Memory Safety for Persistent Memory: Safe Persistent Pointers

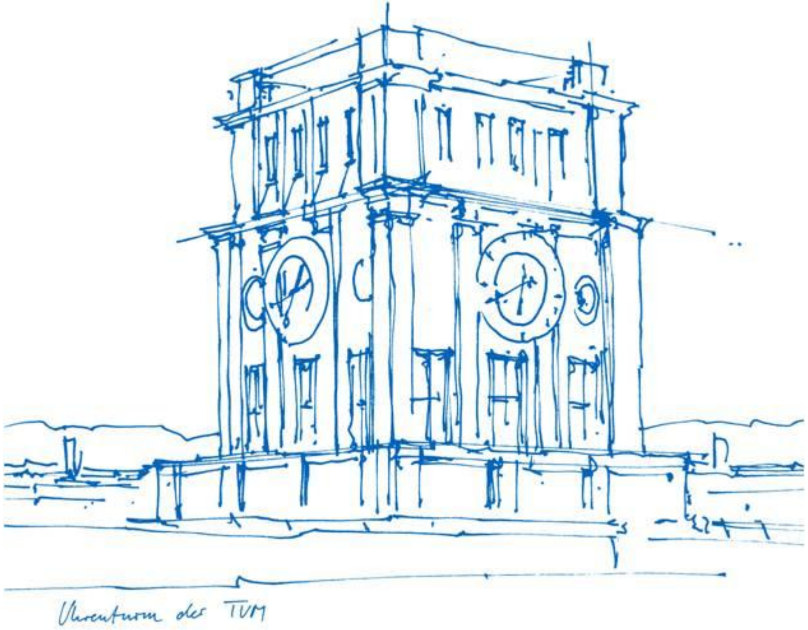
Alexandrina Panfil

Technical University of Munich (TUM)

Department of Informatics

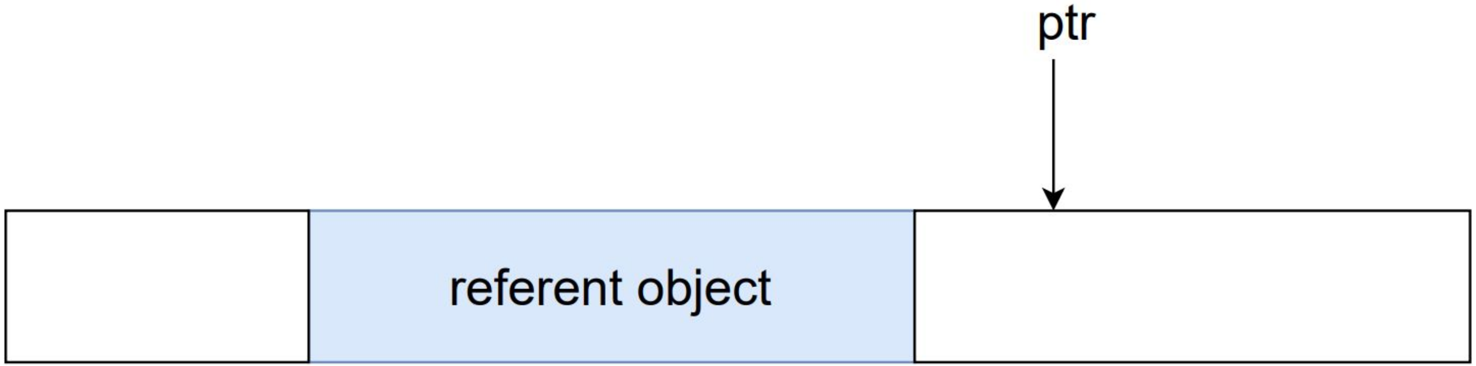
Chair of Decentralized Systems Engineering

Munich, 31st of March 2022

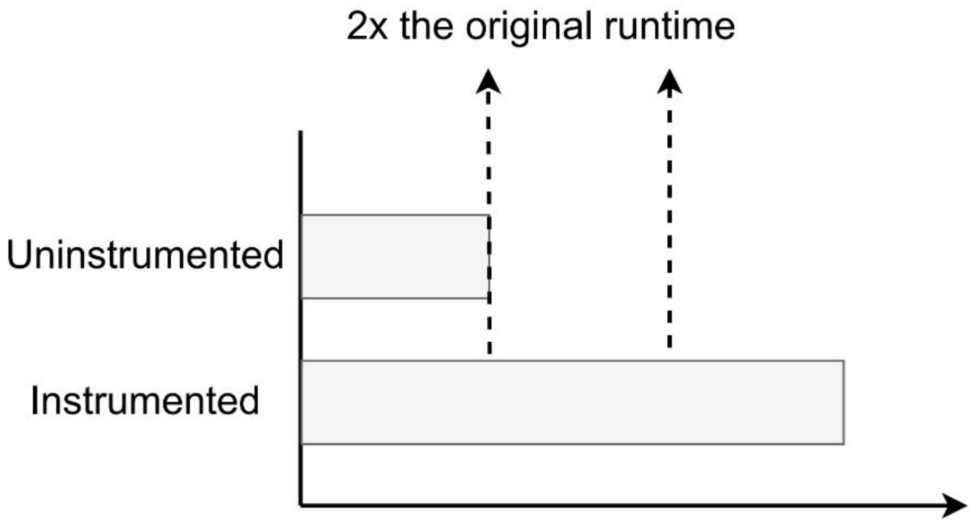


Motivation

- Memory-safety vulnerabilities in C/C++ languages:
  - **Buffer overflows**, dangling pointers, etc.
  - Lead to data leakage, hijack control-flow of the program, etc.



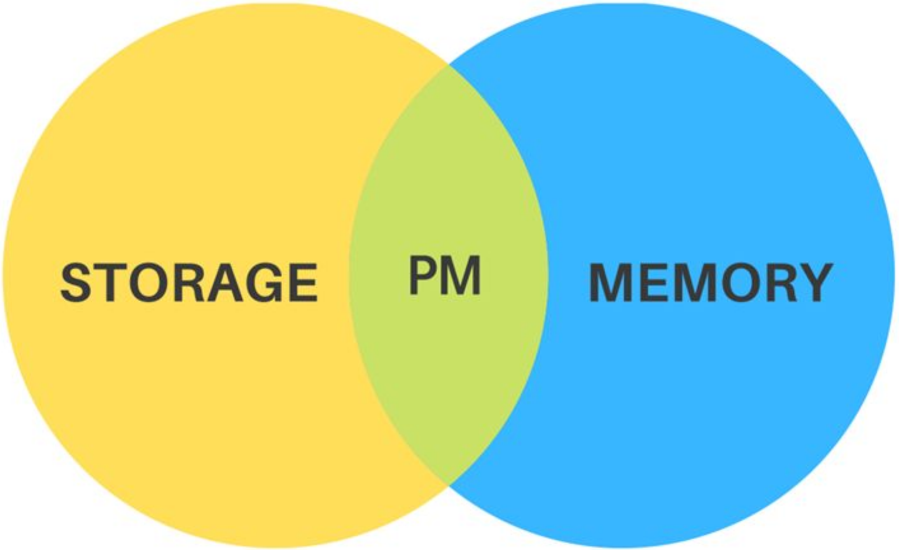
Runtime Overheads



Overhead causes in dynamic analysis tools:

- Metadata update
- Tight loop

Persistent Memory (PM)



Persistent Memory (PM) is susceptible to corruption bugs the same way as volatile memory

Safe Persistent Pointers (SPP)

SPP is a memory safety tool designed for Persistent Memory (PM)

- Leverages the Delta Pointers approach
  - Lower runtime overhead
  - Insignificant space overhead
- Ensures protection against buffer-overflow attacks

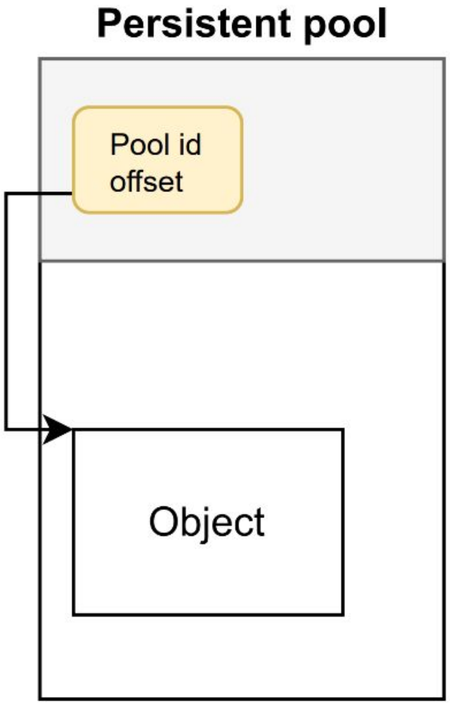
Outline

- ~~Motivation~~
- Background
- Design
- Implementation
- Evaluation
- Summary

Background

PM can be managed through PMDK:

- API for managing persistent objects in **`libpmemobj`**
- Stores metadata about each persistent object



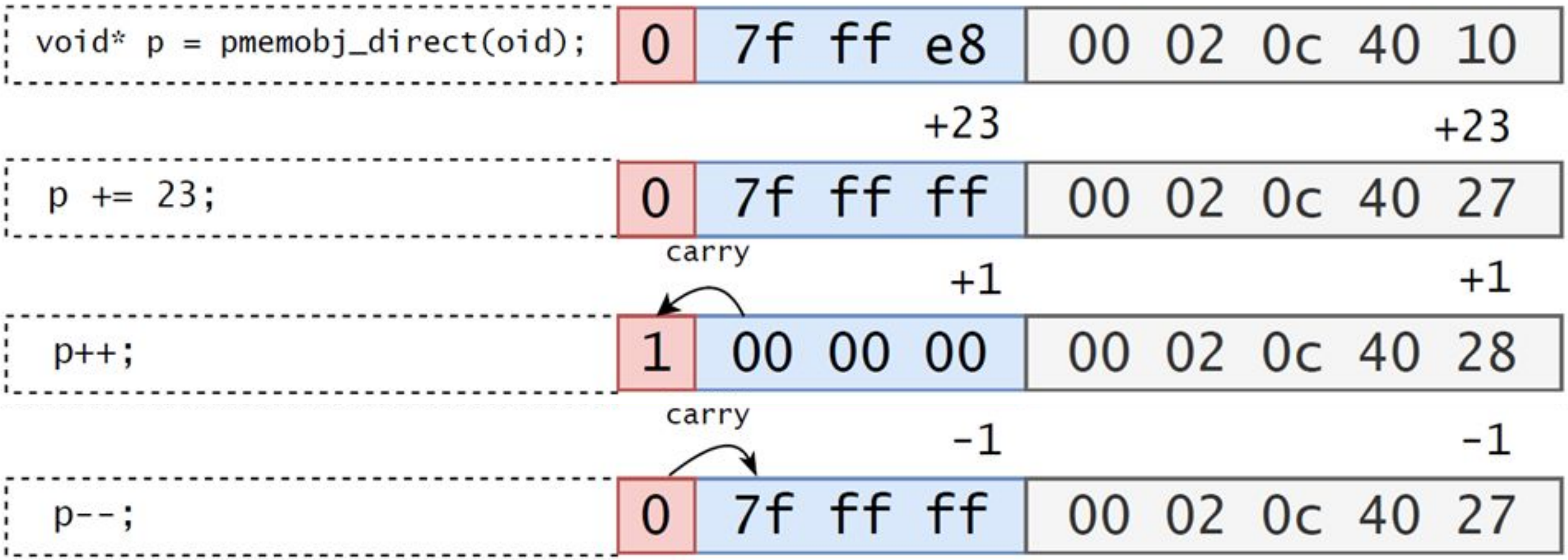
Outline

- ~~Motivation~~
- ~~Background~~
- Design
- Implementation
- Evaluation
- Summary

Design

- First 25 bits reserved for tag encoding (nr. of bits is configurable)
- Initial tag is the negated size of the persistent object

Design


void* p = pmemobj_direct(oid);
0
7f ff e8
00 02 0c 40 10
+23
+23
p += 23;
0
7f ff ff
00 02 0c 40 27
carry
+1
+1
p++;
1
00 00 00
00 02 0c 40 28
carry
-1
-1
p--;
0
7f ff ff
00 02 0c 40 27

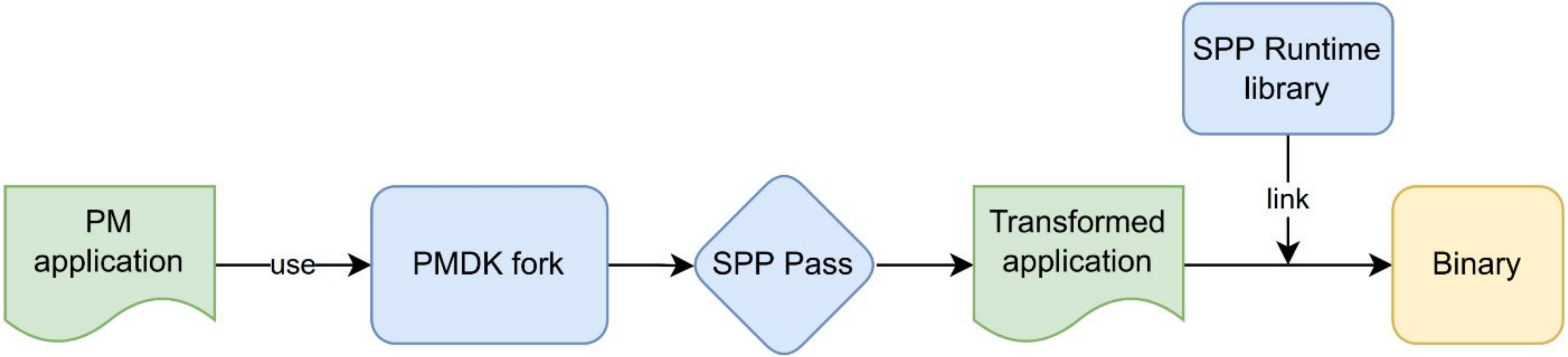
Outline

- ~~Motivation~~
- ~~Background~~
- ~~Design~~
- Implementation
- Evaluation
- Summary

Implementation

Two main parts:

- PMDK fork
- LLVM-assisted instrumentation Stack:
  - Transformation Pass
  - Runtime library
  - Link-Time-Optimization Pass



Implementation

- Additional field for size in persistent object handle
- Native pointer with the encoded tag returned by **`pmemobj_direct()`**

```c
typedef struct pmemoid {
   uint64_t pool_uuid_lo;
   uint64_t off;
   uint64_t size;
} PMEMoid;
```

We introduce changes to **`libpmemobj`** library but maintain the same API

Implementation

Instrumented instructions:

- Pointer arithmetics => Tag update
- Memory accesses => Tag masking, bounds-checking
- External function calls => Tag masking, bounds-checking

Outline

- ~~Motivation~~
- ~~Background~~
- ~~Design~~
- ~~Implementation~~
- Evaluation
- Summary

Evaluation

- Effectiveness:
  - RIPE benchmark
- Performance overhead:
  - pmembench (with ctree, btree, rtree, rbtree, hashmap)
- Baselines:
  - **Non-memory-safe:** native PMDK
  - **Memory-safe:** PMDK with SPP enabled

Evaluation

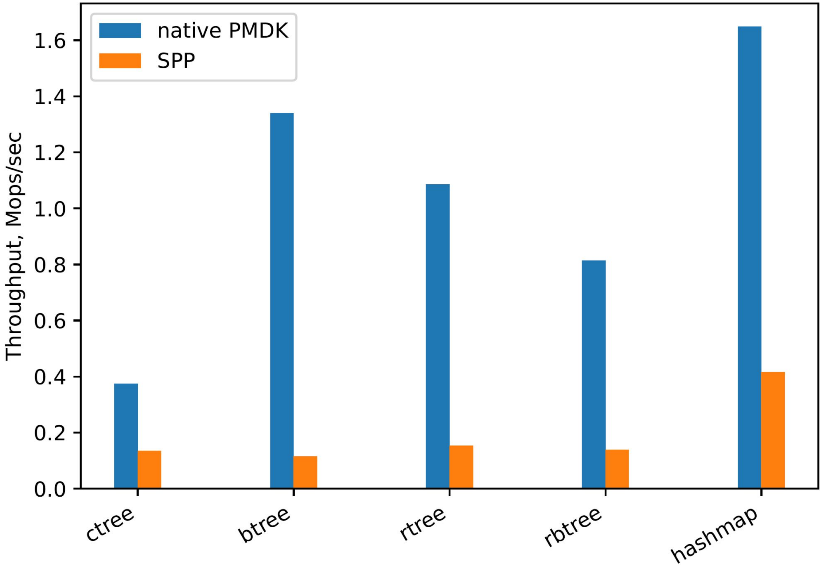
RIPE benchmark, 223 buffer overflow attacks:

RIPE variant	Always	Sometimes	Never
Volatile system heap	83	0	140
PM w/ SPP	4	0	219

SPP reduced the number of successful attacks to 1,7%

Evaluation

Persistent indices, insert/get/remove workloads, relative to PMDK



Biggest performance overhead is in programs with many memory accesses

Outline

- ~~Motivation~~
- ~~Background~~
- ~~Design~~
- ~~Implementation~~
- ~~Evaluation~~
- Summary

Summary

- SPP is a memory safety tool compatible with PM
- Based on the Delta Pointers approach
- Protects against buffer overflow attacks
- Work in progress: performance optimizations

Backup slides

Experimental Setup

- AMD EPYC 7713P CPU with 64 cores,
- 540 GB RAM
- NixOS 21.11 ("Porcupine") with x86_64
- Linux kernel version 5.10.103