Evaluating CHERI for Hardware-Assisted Memory Safety in Databases

Student: Cristian Sandu
Advisor: Martin Fink & Ilya Meignan--Masson
Supervisor: Prof. Pramod Bhatotia

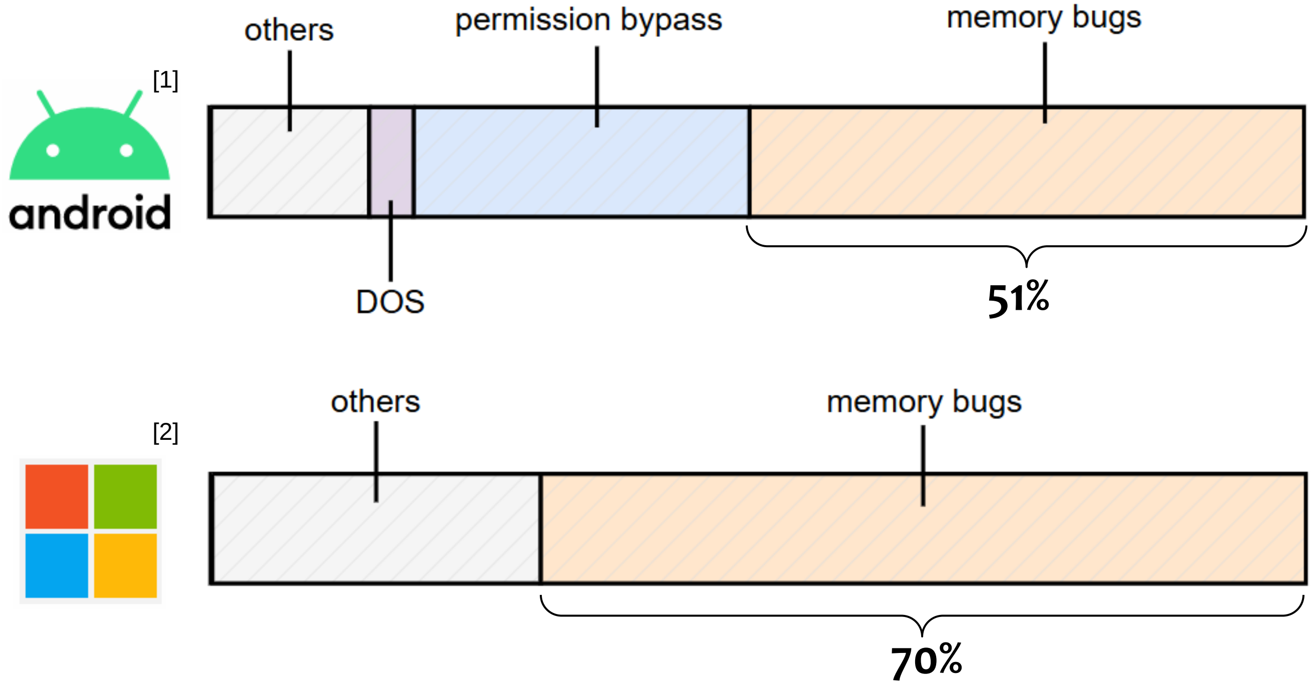
Outline

- Motivation
- Research Gap
- Problem Statement
- Background
- RECIPE for porting to CHERI
- Evaluation

Outline

- **Motivation**
- Research Gap
- Problem Statement
- Background
- RECIPE for porting to CHERI
- Evaluation

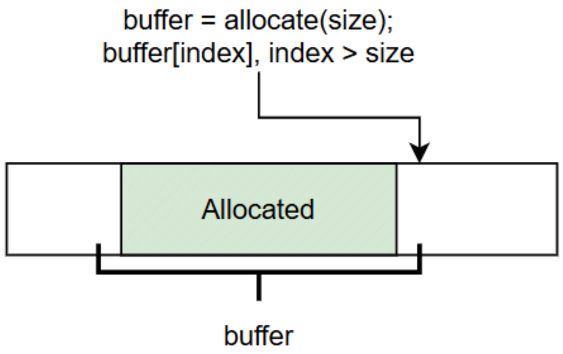
Memory Safety in Software, everywhere



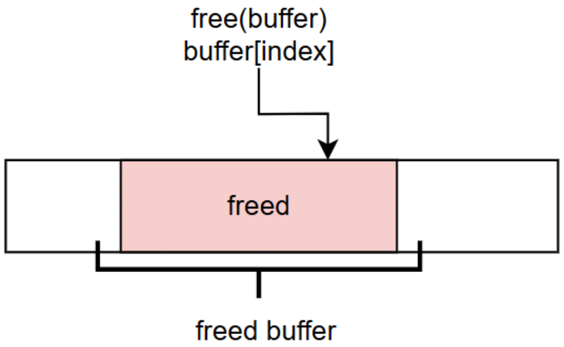
[1] Android Memory Safety
[2] Microsoft Memory Safety

Memory safety

- **Spatial Memory Safety**
  - **violation:** buffer overflow, stack overflow, etc.



- **Temporal Memory Safety**
  - **violations:** use-after-free, double free, etc.



Outline

- Motivation
- **Research Gap**
- Problem Statement
- Background
- RECIPE for porting to CHERI
- Evaluation

Securing Data Structures for DBMS

- there are no memory safety solutions designed for DB systems:
  - Hashtables
  - Trees

- general software solutions:
  - ASLR, DEP, Stack Canaries, etc.
  - AddressSanitizer, Valgrind

Outline

- Motivation
- Research Gap
- **Problem Statement**
- Background
- RECIPE for porting to CHERI
- Evaluation

CHERI technology

- **Databases** are susceptible to attacks that exploit memory vulnerabilities.

- **CHERI** tackles memory vulnerabilities by introducing fine-grained memory protection.

- the community did not focus on applying the CHERI to databases:
  - No RECIPE for porting complex data structures to CHERI
  - Unknown Overheads (Performance, Memory, etc.)

Outline

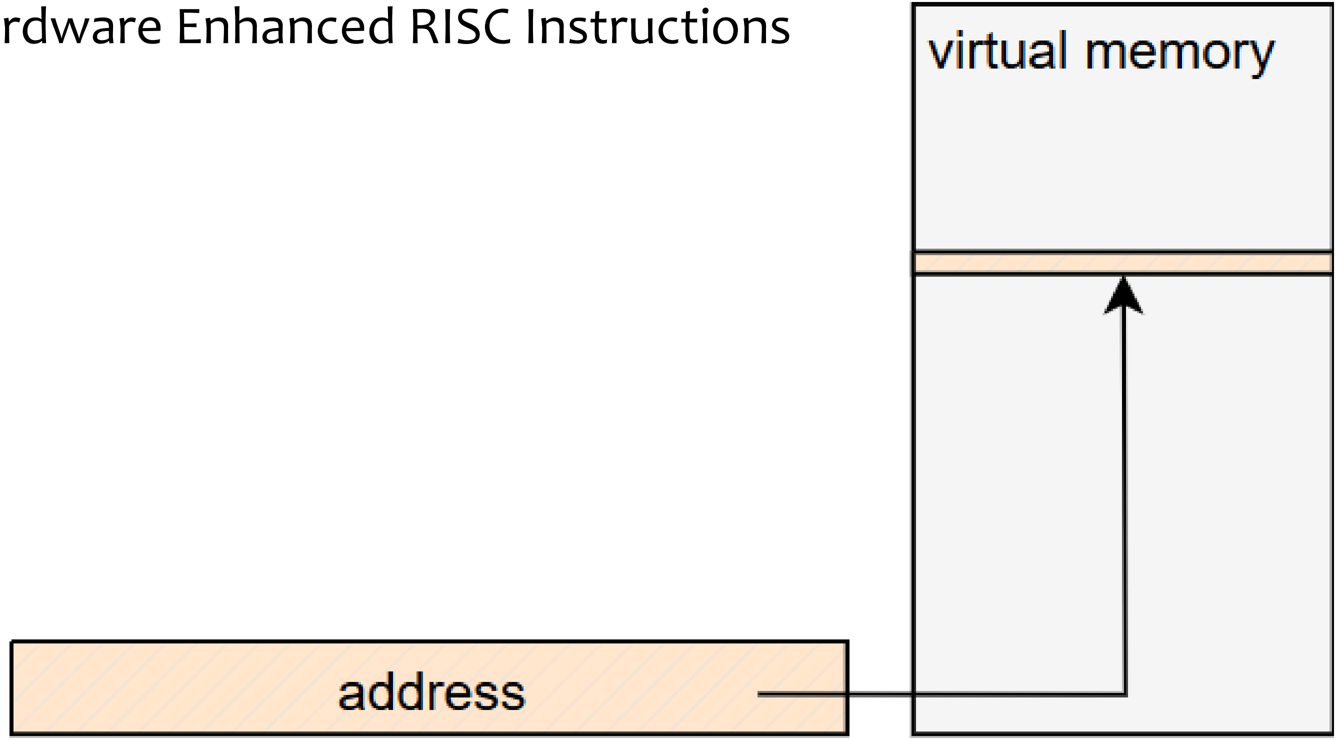
- Motivation
- Research Gap
- Problem Statement
- **Background**
- RECIPE for porting to CHERI
- Evaluation

CHERI

CHERI - Capability Hardware Enhanced RISC Instructions

CHERI

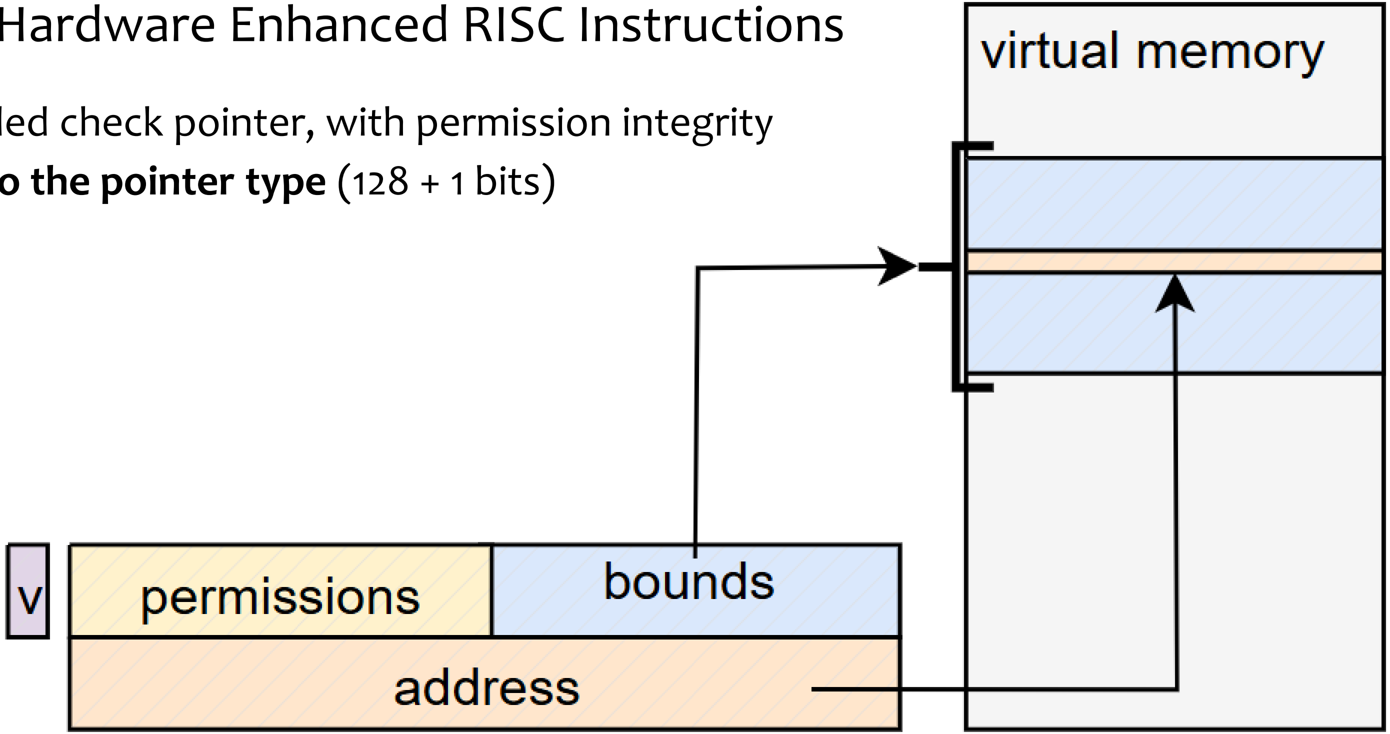
CHERI - **Capability** Hardware Enhanced RISC Instructions



CHERI

CHERI - **Capability** Hardware Enhanced RISC Instructions

- capability = bounded check pointer, with permission integrity
- **metadata added to the pointer type** (128 + 1 bits)



C**HERI** - Capability **Hardware Enhanced RISC Instructions**

- newly added instructions:
  - memory access (load & store) via boundary & permissions check
  - permissions managment: decrease bounds, decrease permissions, etc.

CHERI

CHERI faces disadvantages due to incompatibilities arising from the capability pointer format:

- pointers’ capabilities of meta-data can interfere with pointer-arithmetic operations.
- pointers’ size is extended, increasing the size of structures, breaking paddings, etc.

Outline

- Motivation
- Research Gap
- Problem Statement
- Background
- **RECIPE for porting to CHERI**
- Evaluation

RECIPE for porting to CHERI

- **Data structures** ported on CHERI:
  - Cache-Line Hash Table (CLHT): Lock-Based & Lock-Free versions
  - Cuckoo (libcuckoo library)
  - oneTBB (library) :: Concurrent Hash-Map (and subtypes)

  - Adaptive Radix Tree
  - Google BTree

RECIPE for porting to CHERI

1. Compiler & CHERI community

2. Base-Addresses used in Operations
    1. Pointer's Memory Address is used in Operations
    2. Primitive data is used interchangeably with pointer type

3. Source of provenance

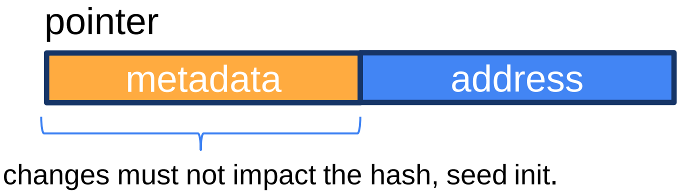
4. Pointer Size Assumptions
    1. Updating the structures
        1. Re-Padding
        2. Re-Sizing
    2. Updating Functions

RECIPE for porting to CHERI

1. Compiler & CHERI community – compatibility with C++
   - standard libraries (boost, stdlib, etc.) are supported by the CHERI platform.
   - code that clearly separates pointer and primitive types does not require adjustments.

RECIPE for porting to CHERI

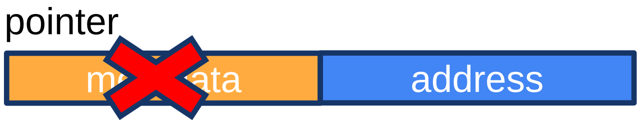
1. Compiler & CHERI community
2. Base-Addresses used in Operations
   a. **Pointer’s Memory Address is used in Operations**
      - used in: hash function computations, seed initialization, etc.




```
get(void* address) {
    return addresstable[hash(address)];
}
```

RECIPE for porting to CHERI

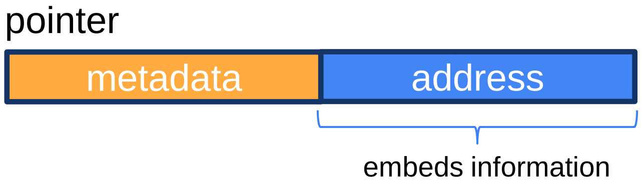
1. Compiler & CHERI community
2. Base-Addresses used in Operations
   a. **Pointer’s Memory Address is used in Operations**
      - used in: hash function computations, seed initialization, etc.




```
get(void* address) {
#ifdef __CHERI_PURE_CAPABILITY__
    return addresstable[hash(cheri_getaddress(address))];
#else
    return addresstable[hash(address)];
#endif
}
```

RECIPE for porting to CHERI

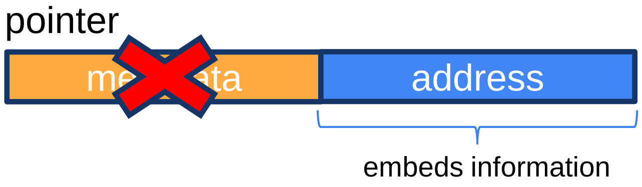
1. Compiler & CHERI community
2. Base-Addresses used in Operations
   a. Pointer's Memory Address is used in Operations
   b. Primitive data is used interchangeably with pointer type
      - embedding information into pointer,




```
lock(void *address) {
    state_type s = state.get();
    if (!(s & BUSY)) {
        state.update(address, WRITER);
    }
 }
```

RECIPE for porting to CHERI

1. Compiler & CHERI community
2. Base-Addresses used in Operations
   a. Pointer's Memory Address is used in Operations
   b. Primitive data is used interchangeably with pointer type
      - embedding information into pointer,



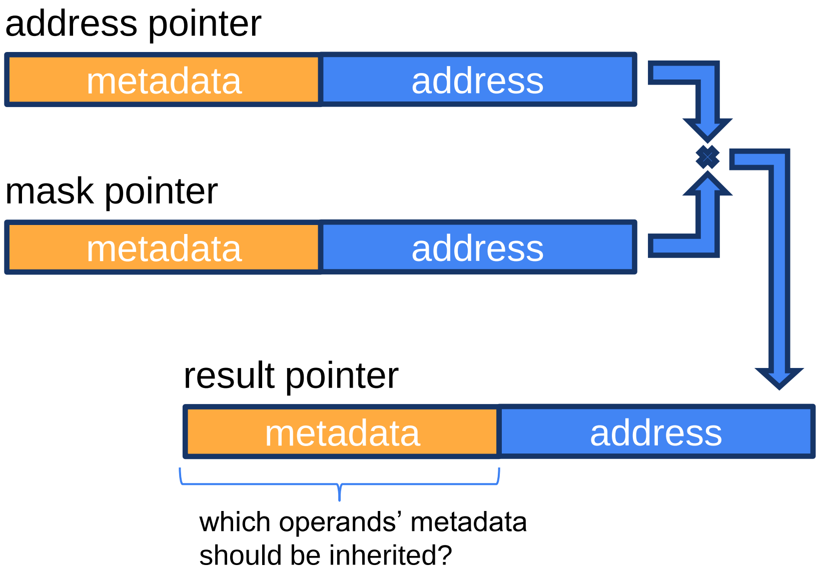

```
lock(void *address) {
   state_type s = state.get();
#ifdef __CHERI_PURE_CAPABILITY__
    if (!(cheri_getaddress(s) & BUSY)) {
#else
    if (!(s & BUSY)) {
#endif
         state.update(address, WRITER);
    }
 }
```

RECIPE for porting to CHERI

1. Compiler & CHERI community
2. Base-Addresses used in Operations
3. Source of provenance

multiple pointers are involved in arithmetic operations.

```
void *foo() {
    void *address = get_address();
    void *mask    = get_coaddress();
    return (address ^ mask);
}
```



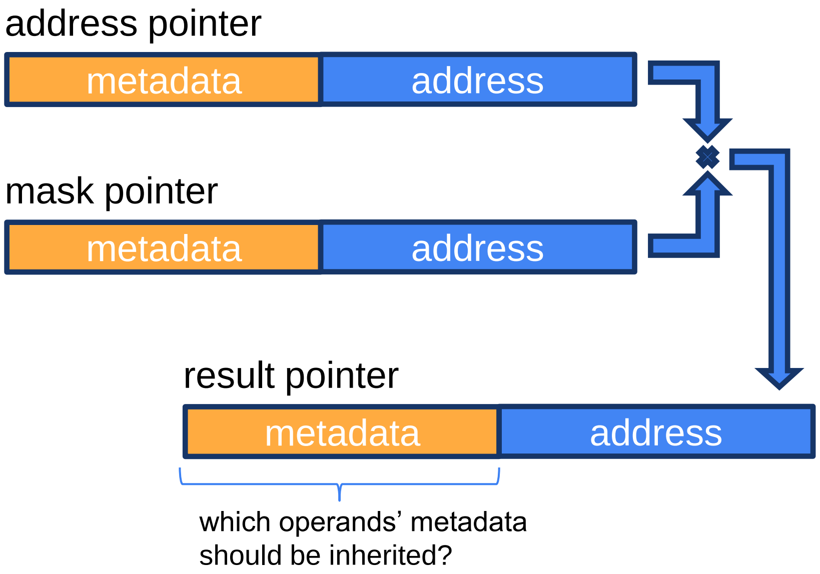
RECIPE for porting to CHERI

1. Compiler & CHERI community
2. Base-Addresses used in Operations
3. Source of provenance

multiple pointers are involved in arithmetic operations.

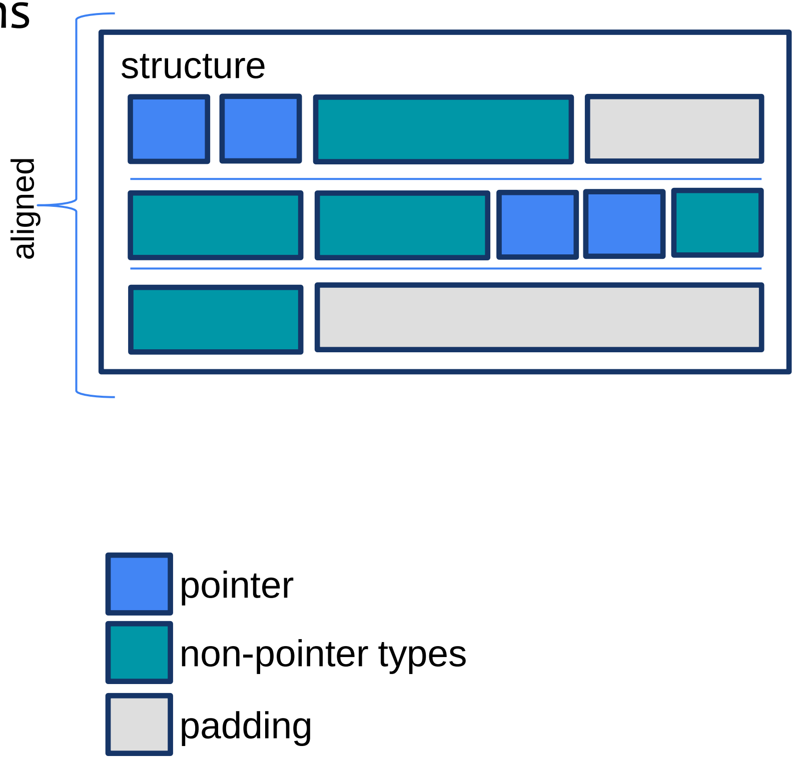
```c
void *foo() {
    void *address = get_address();
    void *mask    = get_coaddress();

    /* breaking down the operation */
    address ^= mask;
    return address;
}
```



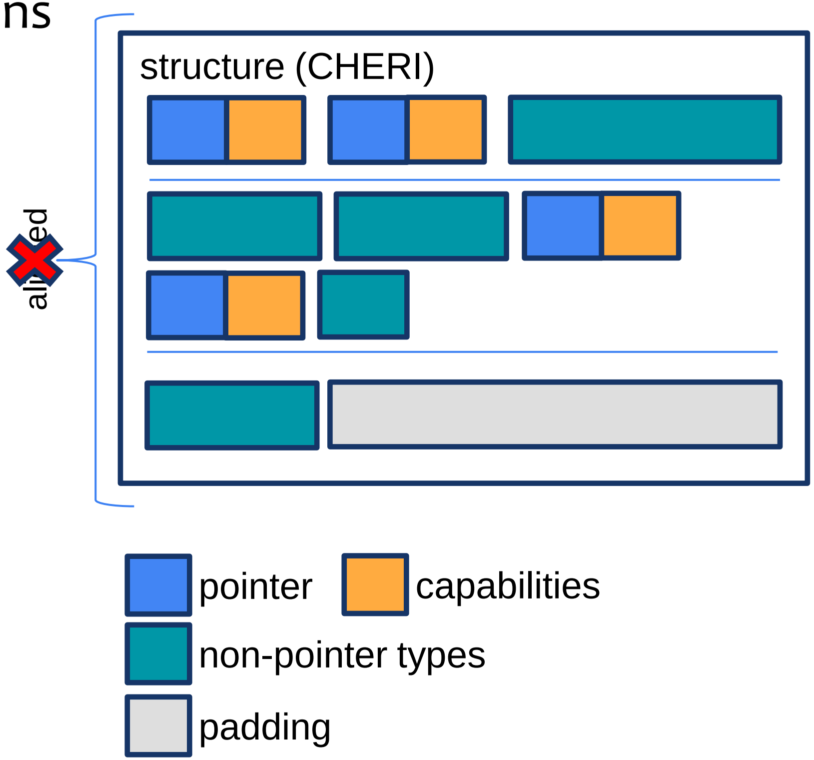
RECIPE for porting to CHERI

1. Compiler & CHERI community
2. Base-Addresses used in Operations
3. Source of provenance
4. Pointer Size Assumptions
    a. **updating the structures**
        i. **re-padding structures**



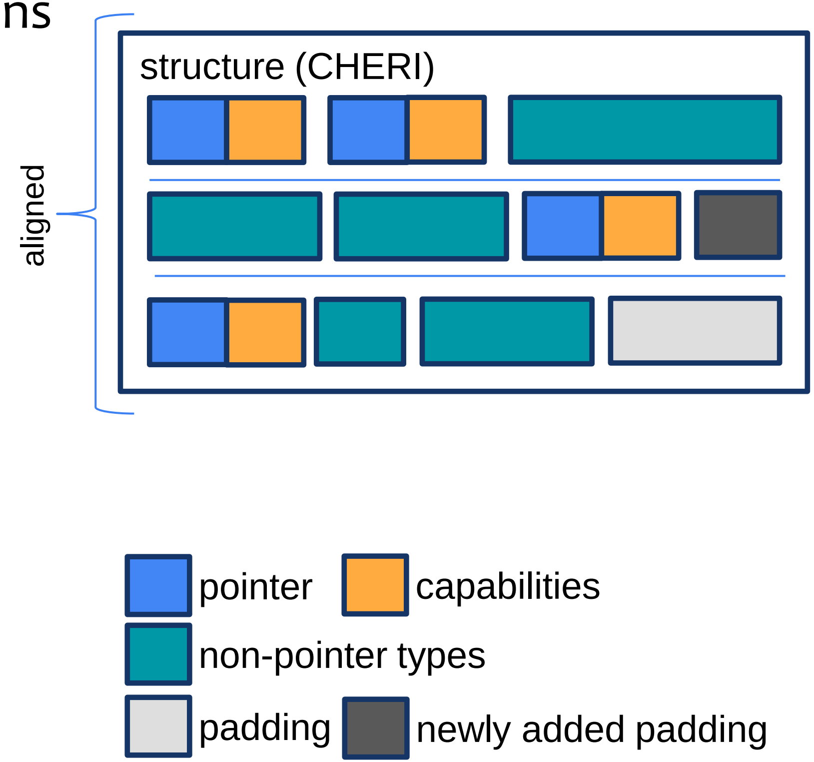
RECIPE for porting to CHERI

1. Compiler & CHERI community
2. Base-Addresses used in Operations
3. Source of provenance
4. Pointer Size Assumptions
    a. **updating the structures**
        i. **re-padding structures**



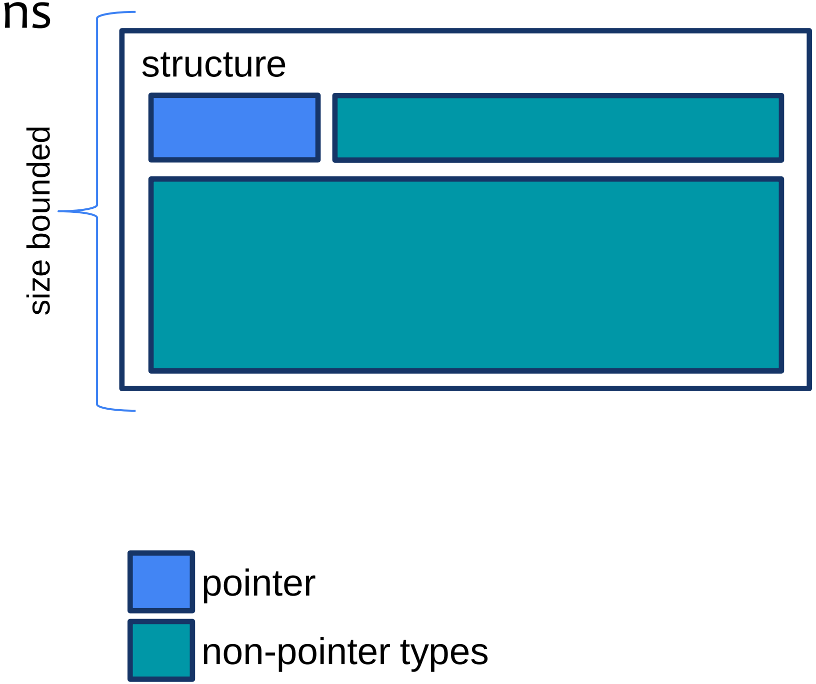
RECIPE for porting to CHERI

1. Compiler & CHERI community
2. Base-Addresses used in Operations
3. Source of provenance
4. Pointer Size Assumptions
   a. **updating the structures**
      i. **re-padding structures**



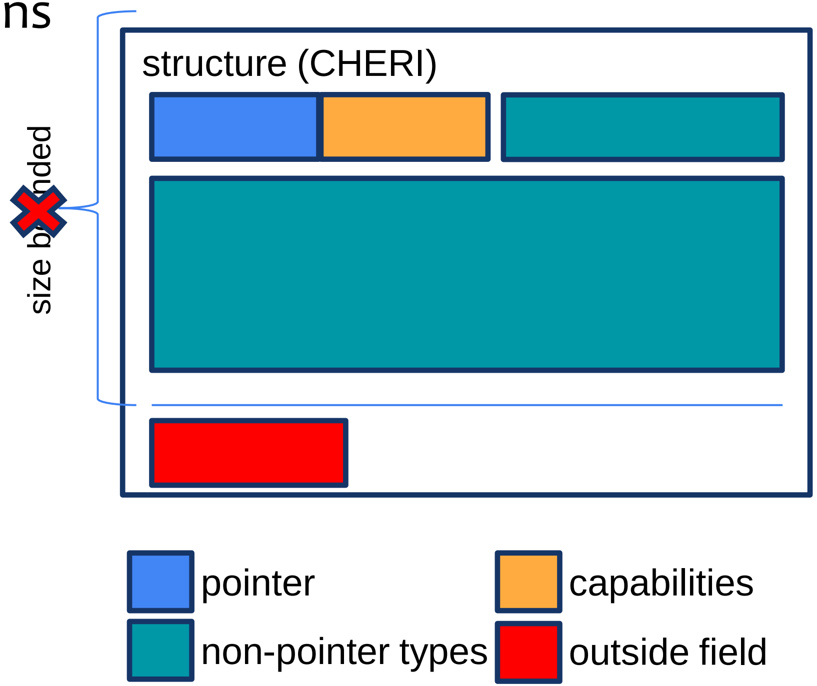
RECIPE for porting to CHERI

1. Compiler & CHERI community
2. Base-Addresses used in Operations
3. Source of provenance
4. Pointer Size Assumptions
   a. **updating the structures**
      i. re-padding structures
      ii. **re-sizing structures**



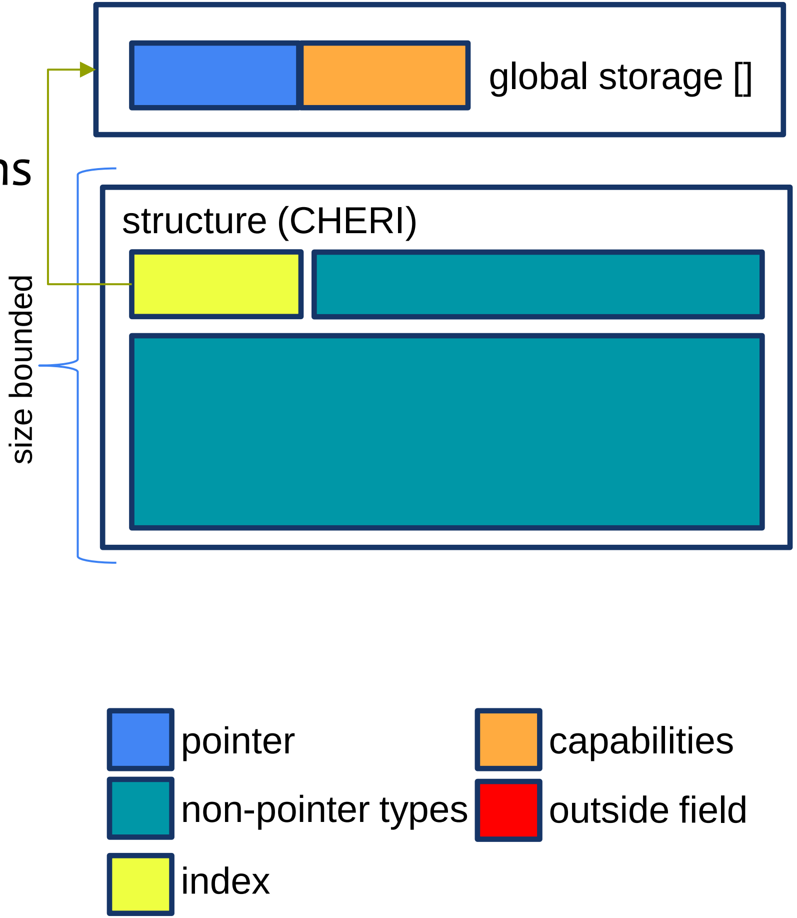
RECIPE for porting to CHERI

1. Compiler & CHERI community
2. Base-Addresses used in Operations
3. Source of provenance
4. Pointer Size Assumptions
   a. **updating the structures**
      i. re-padding structures
      ii. **re-sizing structures**



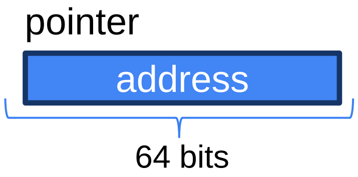
RECIPE for porting to CHERI

1. Compiler & CHERI community
2. Base-Addresses used in Operations
3. Source of provenance
4. Pointer Size Assumptions
   a. **updating the structures**
      i. re-padding structures
      ii. **re-sizing structures**



RECIPE for porting to CHERI

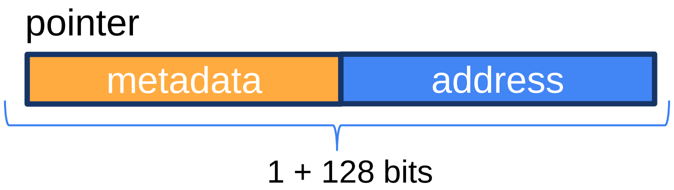
1. Compiler & CHERI community
2. Base-Addresses used in Operations
3. Source of provenance
4. Pointer Size Assumptions
   a. updating the structures
      i. re-padding structures
      ii. re-sizing structures
   b. updating the functions, changing arguments




```
void *foo(void* address) {
    void *mem = allocate_mem(sizeof(address));
    return mem;
}
```

RECIPE for porting to CHERI

1. Compiler & CHERI community
2. Base-Addresses used in Operations
3. Source of provenance
4. Pointer Size Assumptions
   a. updating the structures
      i. re-padding structures
      ii. re-sizing structures
   b. updating the functions, changing arguments




```
#ifdef __CHERI_PURE_CAPABILITY__
void *foo(uint64_t address) {
#else
void *foo(void* address) {
#endif
    void *mem = allocate_mem(sizeof(address));
    return mem;
}
```

Evaluation Setup

- **Custom Framework** was developed for evaluating the data-structures
  - designed microbenchmarks to evaluate the latency, memory and PMCstats for all data-structure operations – insert, query and delete.
  - used a modified version of YCSB to support data-structure evaluation.

- **Evaluation**
  - performed using ARM (without capabilities) as baseline.
  - experiments performed on 100.000 operations.
  - filling factor of 95%.

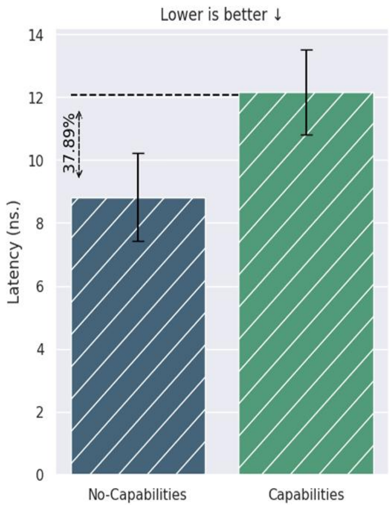
Evaluation

1. Latency – Microbenchmarks, Macrobenchmarks (YCSB)
    1. How does CHERI impact the execution time of DBMS operations?
2. Memory Overhead – Microbenchmarks
    1. How much extra memory is required to store capabilities compared to traditional pointers?
3. Cache Usage – Microbenchmarks
    1. Does the larger size of capabilities affect cache efficiency?

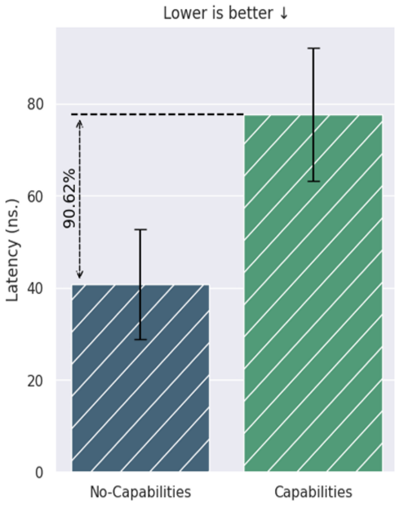
Evaluation :: Latency Overhead

Insertion Operation:

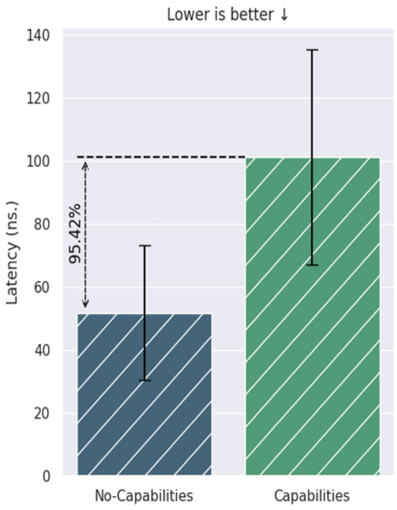
- hashtables: 40% - 90%
- trees: 95% - 130%



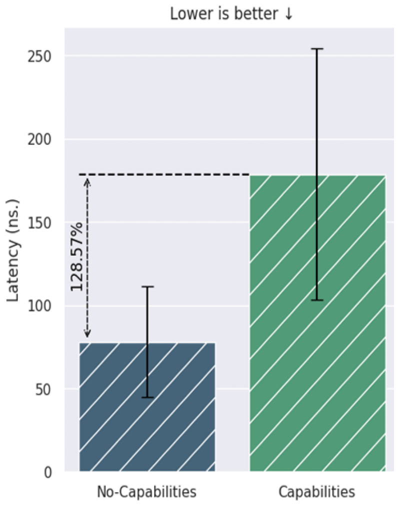
CLHT Lock-Free



oneTBB Concurrent HashMap



Adaptive Radix Tree

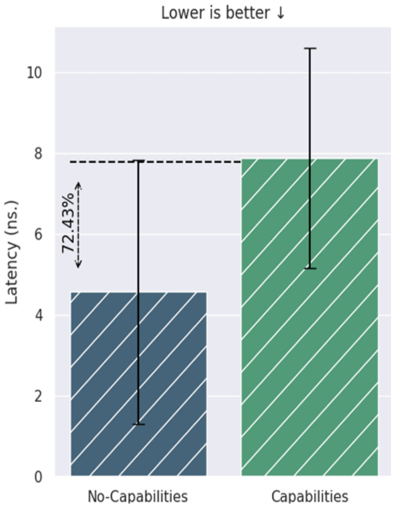


Google BTree

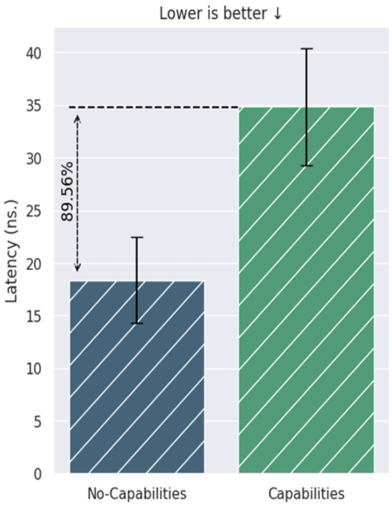
Evaluation :: Latency Overhead

Query Operation:

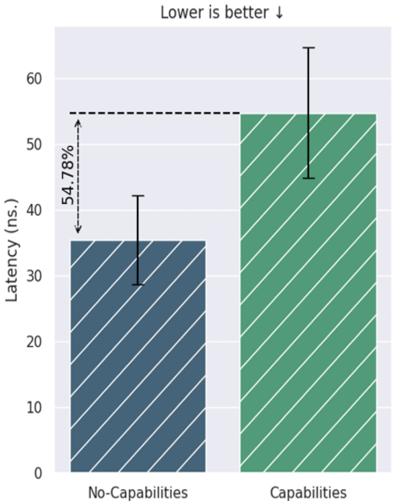
- hashtables: 70% - 80%
- trees: 50% - 120%



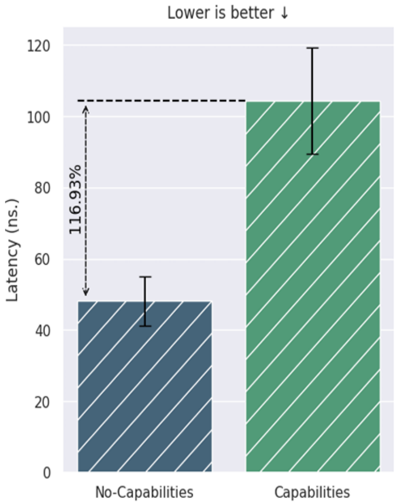
CLHT Lock-Free



oneTBB Concurrent HashMap



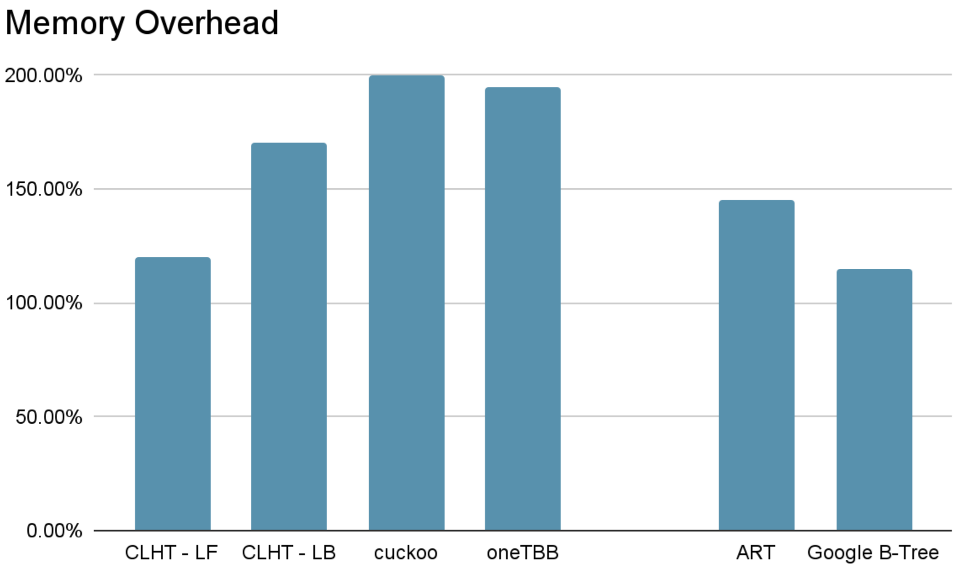
Adaptive Radix Tree



Google BTree

Evaluation :: Memory Overhead

- hashtables: 120% - 200%
- trees: 115% - 150%



Evaluation :: Cache Overhead

Data Structure	Cache L1 (Miss / Read) Overhead
Cuckoo	~ 1.0x
oneTBB Concurrent Hashtable	~ 1.2-2.0x
CLHT Lock-Free	~ 3.0-5.0x
CLHT Lock-Based	~ 3.0-5.0x
Adaptive Radix Tree	~ 1.1-1.2x
Google BTree	~ 1.1-1.2x

Summary

- Databases are vulnerable to memory-based exploits.
  - CHERI offers a solution to mitigate the memory-safe problem.

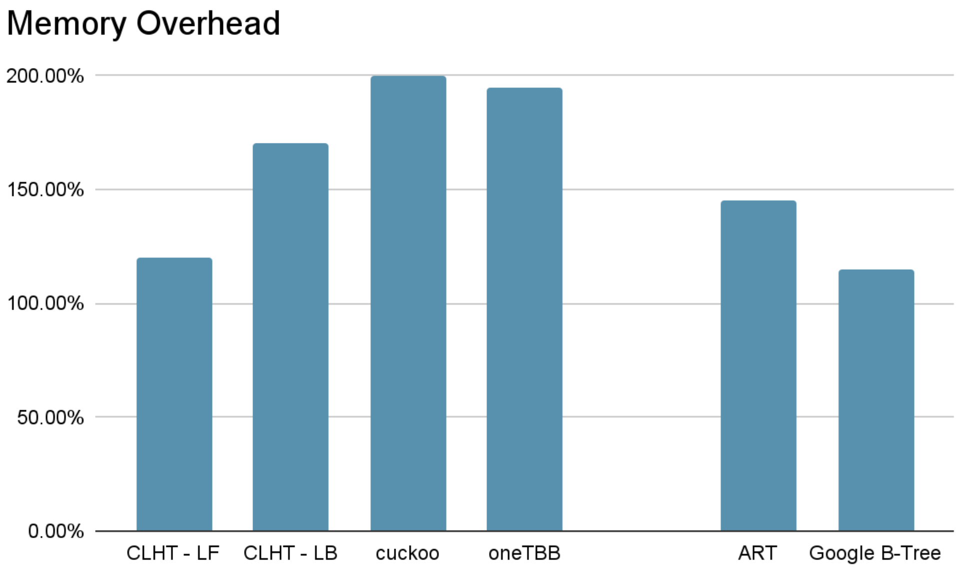
- CHERI requires careful consideration during the porting procedure.
  - CHERI uses larger pointers to accommodate metadata for memory protection.
  - it impacts data structure, existing code, and external-library usage.
  - existing code often needs adjustments to handle these larger pointers.

- CHERI technology introduces performance overheads.
  - CHERI's security benefits come with potential performance trade-offs in:
  memory management, latency, and cache efficiency.

Backup

Evaluation :: Memory Overhead

- hashtables: 120% - 200%
- trees: 115% - 150%



Evaluation :: Performance Overhead

Insertion Operation:

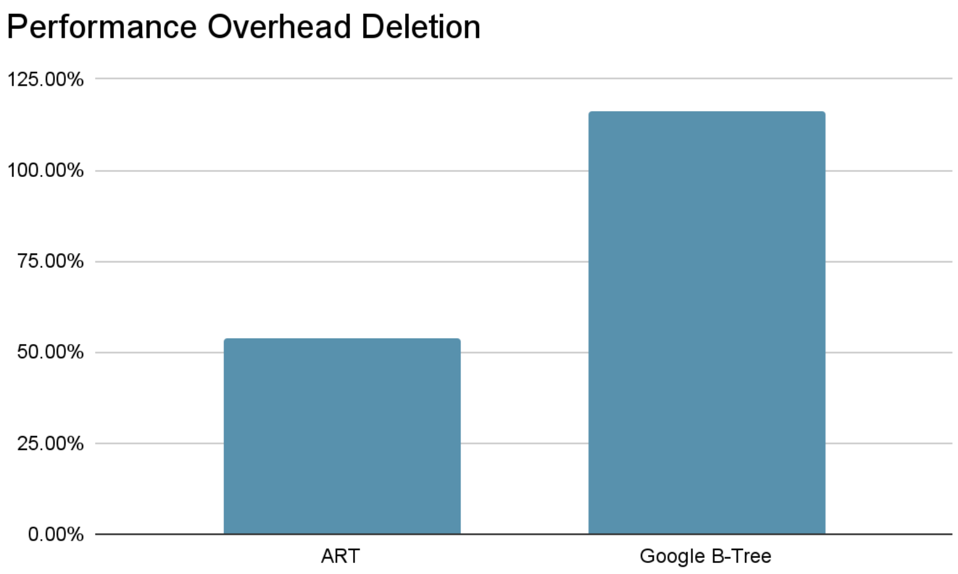
- hashtables: 35% - 140%
- trees: 95% - 130%

Query Operation:

- hashtables: 70% - 80%
- trees: 50% - 120%

Deletion Operations:

- trees: 55% - 125%



Evaluation :: Cache Overhead

Data Structure	PMC Stats			
	Stat Name	No Capability	Capability	Increase
Cuckoo Hashtable	LL read	29.2 M.	27.5 M.	0.941x
	LL miss	18.0 M.	15.9 M.	0.883x
	L1 read	4772.8 M.	4726.4 M.	0.99x
	L1 write	2801.8 M.	3522.2 M.	1.257x
OneTBB Conc. HT.	LL read	2.5 M.	2.9 M.	1.16x
	LL miss	2.2 M.	2.7 M.	1.227x
	L1 read	1160.3 M.	1284.8 M.	1.107x
	L1 write	868.0 M.	1002.7 M.	1.155x
CLHT Lock-Base	LL read	86.7 M.	321.9 M.	3.712x
	LL miss	54.2 M.	248.7 M.	4.588x
	L1 read	1861.2 M.	1810.7 M.	0.972x
	L1 write	841.1 M.	854.0 M.	1.015x
CLHT Lock-Free	LL read	1.8 M.	3.1 M.	1.722x
	LL miss	1.7 M.	3.1 M.	1.823x
	L1 read	749.1 M.	1177.4 M.	1.571x
	L1 write	542.0 M.	902.2 M.	1.664x

Evaluation :: Cache Overhead

Data Structure	PMC Stats			
	Stat Name	No Capability	Capability	Increase
Adaptive Radix Tree	LL read	3.0 M.	3.8 M.	1.266x
	LL miss	2.9 M.	3.8 M.	1.31x
	L1 read	1497.5 M.	1651.9 M.	1.103x
	L1 write	1056.7 M.	1222.1 M.	1.156x
Google BTree	LL read	3.0 M.	3.6 M.	1.2x
	LL miss	3.0 M.	3.6 M.	1.2x
	L1 read	1725.6 M.	1884.8 M.	1.092x
	L1 write	1238.1 M.	1416.3 M.	1.143x