SPP

Safe Persistent Pointers for Memory Safety

Dimitrios Stavrakakis, Alexandra Panfill
MJin Nam, Pramod Bhatotia

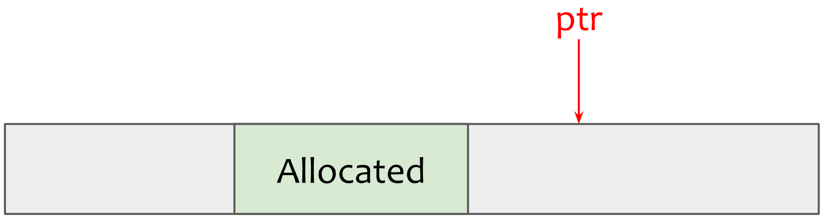
Technische Universität München
THE UNIVERSITY of EDINBURGH

IEEE/IFIP DSN 2024

Memory safety

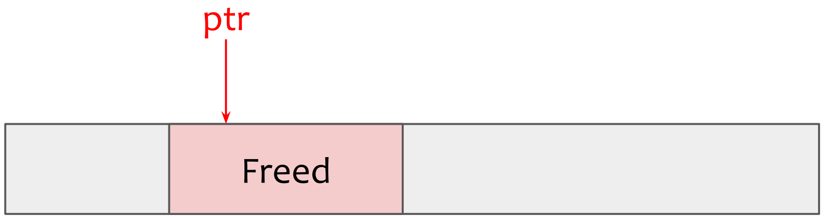
Memory safety violations : Illegal accesses to unintended memory regions

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



Address space

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



Address space

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

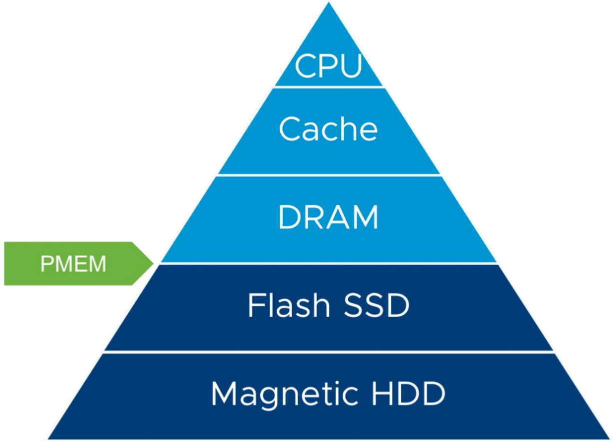
Android [3]

- 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) safety challenges

Persistent memory management is susceptible to memory safety vulnerabilities



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

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

Problem statement

How to design a **practical memory safety solution** for PM applications with minimal performance overheads?

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
  - Challenges
  - Example
  - Persistent memory operations
- Implementation
- Evaluation

Design overview

SPP enforces a tagged pointer-based approach for memory safety

- Enhanced PM pointer representation

Persistent pointer representation		
pool ID	object offset	object size

- Native tagged pointer scheme

PM bit	Overflow bit	Tag	Virtual address

64 63 62 x 0

- Implicit runtime checks

~~PM bit~~ 0	Overflow bit	~~Tag~~ 0	Virtual address

64 63 62 x 0

Design challenges

#1 Performance & PM storage overheads

#2 Transparent integration in existing toolchains

#3 Crash consistency for PM safety metadata

#4 Compatibility with existing applications and libraries

Challenge #1: Performance & PM storage overheads

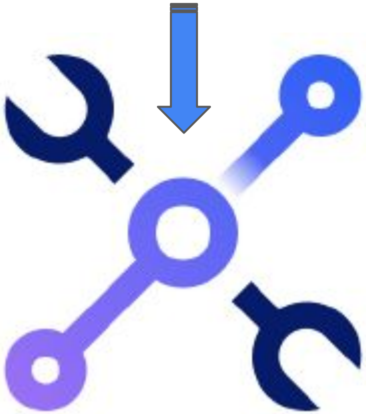
- PM pointer representation ⇨ 64 bits of PM safety metadata per object
- Pointer tracking ⇨ omit checks for pointers to volatile memory
- Bound checks preemption ⇨ reduce the inserted runtime operations

Minimal PM safety metadata & optimized runtime instrumentation

Challenge #2: Integration in existing toolchains

- Transparent creation of the tagged pointers via the PM API
- Wrappers for PM management operations
- Wrappers for memory intrinsic functions

SPP



SPP supports the PM API and memory intrinsic functions without modifications

Challenge #3: Crash consistency for PM safety metadata

- PM guarantees atomicity only for aligned 8-byte stores

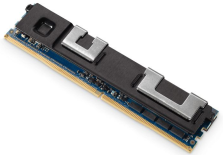
- Atomic operations & software transactions

- Include PM safety metadata in transaction logs

Crash

Recovery

Consistent
Meta(data)



Crash-consistent (meta)data updates using atomic operations and transactions

Challenge #4: Compatibility with applications/libraries

- Pre-compiled shared libraries/applications

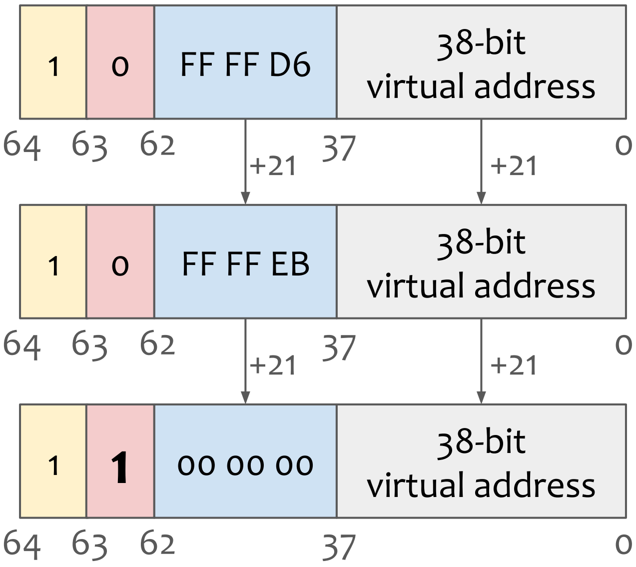
```
void* tagged_pm_ptr = pmemobj_direct(obj_id); //get tagged ptr
...
internal_foo(tagged_pm_ptr); //internal function call
...
external_foo(tagged_pm_ptr); //external function call
...
clean_pm_ptr = __spp_cleantag(tagged_pm_ptr); //tag masking
external_foo(clean_pm_ptr); //external function call
...
```

SPP masks the tagged pointers passed to external shared libraries/applications

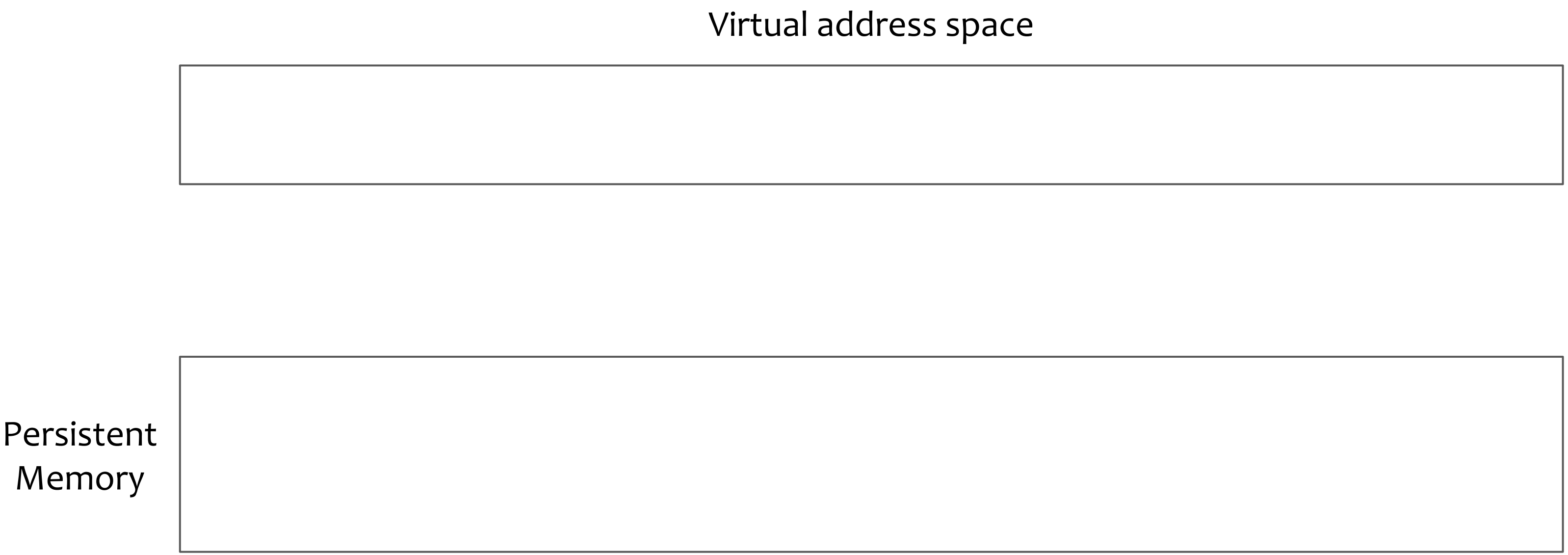
Design overview - An example

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

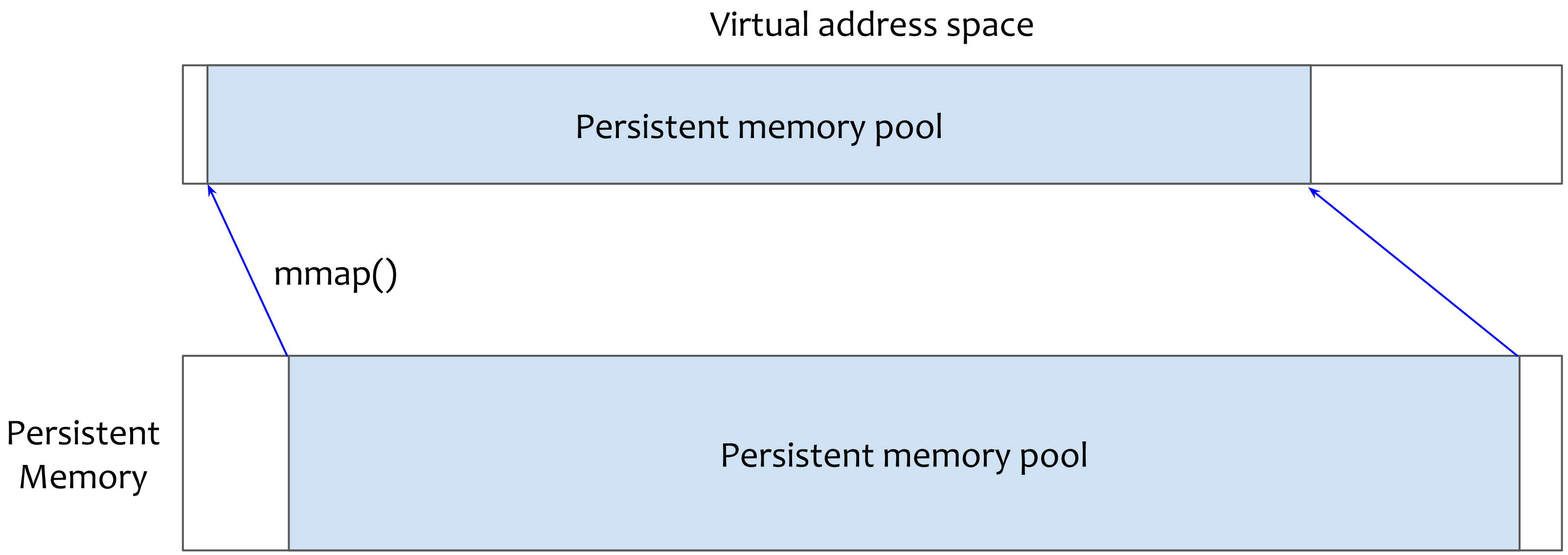
- pm_ptr : pointer to a 42-bytes PM object
- pm_ptr += 21; // ptr is **in** bounds
- pm_ptr += 21; // ptr gets **out** of bounds



Design overview - PM layout

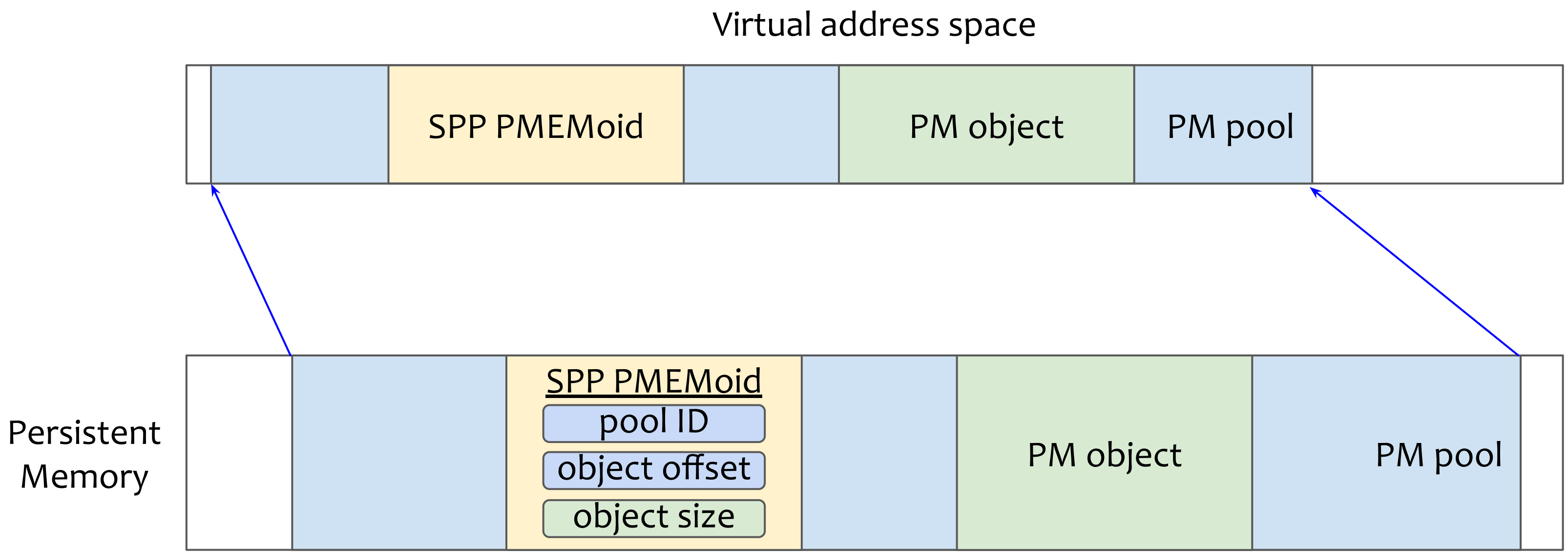


Design overview - PM layout



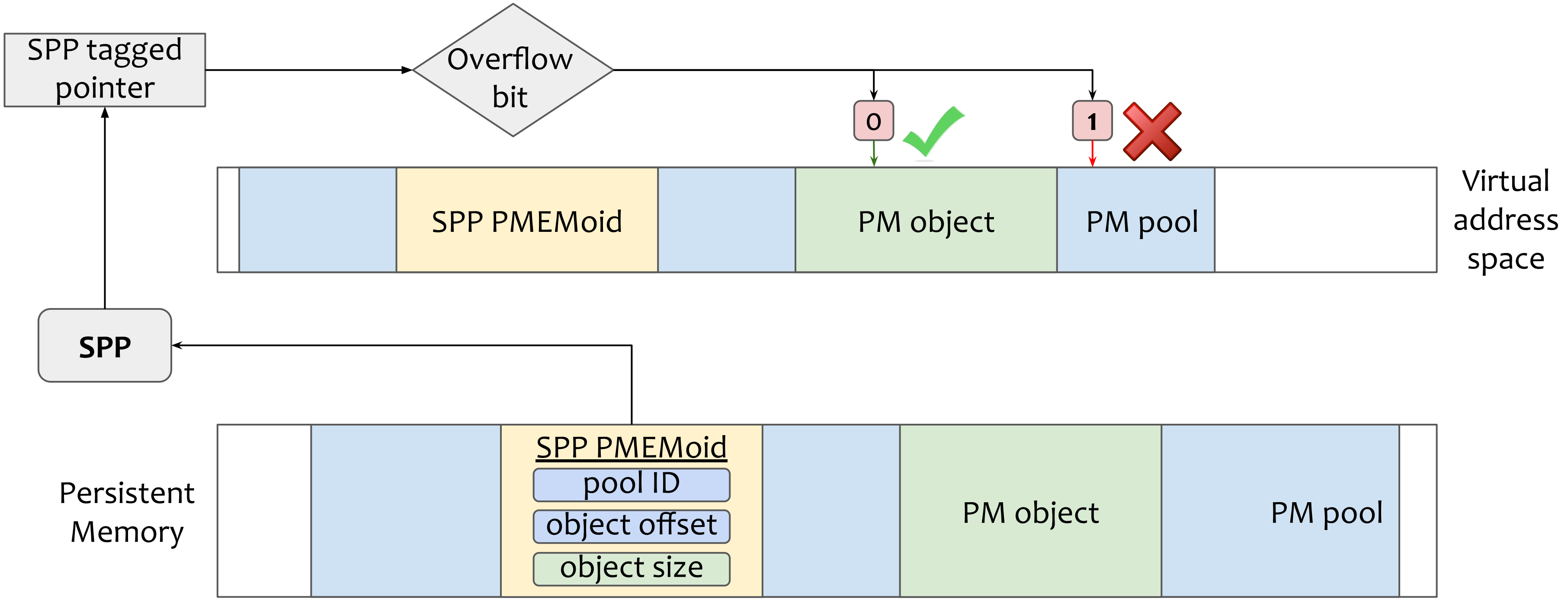
Persistent memory pools are directly mapped to the virtual address space of an application

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

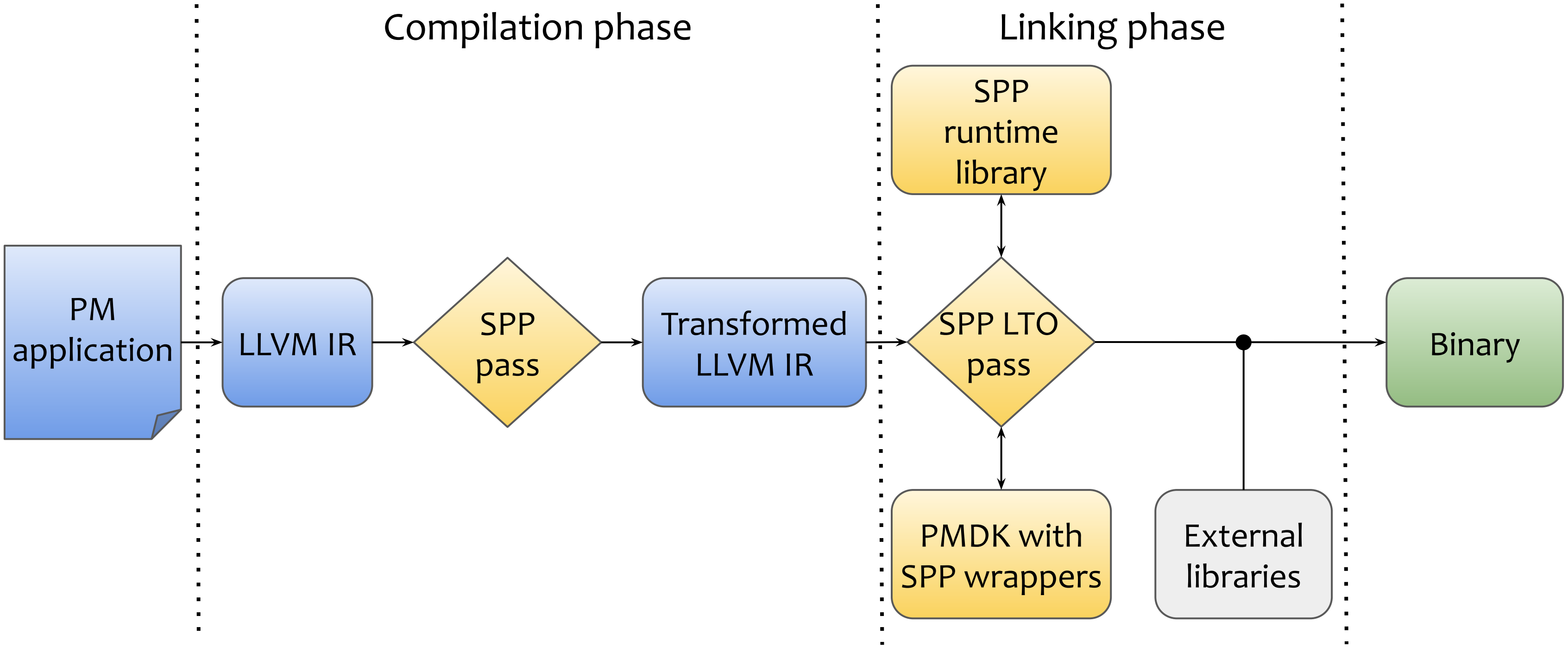
Implementation

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

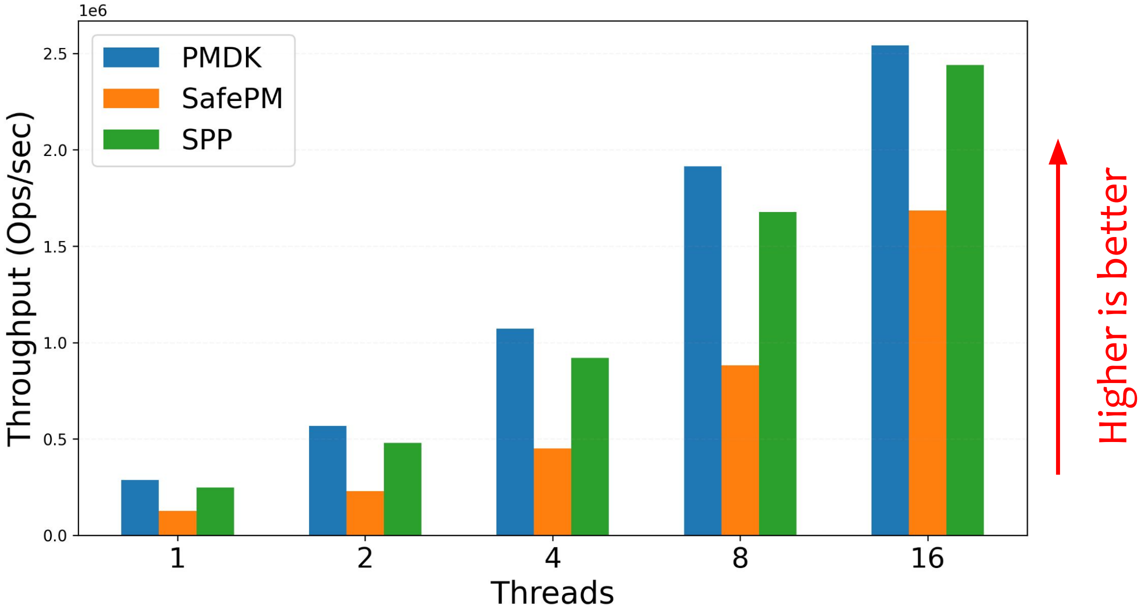
- What is the performance overhead of SPP?
  - Persistent memory KV store (pmemkv), PM phoenix benchmark suite
- How much PM space overhead does SPP introduce?
  - Persistent indices (ctree, rtree, rbtree, hashmap)
- How robust is SPP in detecting memory safety vulnerabilities?
  - RIPE benchmark framework

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

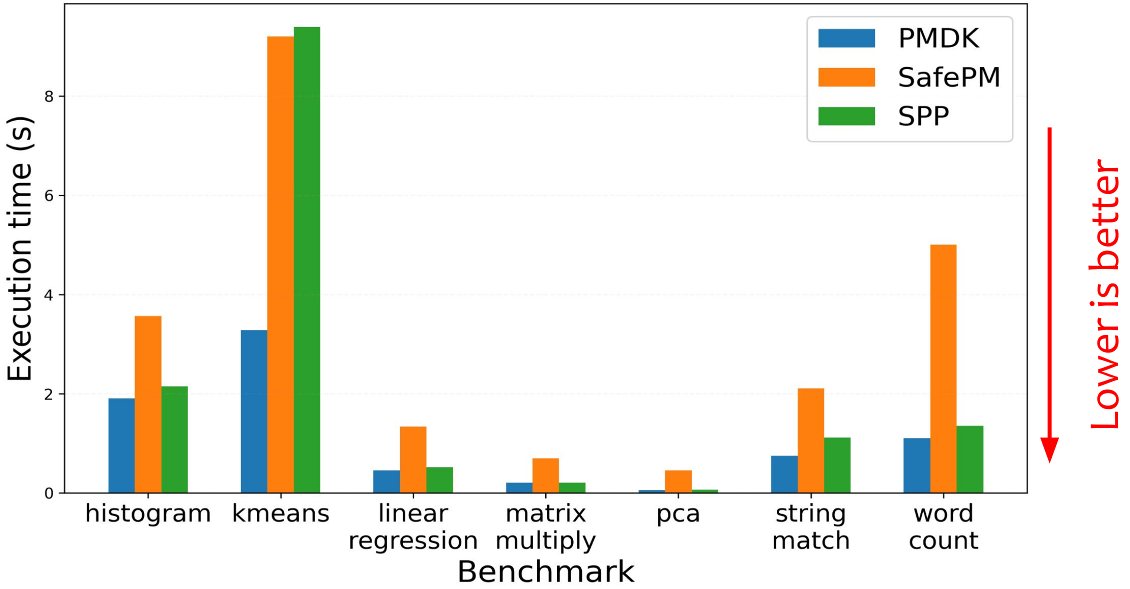
Persistent KV-store benchmark, **10M** ops, **50%** reads / **50%** writes



SPP incurs notably lower performance overheads compared to SafePM

Performance overhead

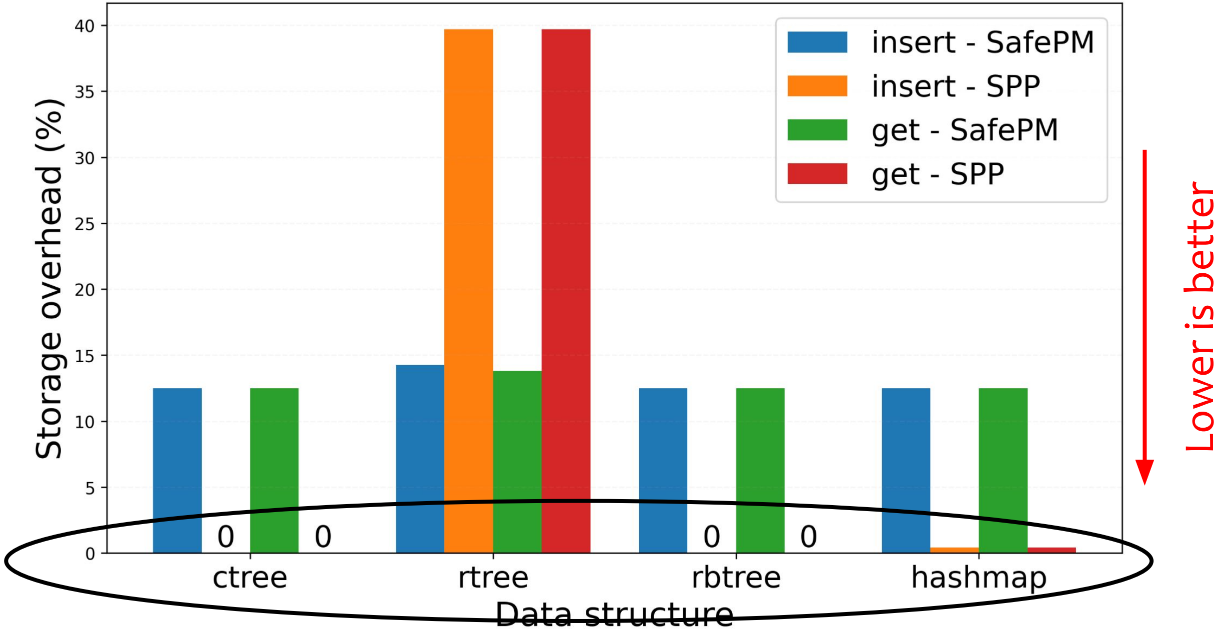
PM Phoenix benchmark, 8 threads, largest dataset for each benchmark



SPP introduces minimal performance overheads even in CPU intensive scenarios

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

Paper **Code**

Try it out!

https://github.com/dimstav23/SPP

Code Reproducible

Dataset Reproducible

Sources

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