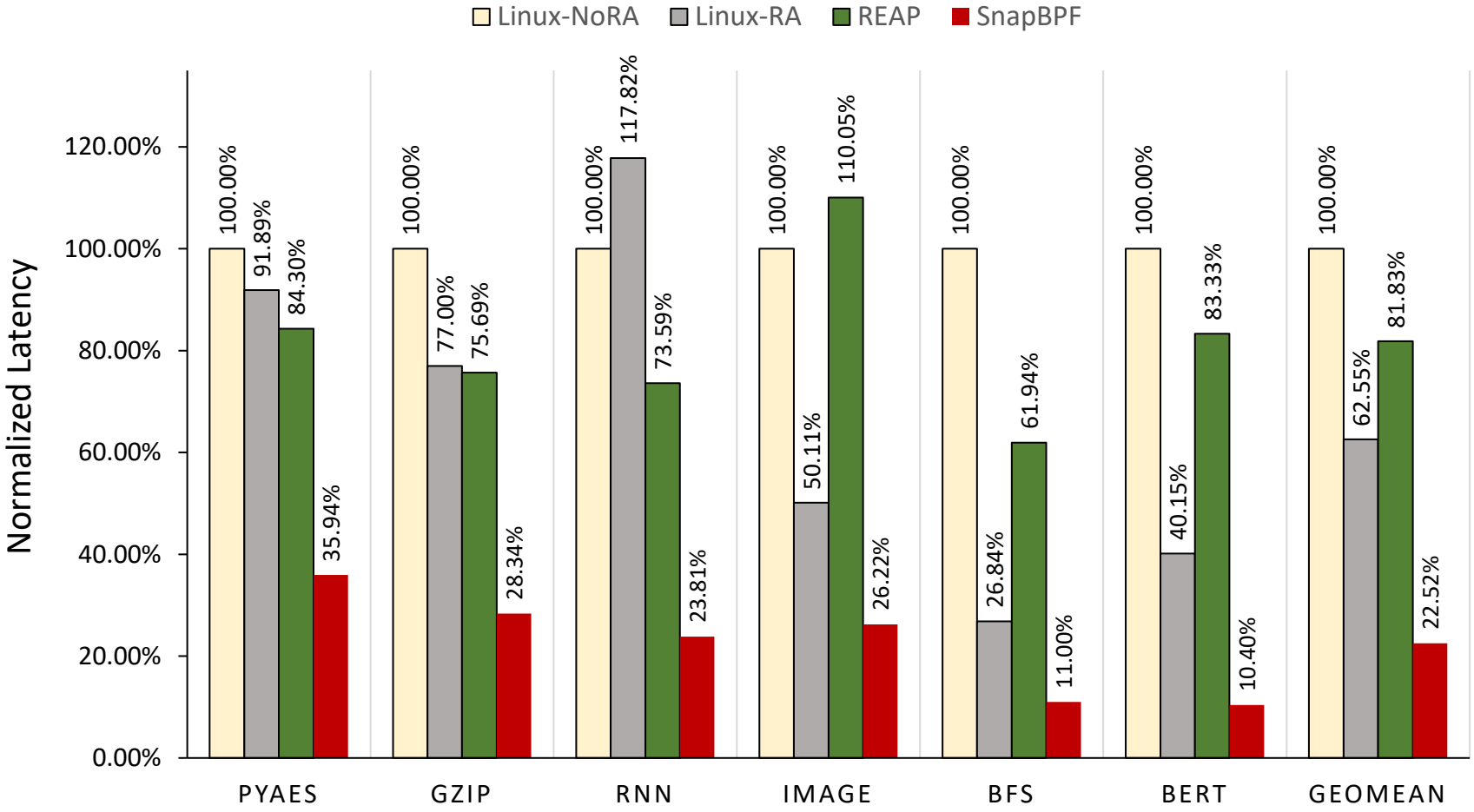
Concurrent Execution: Latency



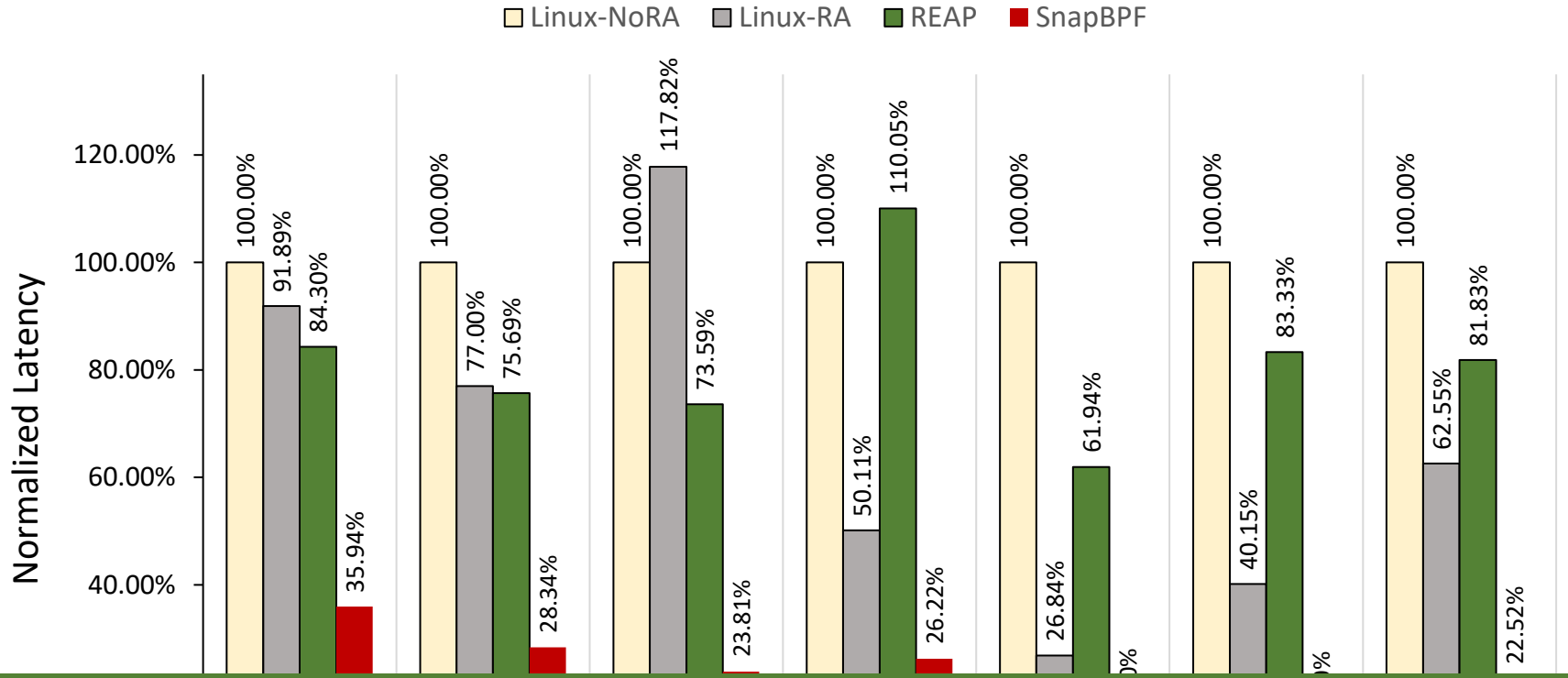
Concurrent Execution: Latency



Concurrent Execution: Latency



Concurrent Execution: Latency



SnapBPF deduplicates memory without sacrificing performance

Outline

→ FaaS Snapshotting

→ Working Set Prefetching

→ SnapBPF

i. eBPF capture and prefetch

ii. PV free page filtering

→ Evaluation

→ Conclusion

Conclusion and next steps

Conclusion and next steps

→ SnapBPF:

→ Minimizes complexity without sacrificing performance

Conclusion and next steps

→SnapBPF:

→ Minimizes complexity without sacrificing performance

→Key enablers:

Conclusion and next steps

→ SnapBPF:

→ Minimizes complexity without sacrificing performance

→ Key enablers:

→ eBPF: Move capture and prefetch in kernelspace

Conclusion and next steps

→ SnapBPF:

→ Minimizes complexity without sacrificing performance

→ Key enablers:

→ eBPF: Move capture and prefetch in kernelspace

→ PTE marking: Bridge the virtualization semantic gap

Conclusion and next steps

Conclusion and next steps

→ Device Performance: Sequential vs Random I/O

Conclusion and next steps

→Device Performance: Sequential vs Random I/O

→Memory contiguity and large folios

Conclusion and next steps

→Device Performance: Sequential vs Random I/O

→Memory contiguity and large folios

→Overhead of PV PTE marking

Conclusion and next steps

→Device Performance: Sequential vs Random I/O

→Memory contiguity and large folios

→Overhead of PV PTE marking

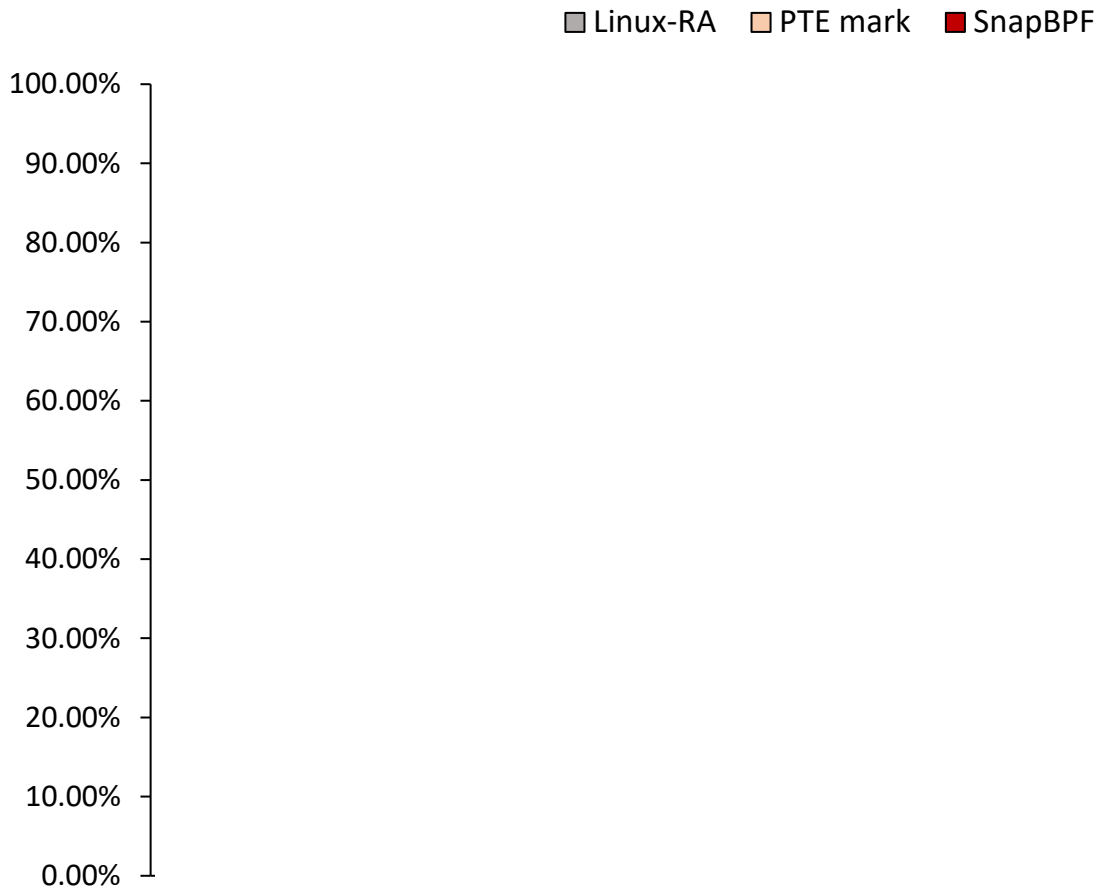
→Userfaultfd features (remote fetch)

Thank you!

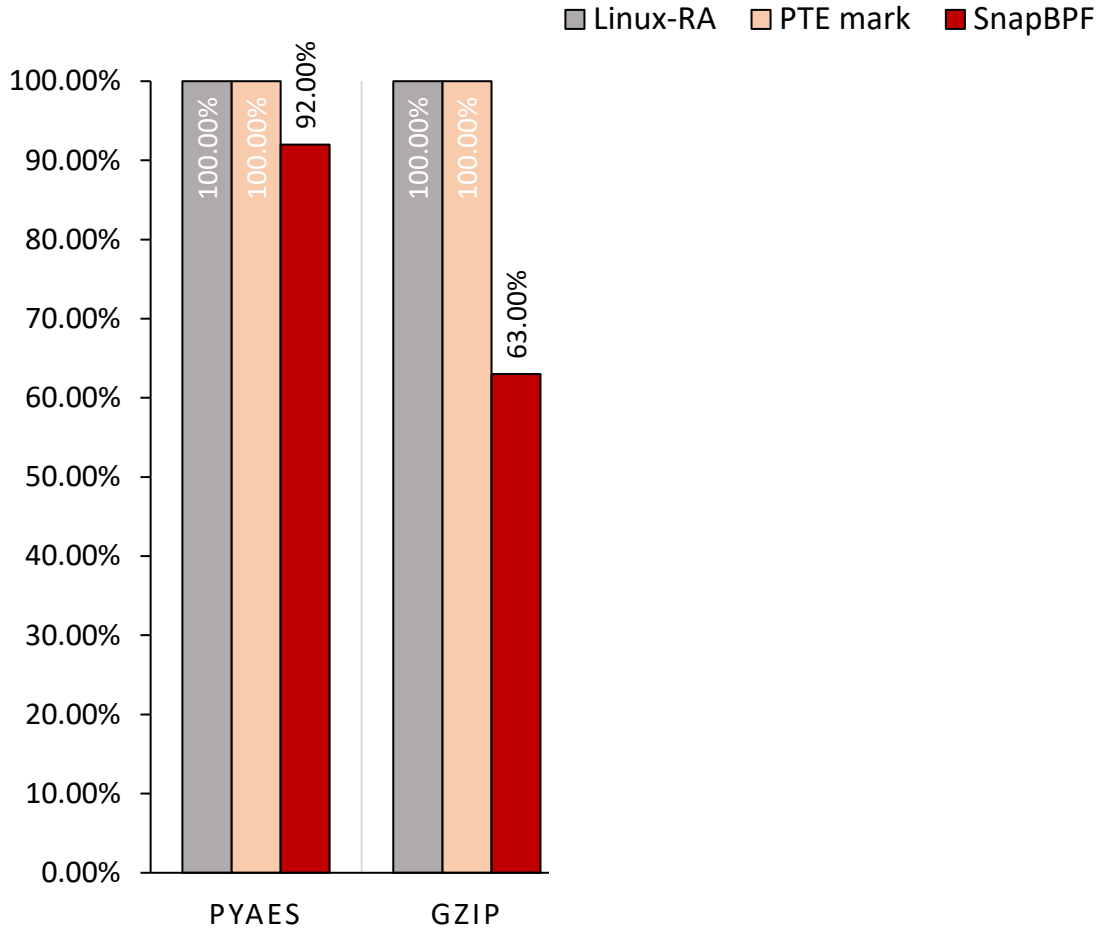
Thank you!
Questions?

SnapBPF performance breakdown

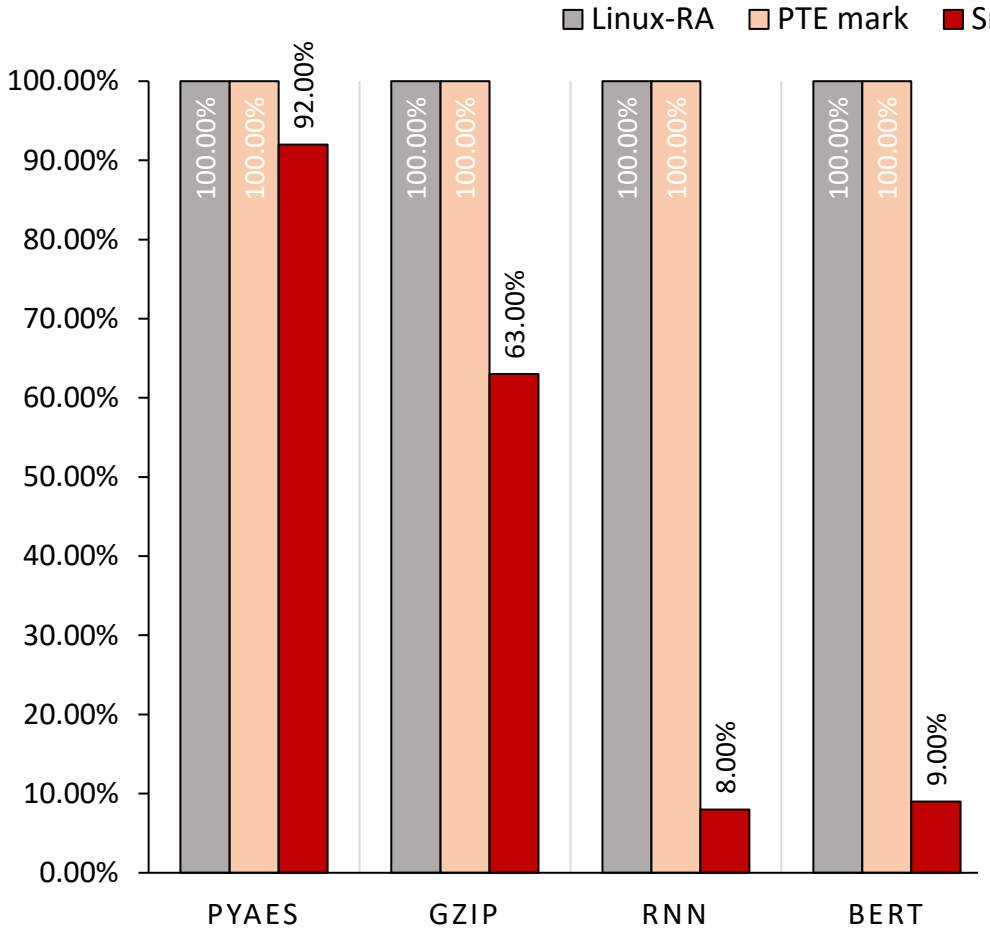
SnapBPF performance breakdown



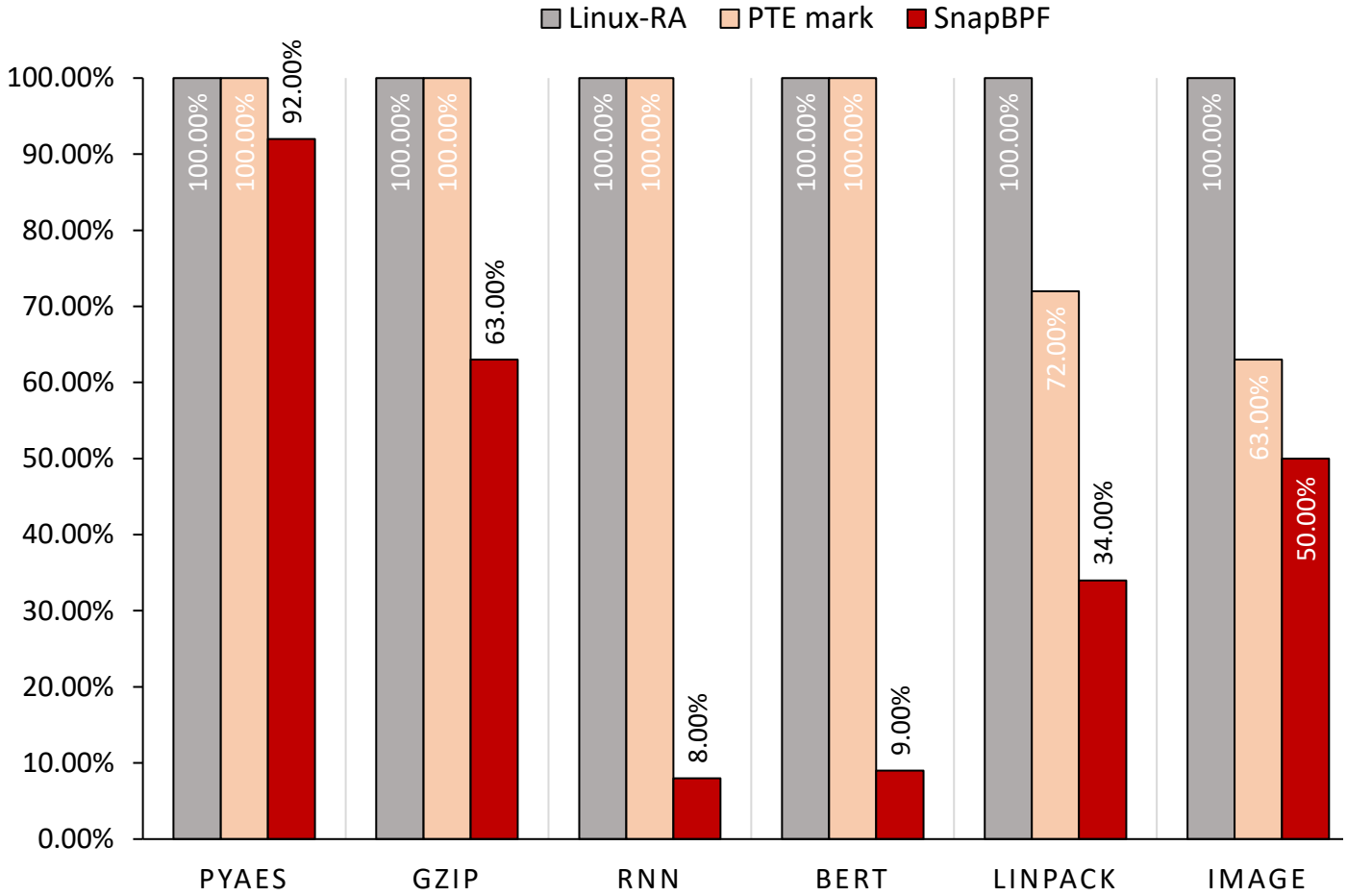
SnapBPF performance breakdown



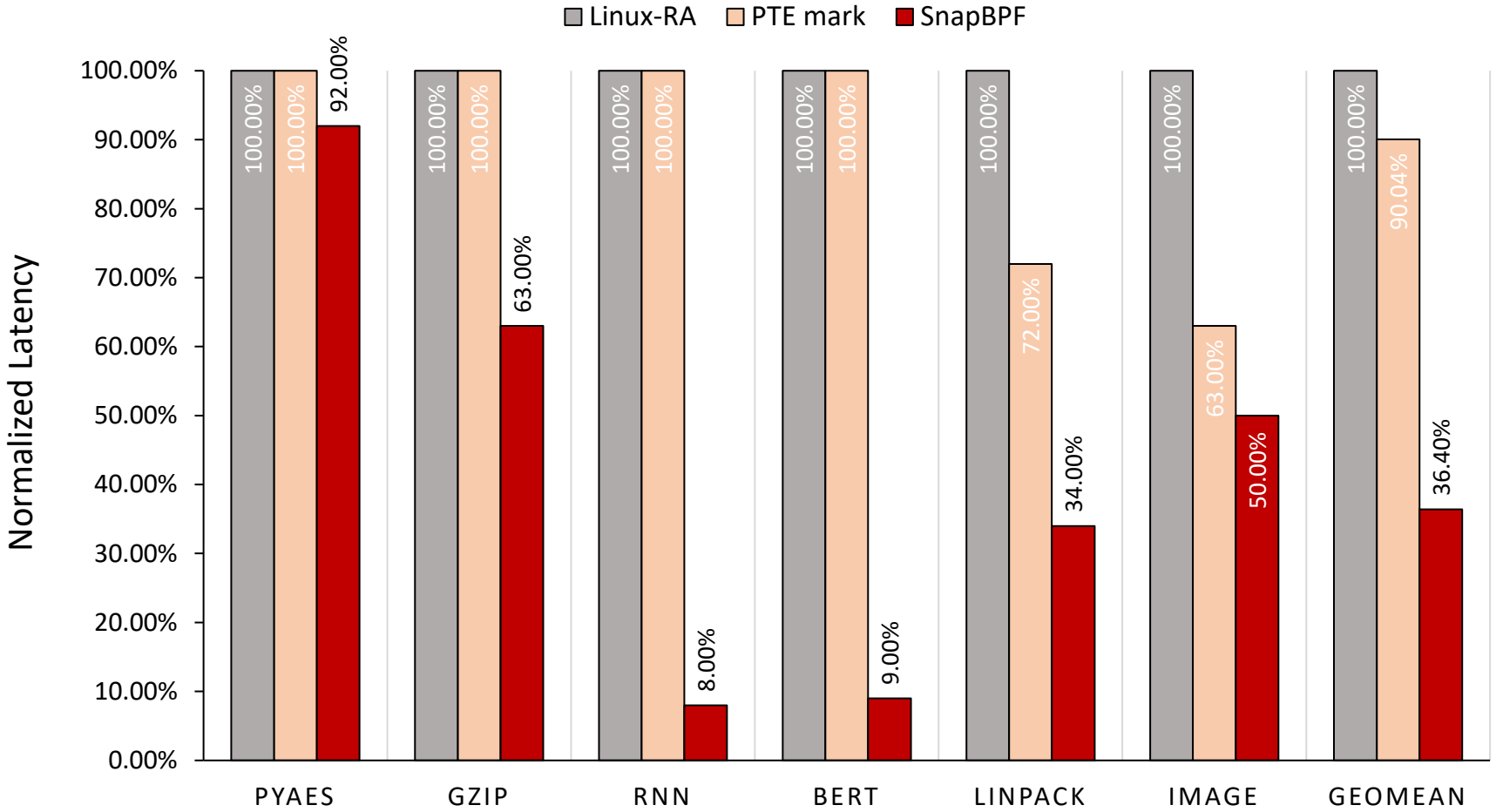
SnapBPF performance breakdown



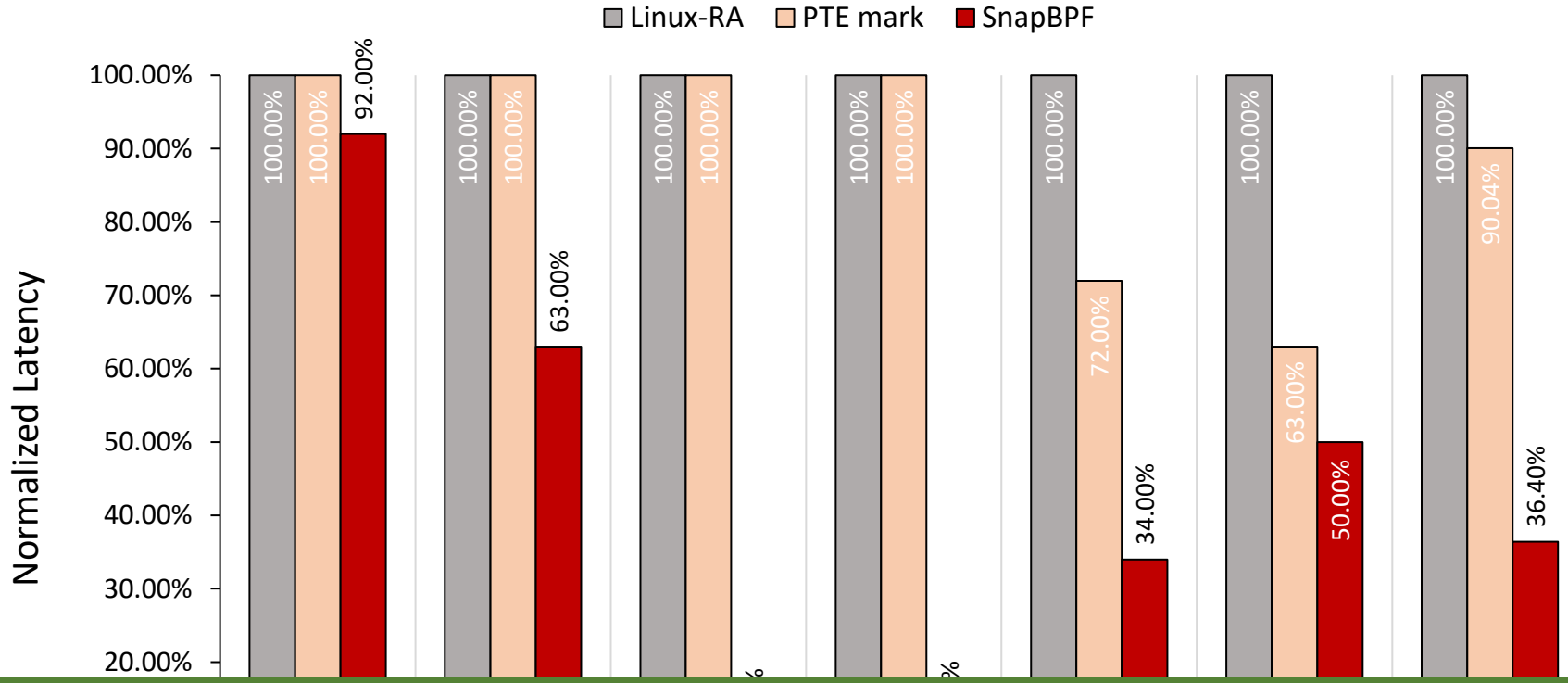
SnapBPF performance breakdown



SnapBPF performance breakdown



SnapBPF performance breakdown



Online free page filtering boosts performance for allocation-heavy functions