SPP

Safe Persistent Pointers for Memory Safety

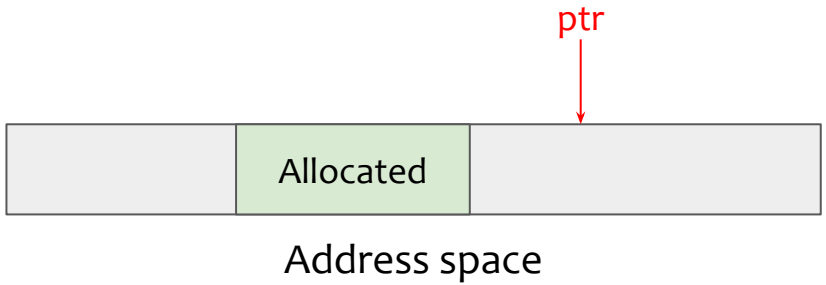
Dimitrios Stavrakakis, Alexandra Panfill
MJin Nam, Pramod Bhatotia

IEEE DSN 2024

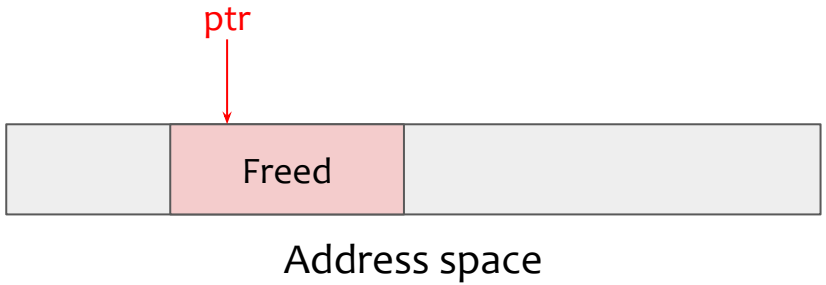
Memory safety

Memory safety violations : Illegal accesses to unintended memory regions

Spatial memory safety
e.g., buffer overflow, stack overflow



Temporal memory safety
e.g., dangling pointer, double free



Memory safety in practice

Prevalent in almost all low-level unsafe C/C++ code

Chromium project [1]

- 70% of vulnerabilities are memory safety problems

Microsoft [2]

- 70% of vulnerabilities fixed in security patches are memory safety violations

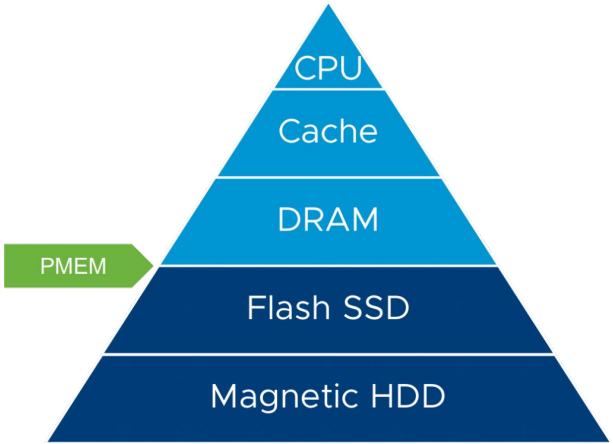
Android [3]

- 75% of vulnerabilities are memory safety issues

[1] Chromium project: https://www.chromium.org/Home/chromium-security/memory-safety
[2] Microsoft: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/
[3] Android: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html

Persistent memory (PM)

Persistent memory management is susceptible to memory safety vulnerabilities



- Persistent memory programming model
- Durability & crash consistency
- Recovery code paths
- Performance overheads

Memory safety approaches for PM are **non-practical** for production deployment

SPP: Safe Persistent Pointers for Memory Safety

Memory safety mechanism for PM-based applications

System properties:

- Spatial memory safety
- Transparency
- High coverage
- Crash consistency

Performance

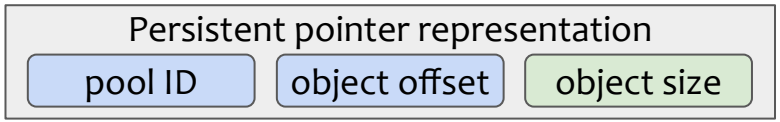
Outline

- ~~Motivation~~
- Design
  - Overview
  - Persistent memory operations
- Implementation
- Evaluation

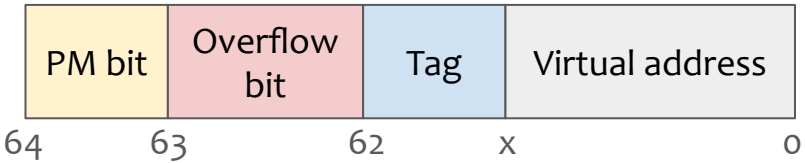
Design overview

SPP enforces a tagged pointer-based approach for memory safety

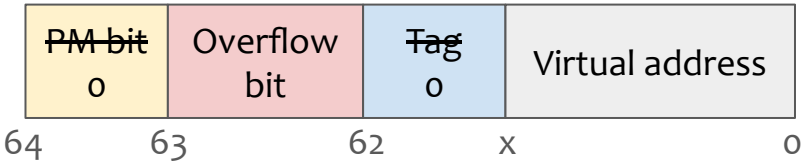
- Enhanced PM pointer representation



- Native tagged pointer scheme



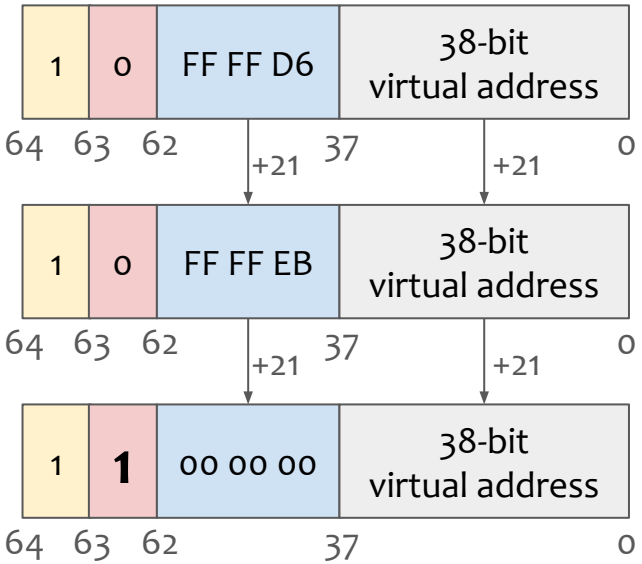
- Runtime checks



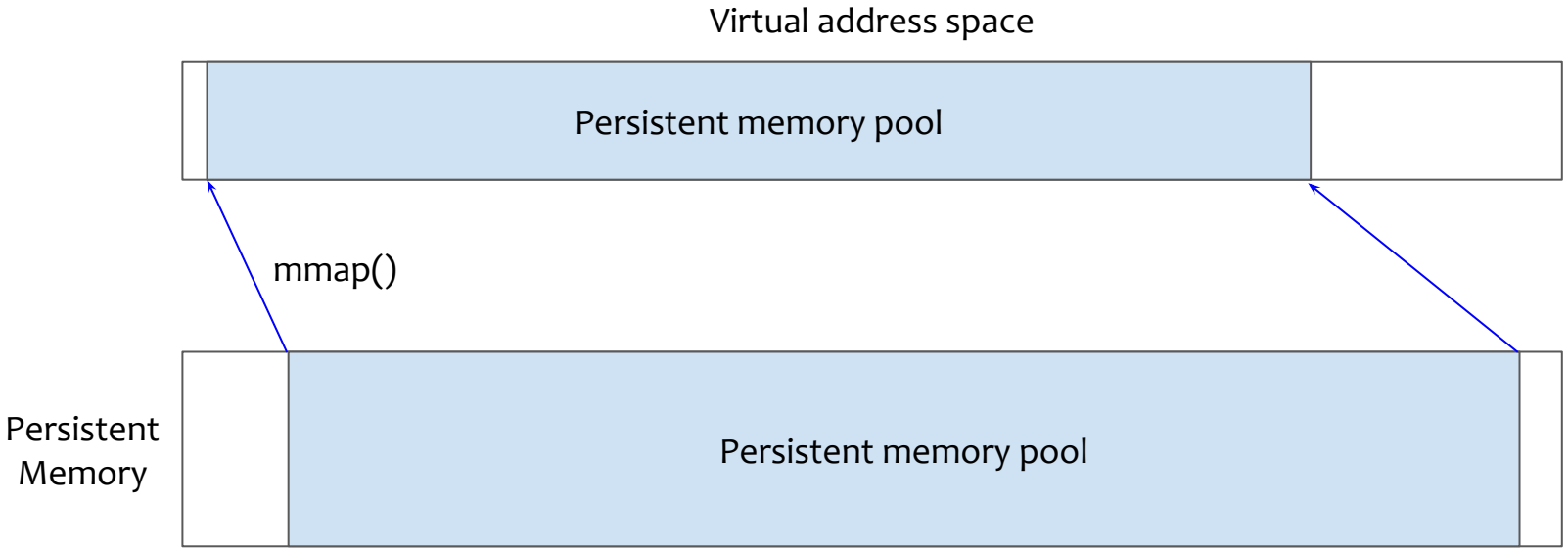
Design overview - An example

SPP sets and updates the native pointer **tag** on pointer operations

- pm_ptr : pointer to a 42-bytes PM object
- pm_ptr += 21; // ptr is **in** bounds
- pm_ptr += 21; // ptr gets **out** of bounds

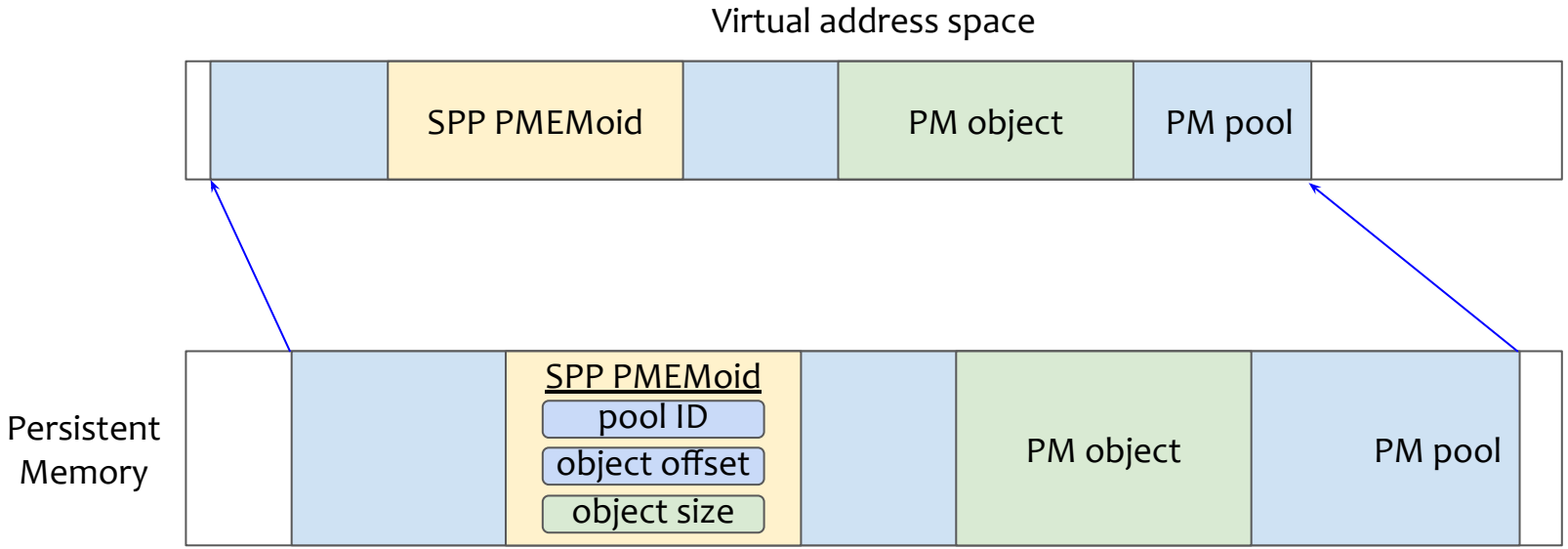


Design overview



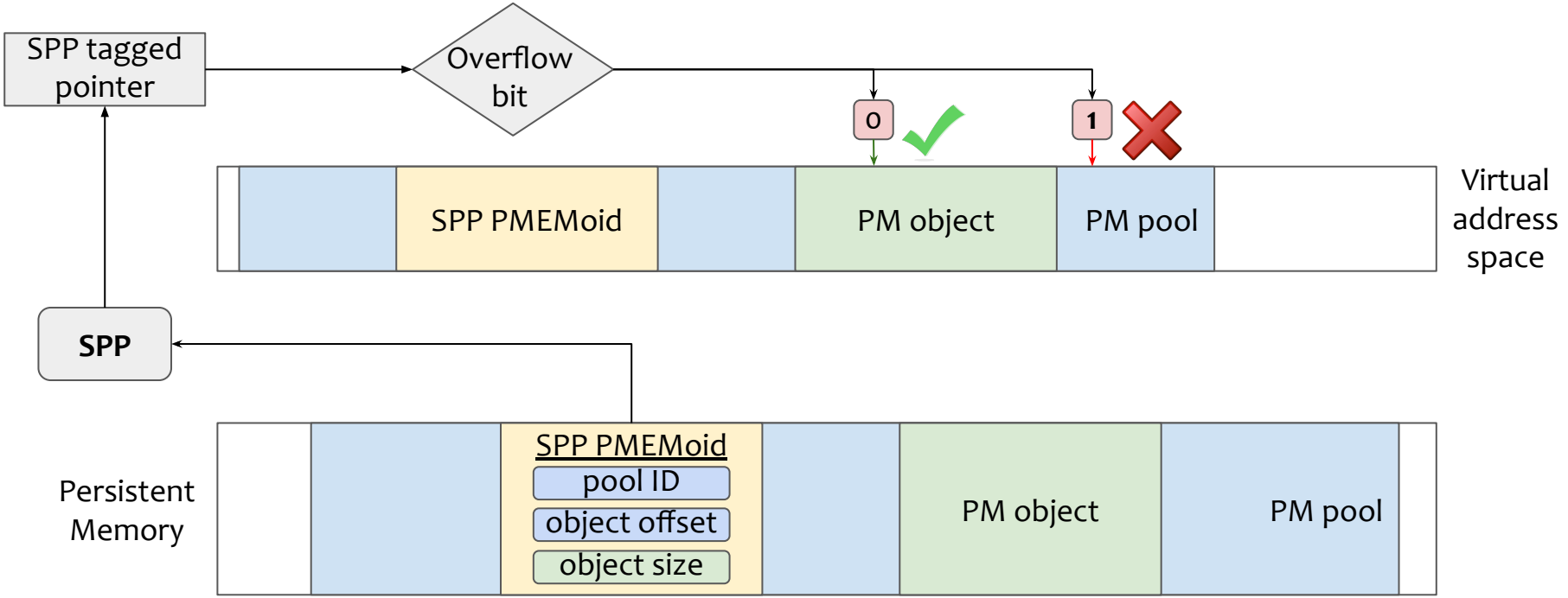
Persistent memory pools are directly mapped to the virtual address space of an application

Design overview - PM allocation



SPP allocates the object and atomically sets the **object size** field

Design overview - PM access



On a memory access SPP preserves the **overflow bit** and performs an implicit bounds check

Outline

- ~~Motivation~~
- ~~Design~~
- Implementation
- Evaluation

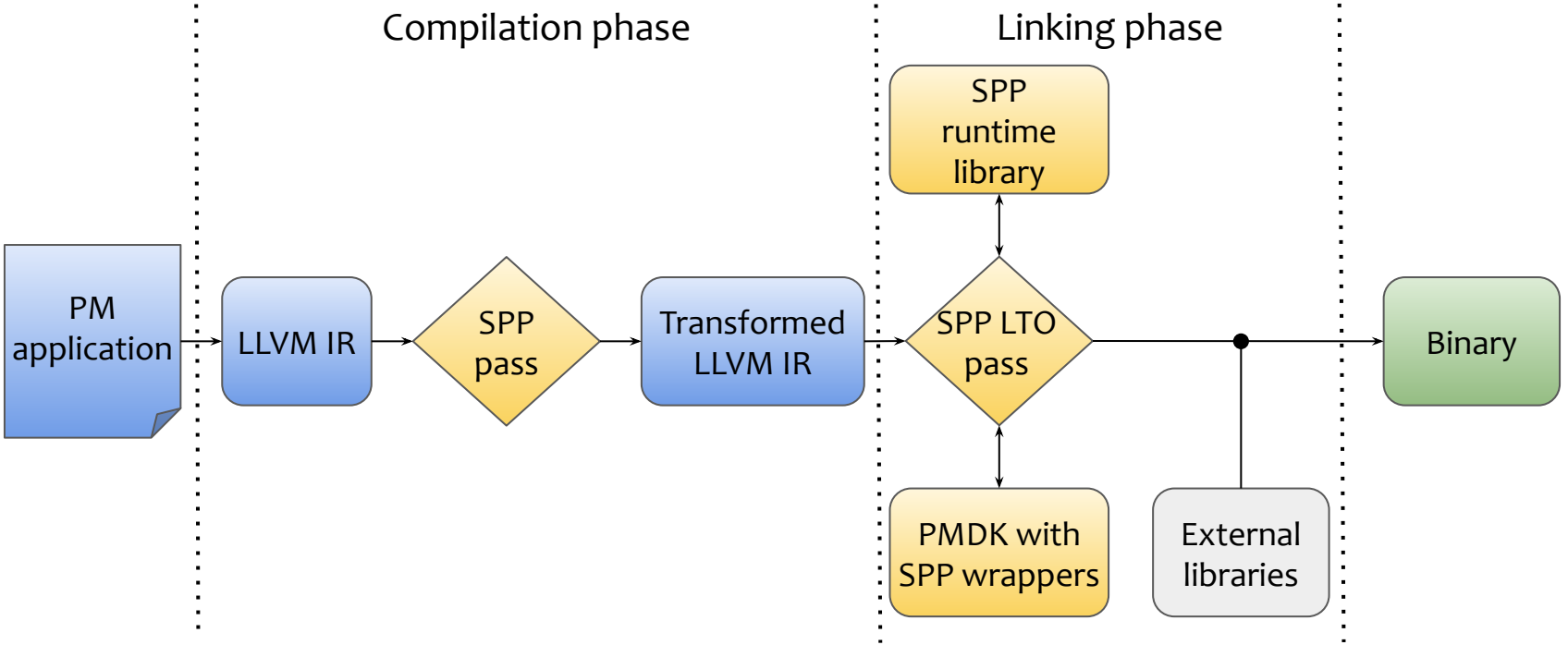
Implementation

SPP is built on PMDK[1] and LLVM[2]

- SPP pointer representation – minimization of space overheads & runtime checks!
- Transformation & LTO compiler passes – performance & compatibility!
- PMDK programming model – transparent support!
- Crash consistency via PMDK transactions & atomic operations

[1]Persistent memory development kit (PMDK): https://github.com/pmem/pmdk
[2]LLVM: https://github.com/llvm/llvm-project

SPP hardening workflow



The application is finally linked with SPP runtime library, PMDK and external libraries

Outline

- ~~Motivation~~
- ~~Design~~
- ~~Implementation~~
- Evaluation

Evaluation

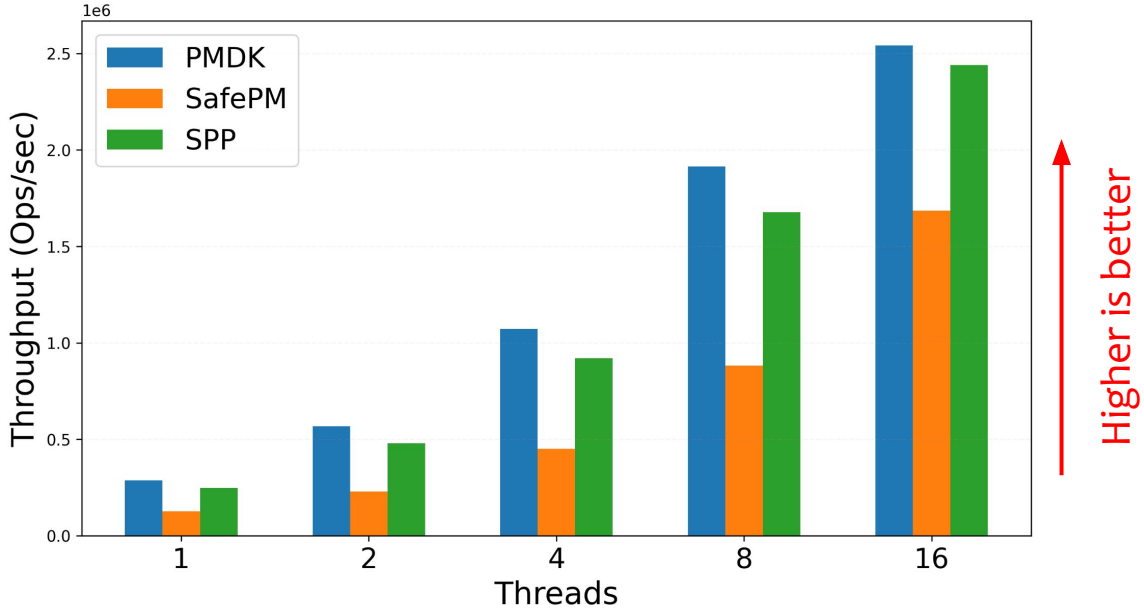
- What is the performance overhead of SPP?
  - Persistent memory KV store (pmemkv)
- How much PM space overhead does SPP introduce?
  - Persistent indices (ctree, rtree, rbtree, hashmap)
- How robust is SPP in detecting memory safety vulnerabilities?
  - RIPE benchmark framework

Evaluation

- Experimental setup:
  - Dual socket Intel Xeon Gold 6326 CPU (16 cores)
  - 64 GB DRAM / socket
  - 1 TB Intel Optane DC DIMMs / socket
  - PM configured in *App-Direct* mode
- Variants:
  - *PMDK*→No memory safety
  - *SafePM*→Application hardened with SafePM
  - *SPP*→ Application hardened with SPP

Performance overhead

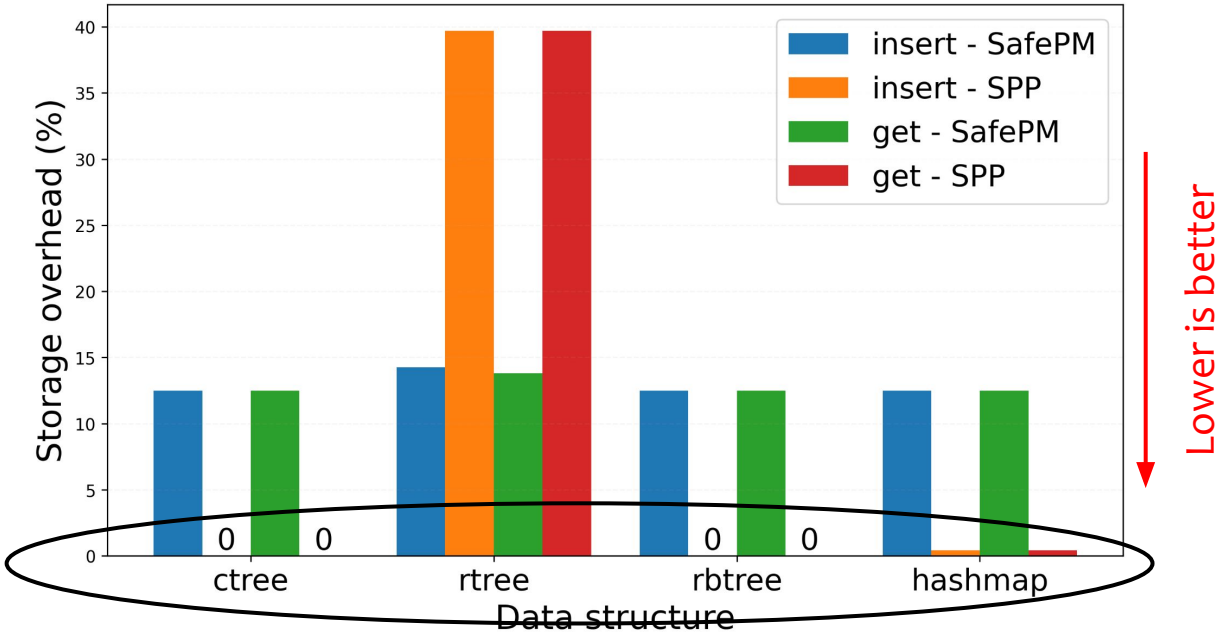
Persistent KV-store benchmark, **10M** ops, **50%** reads / **50%** writes



SPP incurs notably lower performance overheads compared to SafePM

Space overhead

Persistent indices, insert/get workloads, relative to PMDK



SPP does not introduce significant PM space overhead on average

Efficiency

RIPE benchmark, **223** buffer overflow exploits

Variant	**Exploitable PM buffer overflows**
PMDK	**83**
memcheck	**20**
SafePM	**6**
SPP	**4**

SPP is an efficient memory safety solution for PM with low performance overheads

Summary

Current PM memory safety approaches are designed for debugging purposes

- high performance overheads
- considerable PM storage overheads

Safe Persistent Pointers (SPP):

- comprehensive spatial memory safety
- low performance & PM storage overheads
- no source code modifications
- crash consistency & durability

Try it out!

https://github.com/dimstav23/SPP

Sources

[1] PM hierarchy image, https://www.starwindsoftware.com/blog/persistent-memory-in-vmware-vsphere-6-7-what-is-it-how-fast-is-it

Backup

Tackling memory safety issues

Deterministic dynamic bounds checking

- Augment unmodified applications with memory safety metadata
- Perform runtime checks on memory accesses

Software-based approaches

- Compiler transformations
- Runtime libraries
- “High” runtime overhead
- No special hardware requirement

Hardware-based approaches

- Hardware support via special registers or ISA modifications
- “Low” runtime overhead
- Special hardware requirements

PM memory safety challenges

Memory safety approaches for volatile memory are insufficient for PM

- Persistent memory programming model
  - Introduction of persistent pointer representation
- Dedicated persistent memory allocators
  - Maintain durable and valid heap metadata & objects
- Memory safety on application recovery paths
  - Require consistent memory safety metadata across application runs

How do we address these challenges?

Why memory safety is a such a major issue?

Security

Performance

- Direct access to the memory regions, leading to safety violations
- High-level languages, e.g., “Java, what a horrible horrible language!” -- Torvalds
- Notable exception is Rust

- Low-level languages (C/C++)
- A mandatory requirement for all critical/foundational software systems (OS, Databases, storage/filesystems, browser, network, crypto)
- Legacy softwares