DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master’s Thesis in Informatics

Evaluating CHERI for Hardware-Assisted Memory Safety in Databases

Cristian Sandu

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Evaluating CHERI for Hardware-Assisted Memory Safety in Databases

Author:	Cristian Sandu
Supervisor:	Prof. Pramod Bhatotia
Advisor:	Martin Fink & Ilya Meignan--Masson
Submission Date:	28.Oct.2024

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 28.Oct.2024 Cristian Sandu

Acknowledgments

First and foremost, I extend my deepest gratitude to my thesis advisors, Ilya Meignan–Masson and Martin Fink. Their expert guidance was essential in shaping my research direction. They generously shared valuable insights that consistently challenged and broadened my thinking. Moreover, their unwavering support and availability throughout this journey were invaluable in helping me navigate any obstacles I encountered. I am incredibly grateful for their dedication and mentorship.

I am thankful to my supervisor, Professor Pramod Bhatotia, for the privilege of conducting my thesis research at the Chair of Distributed and Operating Systems. His insightful suggestions and invaluable advice were instrumental in strengthening my thesis, providing me with a deeper understanding of the research topic. I feel truly fortunate to have had the opportunity to learn from and work with him.

I am deeply appreciative for the guidance and support that have enabled me to reach this significant milestone in my academic career. I owe gratitude to Ilya Meignan-Masson and Martin Fink for their invaluable mentorship, and to Professor Pramod Bhatotia for his unwavering support. Furthermore, the encouragement of my family and friends has been a constant source of strength throughout this journey. I am truly thankful for everyone who has contributed to my success.

Abstract

Database Management Systems (DBMS) are vulnerable to memory-related attacks due to their complex data structures and frequent memory operations. These systems are critical for ensuring data integrity, security, and availability, making them targets for exploits such as buffer overflows and use-after-free errors. Memory violations can lead to data corruption, unauthorized access, system crashes, and significant performance degradation. Moreover, protecting DBMS against these vulnerabilities is essential for compliance with industry regulations that mandate strong data security and integrity standards.

This thesis explores leveraging CHERI, a capability-based security architecture, to enhance DBMS protection. CHERI's fine-grained memory protection enables precise control over memory access, potentially mitigating these vulnerabilities. We investigate adapting core DBMS data structures, specifically hash tables and trees, to the CHERI environment. Our work focuses on porting data structures to the CHERI capabilities environment and evaluating their performance impact compared to conventional implementations.

With a thorough evaluation of the ported data structures with micro and macro-benchmarks, we observe that CHERI imposes a significant overhead to all the operations with latency overheads of up to 60% and a memory overhead of over 30%.

Contents

1. Introduction

The ever-growing dependence on data-driven applications, particularly in artificial intelligence, underscores the critical role of databases in modern society. Memory safety, in particular, has become a crucial concern, as vulnerabilities could cause consequences for data integrity and system stability. One of the essential security aspects is memory safety, which has emerged as a critical battleground.

Memory vulnerabilities, such as buffer overflows and use-after-free errors, remain a dominant and prevalent source of exploitable weaknesses in the landscape of software vulnerabilities [1]. Malicious actors can leverage these vulnerabilities to compromise sensitive data and disrupt system stability, highlighting the critical need for robust memory safety mechanisms in data structures used for database systems.

Traditional software hardening techniques, such as Address Space Layout Randomization [2] and Data Execution Prevention [3], provide a rudimentary level of memory protection. Attackers have developed methods to bypass these protections, such as return-oriented programming (ROP) to circumvent DEP and techniques to predict or discover memory layouts to defeat ASLR [4]. These limitations highlight the need for more robust security measures to ensure comprehensive protection against memory-related vulnerabilities

The CHERI (Capability Hardware Enhanced RISC Instructions) architecture offers a promising solution to memory safety challenges by introducing hardware fine-grained memory boundaries and capabilities checks. This significantly enhances software security, enabling precise control over memory access, and preventing common vulnerabilities such as buffer overflows and use-after-free exploits. While the potential benefits of CHERI's pure-capability modules are substantial, the complexities of integrating them into existing software, particularly complex systems like database structures, remain largely unexplored. This thesis aims to investigate these challenges and assess the feasibility of porting such systems to CHERI.

This thesis explores the complexities of porting fundamental database structures using DBMS to leverage the security benefits of a CHERI capabilities-enabled environment. The data structures chosen for this study were selected based on their community popularity, with a deliberate focus on ensuring a diverse range of data structures. This investigation provides insights into the complexities and considerations of leveraging CHERI's enhanced memory protection capabilities in performance-critical database environments.

This thesis conducts a comprehensive analysis of the performance implications associated with utilizing the capabilities provided by the CHERI architecture. To facilitate this performance review, a dedicated testing framework was meticulously developed. This framework enables the evaluation of the performance of a diverse range of data structures operating within a CHERI-enabled environment. The insights gained from this analysis will shed light on the practical performance trade-offs inherent in adopting CHERI for memory safety.

The core contribution of this thesis can be outlined as follows:
Porting to ARM-Morello RISC processors: The changes needed to move the data structure on ARM processors.
Supporting CHERI capabilities: the recipe used allows the data structure to operate with CHERI capabilities enabled.
Performance evaluation: evaluation of data structure operations, specifically focusing on insertion, search, and deletion performance to further assess the performance implications of CHERI's capability-enabled mode.

2. Background

This chapter presents the concepts upon which our research is built. It begins by offering details regarding the database data structures ported to the CHERI architecture, functioning principle, and further classification. After that, the CHERI architecture's key features and security benefits are discussed. Finally, it is presented briefly the performance evaluation framework.

2.1. Data structures in DBMS

Managing and organizing data quickly and reliably in complex database management systems (DBMS) is crucial. To address this challenge, the research community has developed various data structures that ensure quick retrieval, reliable storage, and optimal system performance. The DBMS is a sophisticated system composed of interconnected subsystems, many of which depend on index structures to optimize performance.

Index structures are components of DBMS designed for efficient data retrieval by providing a mechanism for quickly locating specific data records based on their values. They create a mapping between search keys and their corresponding data records, allowing rapid access to the desired information. Common index structures include B-trees, hash tables, tries, and bitmap indexes. This paper's focus is narrowed to hash tables and tree structures.

Hash tables and tree structures are fundamental data structures for efficient data management. Hash tables use a hash function to map keys to values, allowing for average-case constant time complexity for insertion, deletion, and lookup operations. However, their performance can degrade to linear time in the worst case due to hash collisions. In contrast, tree structures, such as binary search trees (BSTs) and AVL trees, maintain data hierarchically and in an ordered manner, providing logarithmic time complexity for these operations. Trees are particularly advantageous when data needs to be stored in a sorted order, as they inherently maintain order through their structure. These structures can be more complex to implement correctly and prone to errors.

Table 2.1.: Ported CHERI data structures

internal structure type	data structure name
hashtable	CLHT (LF & LB), Cuckoo Hashing, oneTBB::concurrent_hash_map
trees	ART and Google-BTree

2.1.1. CLHT (Concurrent Linear Hash Table)

CLHT (Concurrent Linear Hash Table) [5] is a hash-table implementation optimized for high performance in multi-threaded environments. The hash table data resides within the cache to accelerate the operation speed. In this paper, we focused on two specific variants of CLHT, namely the lock-free and lock-based versions. The lock-free version eliminates the need for explicit locking mechanisms through atomic operations and careful memory management. This approach aims to maximize concurrency and minimize contention in highly parallel environments. In contrast, the lockbase version adds traditional locks to synchronize access to shared data structures, decreasing performance.

2.1.2. Cuckoo Hashtable

Cuckoo hashing [6] is a hash table implementation that uses two hash functions and two corresponding arrays to store key-value pairs. This approach minimizes collisions and ensures efficient search, insertion, and deletion operations. When a new key is inserted, it is hashed using both hash functions to determine the location in both arrays. If one of the locations is empty, the key is inserted there. If both places are occupied, one of the existing keys is evicted and rehashed using the second hash function to find a new location, potentially triggering a cascade of evictions.

2.1.3. oneTBB (oneAPI Threading Building Blocks)

The Intel oneAPI Threading Building Blocks (oneTBB) [7] is a popular C++ library for parallel programming that offers a rich set of tools and algorithms to simplify the development of high-performance applications.

The oneTBB library provides a highly efficient, thread-safe hash table for parallel computing environments. Thanks to its fine-grained locking mechanisms, this hash table allows multiple threads to perform insertions, deletions, and lookups concurrently without compromising data integrity. For this project, we utilized oneTBB's `concurrent_hash_map`.

2.1.4. Google BTree

Google's B-tree [8] efficiently implements the B-tree data structure optimized for in-memory use. It's designed to provide a simple, ordered, and performant way to store and retrieve data.

Unlike traditional binary trees, B-trees store multiple keys and children in each node, reducing tree height and minimizing disk access. This makes Google's B-tree particularly well-suited for scenarios with large datasets where performance is critical.

2.1.5. ART (Adaptive Radix Tree)

Adaptive Radix Tree (ART) [9] is an efficient data structure for in-memory indexing, offering good performance for lookup and update operations. ART uses a clever strategy of adaptive node sizes - the structure of each node within the tree dynamically adjusts based on the number of child elements it has. This leads to significant space savings, particularly in scenarios with sparse keys or varying key lengths. Furthermore, ART's design allows for efficient prefix compression and optimized memory access patterns, contributing to its impressive speed.

2.1.6. Memory Safety

Ensuring memory safety has long been a focus in computer science, especially during program development. While low-level languages like C and C++ offer superior performance, they are particularly susceptible to various memory safety issues. Memory exploits are the top CVE (Common Vulnerability Exposure) ranked software vulnerabilities, according to MITRE ranking [10].

Further studies point out the prevalence of memory vulnerability attacks. Memory attacks are invariant of the used operating distribution: around 70% of vulnerabilities on MacOS and iOS [11], 70% of all vulnerabilities in different Microsoft products [12], and 90% of Android vulnerabilities [13].

2.2. CHERI (Capability Hardware Enhanced RISC Instructions)

CHERI (Capability Hardware Enhanced RISC Instructions) [14] is a hardware architectural extension of Armv8-A and for RISC-V processors developed collaboratively by SRI International and the University of Cambridge. CHERI extends the conventional Instruction-Set-Architectures (ISAs) with new architectural features to enhance software security by integrating fine-grained memory protection. It allows program developers to enforce strict memory access control policies, mitigating vulnerabilities such as buffer overflows, use-after-free exploits, code injection attacks, information flow, pointer forgery, and type confusion. CHERI implements fine-grained memory checks directly at the hardware level. Doing so ensures that security measures are significantly more efficient than software-based solutions, promising much better performance.

Morello CHERI Architecture

Building upon the foundational principles of CHERI, the Morello [15] architecture is an experimental research-prototype project designed to further explore and enhance CHERI pointer capabilities at hardware level before scaling up to industrial applications. Morello architecture is designed to extend the ARMv8-A AArch64 processors with the proposed 8th version of CHERI ISA. The Morello architecture is backwards compatible with the existing ARMv8-A architecture. Morello architecture adds the following features:

Capability Data Type: capability is a 128 + 1 bits datatype comprising metadata (boundaries, permissions) and effective value. Capabilities can be used as "Code Capabilities" and "Data Capabilities". Code Capabilities are code pointers that implement control-flow security by constraining the available memory region that can be accessed.

Extended ISA: instructions to manipulate, edit, copy, and fetch fields from capabilities. C64 [15] was added over A64 ISA to allow these operations. The C64 instruction set is an extension designed to handle capabilities, a key feature of the CHERI architecture.

Capabilities Registers: are registered and designed to hold capabilities. Their size is 128+1 bits. The new set of registers is named Cn (from C0 to C30), able to hold 128+1 bits.

Software Development Environment: providing a rich toolchain: C/C++ Morello Compiler, Linkers, debuggers and various assemblers.

2.2.1. Pointer Format

To achieve fine-grained memory protection on Morello systems [16], CHERI extends pointers to 128+1 bits by incorporating metadata specifying the precise memory boundaries offsets for a region, associated permissions (such as read, write, or execute), capability tag, the actual memory address, and source of provenance. The capabilities format of Morello architecture is similar but not identical to the CHERI format. The format of a Morello-CHERI pointer is shown in figure 2.1.

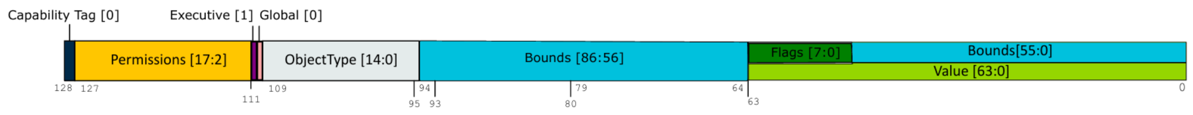


Figure 2.1.: Morello CHERI pointer format [16]

Capability Tag: marks if a pointer is still valid. The capability tag allows the early detection of use-after-free errors. It also helps detect pointer corruption or injection attacks, as an invalid tag indicates a corrupted pointer that cannot be used, thus preventing exploitation. The hardware checks and invalidates the capabilities when the program writes non-pointer values at a memory location. The tag bit is referred to as the ‘129th bit’. It is stored in a separate metadata block.

Permissions: encode the allowed operations for the pointer, such as read, write, execute, load capability, store capability, etc. This field is crucial for enforcing fine-grained access control and ensuring that capabilities can only be used in explicitly permitted ways.

The Global and Executive bits limit how the capability can be used [15].

Executive: controls which register bank is used when accessing specific registers, such as the CSP (Capability Stack Pointer) and DDC (Default Data Capability). It further enhances security by preventing unintended or unauthorized code execution.

Global: is a flag within a capability that determines whether the capability can be stored across different protection domains.

ObjectType: is used to determine if a capability is sealed and how it is sealed. The seal of a capability prevents it from being further dereferenced or modified until the corresponding key unseals it.

Bounds: it specifies the range of memory addresses a capability can access. The hardware will trap any attempt to access memory outside these bounds, preventing potential buffer overflows or other memory violations.

Address: holds the address of a pointer.
Flags: store unrestricted user data.

2.2.2. Fine-grain Protection

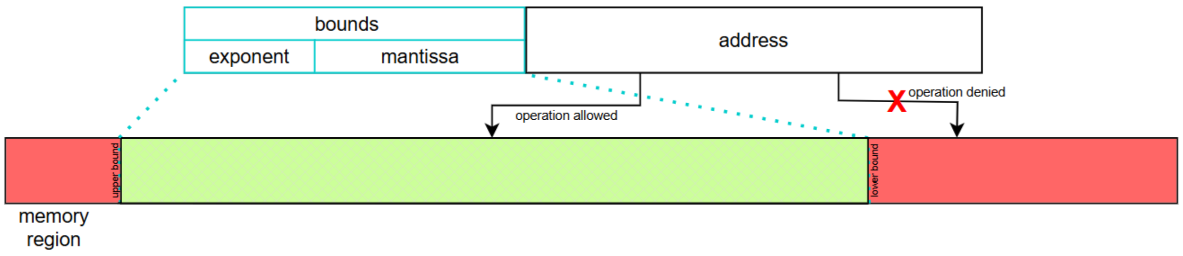


Figure 2.2.: Boundary representation. Green represents the valid memory region, while red represents the invalid scope.

In the CHERI architecture, fine-grained memory protection is achieved by combining the boundary information with the associated permissions. The boundaries are determined using a combination of a base address and a length. CHERI uses an encoding scheme involving an exponent and a mantissa to represent the bounds efficiently. The exponent determines the granularity of the memory region, while the mantissa specifies the exact size within that granularity. The bounds must be aligned and rounded to the nearest power of two, facilitating efficient bounds at the hardware level. Fig.2.2 visually represents the boundary system.

The fine-grained control over memory access is further supported by permissions, which specify allowed operations (such as read, write, and execute) over the memory region's scope.

CHERI implements a multi-layer security check for memory access. First, it verifies if the accessed memory is within the permitted bounds of the pointer. Any attempt to reach beyond these limits triggers an immediate hardware exception, effectively stopping the operation and preventing common vulnerabilities like buffer overflows. If the memory region is valid, CHERI proceeds to the second layer of security, verifying the access permissions associated with the pointer. These permissions dictate the operations allowed (e.g., read, write, execute). Any violation of these permissions, such as attempting to write to a read-only location, again results in a hardware exception.

2.2.3. Vulnerabilities Addressed by CHERI

CHERI is an innovative architectural approach to enhance system security by addressing several critical vulnerabilities. CHERI addresses the following memory vulnerabilities:

OOB Writes & Reads: occurs when a program tries to access memory outside its allocated buffer. This can happen in two ways: writes, where the program attempts to write data beyond the buffer, and reads, where it tries to read data outside the buffer. Out-of-bounds writes can corrupt other data, crash the program, or even allow attackers to execute malicious code. Out-of-bounds reads can leak sensitive information or cause the program to behave unpredictably. CHERI addresses both these issues by associating bound information with memory addresses. Each memory location has a defined range from which it can be accessed. The hardware blocks any attempt to read or write outside those bounds. By preventing both out-of-bounds writes and reads, CHERI significantly enhances software security and makes it much harder for attackers to exploit common memory-related vulnerabilities.

Use-After-Free: once a memory region is freed, its corresponding entry in a shadow bitmap is cleared. This bitmap tracks the validity of memory regions, with each bit representing a specific region. Clearing the bit effectively marks the region as freed. Cornucopia [17] uses a revocation mechanism that periodically scans the system, which uses the shadow bitmap to identify any capabilities that still point to freed memory regions Once identified, these capabilities are invalidated by clearing their validity bits. This makes them unusable, preventing further access to the freed memory and thus preventing use-after-free exploits. This prevents use-after-free types of attacks.

Pointer Forgery: CHERI prevents the exploitation of pointer metadata by ensuring that the derived capabilities are not expandable. When generating new capabilities from existing ones, the resulting permissions are restricted to the intersection of the original and newly applied permissions. Furthermore, CHERI prohibits direct modification of capability metadata. Any alteration to the metadata can only be achieved through authorized CHERI-specific hardware instructions, such as `cheri_setbounds`. CHERI also prevents forging pointers in memory by invalidating the capability tag when writing non-capabilities to memory.

The frequency and impact of memory-based vulnerabilities determine the importance of these protections. Attacks exploiting memory corruption flaws could lead to significant damage. For instance, the WannaCry [18] ransomware attack of 2017 leveraged a memory corruption vulnerability known as EternalBlue to spread rapidly across networks, encrypting files and demanding ransom payments, resulting in a devastating financial impact globally.

Developing a memory attack prevention mechanism that is robust in its security and efficient in its performance is essential in computer science.

2.3. Related Work

Various mechanisms were developed to enhance the memory safety of programs. These mechanisms are both at the hardware and software levels.

2.3.1. DBMS specific mechanisms

To the best of our knowledge, although various software and hardware hardening techniques for memory safety have been developed, none focuses explicitly on the needs of database systems.

2.3.2. Software-based memory safety solutions

Over the past few decades, numerous software-based solutions have been developed to address memory safety issues.

Traditional Defense Mechanisms

As highlighted in Szekeres's survey [19], various defense mechanisms were introduced at the software level to enhance memory safety. These include Stack Cookies, which aim to detect buffer overflows; Data Execution Prevention (DEP) [20], which blocks code execution in data regions; and Address Space Layout Randomization (ASLR) [2], which randomizes memory locations to hinder attackers.

These mitigations have proven insufficient against modern attack strategies. Sophisticated techniques like Return-Oriented Programming (ROP), where attackers chain existing code snippets to achieve malicious goals, and Just-In-Time (JIT) compilation, which can be exploited to execute arbitrary code, have emerged to bypass these traditional defenses.

To counter these evolving attack vectors, researchers have proposed new defense mechanisms. However, their widespread adoption has been hindered by various limitations. These include performance degradation, incompatibility issues across different systems and software, and increased program complexity. The Szekers et al. study [20] further analyzes the performance implications of these new mitigation mechanisms and the reason preventing their wide-range adoption. For example, control-flow integrity (CFI) [21] is a proposed mechanism for addressing ROP attacks. Integrating CFI can incur significant performance penalties, ranging from a few percent to over 20%, depending on the specific implementation and workload.

AddressSanitizer and Valgrind

AddressSanitizer, a feature incorporated into widely used compilers like GCC and Clang, offers a rapid and effective method for detecting memory errors during compilation. Instrumenting the code can identify issues such as buffer overflows, where data is written beyond allocated boundaries, and use-after-free errors, which occur when memory is accessed after deallocation.

AddressSanitizer may not detect all memory errors in situations with complex memory operations.

Valgrind is a debugging tool that helps developers identify and diagnose memory-related issues in their programs. It operates by running the program within a specialized virtual environment, allowing it to track memory operations and detect problems like memory leaks, where allocated memory is not released correctly, and invalid memory access, such as reading or writing to memory locations that are out of bounds or have been freed.

A significant limitation of Valgrind is its reliance on dynamic analysis. This means that if your test suite doesn't exercise all possible code paths, some memory errors may go undetected. Essentially, Valgrind can only identify errors in the code that is actually executed during the testing process.

2.3.3. Hardware-based memory safety solutions

Given the limitations of software-only solutions, hardware-based approaches were developed to ensure memory safety. These solutions are more robust by integrating memory protection mechanisms directly into the processor architecture.

Memory Tagging Extensions (MTE) [22]

MTE [22] works by adding metadata tags to memory addresses and pointers. When a memory access occurs, the hardware checks if the tags match. If they don't, it indicates a potential memory error, such as a buffer overflow or use-after-free vulnerability.

While MTE represents a step towards enhanced memory safety, CHERI addresses some of its inherent limitations. Firstly, MTE typically operates with a fixed tag granularity, which may not detect errors within smaller memory regions. CHERI, however, allows for much finer granularity, providing precise control over individual memory locations. Secondly, MTE primarily focuses on detecting memory errors rather than preventing them. MTE relies on probabilistic memory safety. Tag collisions can occur, especially in long-running applications, allowing some errors to slip through. Also, while MTE can detect out-of-bounds accesses, it doesn't prevent them from happening, potentially leading to unintended side effects before the error is caught. In contrast, CHERI enforces memory safety by proactively blocking unauthorized access.

Hardware-Assisted Address Sanitization (HWASan) [23]

HWASan [23] is another hardware-based approach to memory error detection. It leverages specialized hardware features, like Address Tagging in Android ARMv9 processors, to efficiently identify memory corruption bugs during program execution. This approach significantly reduces the performance overhead compared to software-only AddressSanitizer, making it more suitable for production environments.

However, HWASan primarily focuses on detecting errors rather than preventing them. CHERI, with its capability-based access control, goes a step further by proactively preventing unauthorized memory accesses, providing a more robust security guarantee, and eliminating the need for post-error recovery.

2.3.4. Intel Memory Protection Extensions (MPX) [24]

Intel MPX [24] aimed to prevent common vulnerabilities like buffer overflows by enforcing limits on where a program could write data.

However, MPX faced performance overhead and software compatibility challenges, leading to its discontinuation. CHERI offers a more promising solution for hardware-enforced memory safety with its more efficient design and tighter integration with the processor architecture. It provides comprehensive protection without the significant performance penalties and compatibility issues that hampered MPX.

2.3.5. Memory-safe programming languages

Memory-safe programming languages embed safety mechanisms, eliminating or significantly mitigating risks associated with memory-related exploits. They achieve this primarily through memory management (garbage collection), bounds checking, and robust typing systems with potential ownership models. Famous examples include Rust, renowned for its ownership and borrowing system; Java and C#, employing garbage collection; and Go, incorporating garbage collection and compile-time bounds checking.

However, memory-safe languages alone cannot eliminate memory safety concerns despite their advantages. Challenges remain with handling legacy codebases written in languages like C and C++, which are not inherently memory-safe. Furthermore, some memory-safe features can introduce performance trade-offs, potentially impacting performance-critical applications. Interactions with foreign function interfaces can also create vulnerabilities at language boundaries.

Therefore, while memory-safe languages are valuable for improving software security, they are not a complete solution. CHERI complements these languages by providing a robust, hardware-based foundation for memory safety, addressing the limitations and challenges faced by software-only approaches.

3. Motivation

Database systems are an essential part of modern computing, serving as the backbone for managing and organizing information.

DBMS rely on efficient data structures implemented in languages like C or C++ to achieve optimal performance and maintain low-level control over hardware resources. The reliance on low-level languages is prone to memory vulnerabilities, such as buffer overflows and use-after-free. These vulnerabilities are due to memory safety weaknesses within C and C++, including, but not limited to, the utilization of functions lacking bounds checking (e.g., strcpy, strcat) or the improper utilization of functions that do offer bounds checking (e.g., strncpy, snprintf); challenges in managing object lifetimes - for C++, potentially resulting in use-after-free vulnerabilities (accessing memory after deallocation) or exceeding allocated buffer boundaries. The Listing 3 presents such a vulnerability.

```c
typedef struct {
  uint64_t key;
  char value[100];
} hashtable_entry[SIZE_DB];

bool add_entry(simple_database *db, const char *key, const char *value) {
  for (auto i = 0; i < SIZE_DB; i++) {
    if (db->entries[i].key == 0) {
      db->entries[i].key = key;
      strcpy(db->entries[i].value, value); /* buffer overflow */
      return true;
    }
  }
  return false;
}
```

Listing 1: Vulnerable C code snippet with a potential buffer overflow in strcpy.

The relevance of memory issues is highlighted by the existence of newly discovered CVEs such as CVE-2022-41853 [25] - buffer overflow in PostgreSQL, which can enable remote code execution or CVE-2023-24538 [26] - use-after-free vulnerability in MySQL Server allowing an attacker to cause a denial of service (DoS) or potentially execute arbitrary code, remain a persistent threat to database systems.

The presence of vulnerabilities introduced using low-level language highlights the need for robust security measures to protect critical data and prevent exploitation. Traditional software-based solutions often introduce performance overhead or may not offer comprehensive protection. Given the performance-critical nature of database systems, a hardware-assisted approach like CHERI becomes attractive.

CHERI can enhance the security and resilience of database systems. It provides mechanisms to protect against common memory safety vulnerabilities, such as the previous CVE. Furthermore, CHERI's compartmentalization capabilities offer strong isolation between database components, limiting the impact of security breaches and enabling privilege separation. CHERI's hardware-assisted approach to memory safety has minimal performance overhead compared to traditional software-based techniques.

Compared with the other existing hardware solutions, CHERI introduces fine-grained memory protection, allowing precise control over memory access. Furthermore, CHERI's compartmentalization capabilities offer a powerful mechanism for isolating different software components, enhancing security by limiting the impact of potential breaches and enabling more flexible security architectures.

However, while CHERI appears promising for database security, it is crucial to acknowledge that evaluating its impact on database systems is still lacking. To our knowledge, this research is the first to address the implications of moving DBMS data structures on CHERI capabilities pointers systems. This thesis aims to address this gap by investigating CHERI's potential benefits and challenges in the context of database design, implementation, and performance implications.

3.1. Overview and Structure

This thesis explores the CHERI technology and its potential to enhance the memory security of high-performance data structures commonly used in database management systems (DBMS). This research was motivated by the persistent threat of database memory safety vulnerabilities.

We begin this thesis with an introduction to the core concepts of CHERI: capabilities pointers, Morello architecture, benefits, and limitations (further described in the Background section). We present the relevant technology and the security motivation for considering the adaption of CHERI capabilities in data structures used by DBMS.

Subsequently, the thesis explores how the selected data structures can be adapted to leverage CHERI capabilities. This section provides a guide (a recipe) to assist developers in navigating the complexities of porting these data structures to the CHERI architecture.

The discussion concerns challenges encountered while adapting data structures to the CHERI architecture. The challenges are categorized based on their functional implications. For each challenge, we provided information regarding why it arises within the context of the data structure, how it poses problems for CHERI capabilities, specific instances encountered during the research, and finally, the strategies used to overcome it. This systematic approach ensures a comprehensive understanding of the complexities of enabling CHERI capabilities on data structures.

The last chapter of the thesis presents the performance implication of the CHERI capabilities-enabled environment on the selected data structures. A custom framework was designed for this purpose. The framework enables a comparative analysis of the data structures with and without CHERI capabilities, providing valuable insights into the trade-offs between enhanced security and potential performance impacts.

4. Port to CHERI

This thesis outlines how data structures commonly found in database systems can be adapted to the CHERI capabilities pointer on Morello ARM architecture. Before porting to Morello, we need to adapt the data structures for AArch64. This is crucial because many components are currently designed for x86 architectures. For guidance on porting to AArch64, please refer to Chapter 5.

Porting applications to the CHERI extension has many challenges: compiler and linker options must be adjusted to accommodate CHERI's requirements; data structures often require modification to support the larger pointer sizes introduced by CHERI; pointer arithmetic necessitates careful handling, as the base address of capability pointers must be extracted before performing any calculations; managing provenance information, a key feature of CHERI, adds another layer of complexity to the development process; and finally, the increased size of pointers may lead to higher memory consumption, potentially requiring further code adaptations to optimize memory usage.

4.0.1. Compiler and Linker Options

The CHERI Morello architecture benefits from GNU and LLVM compiler implementations, which have been adapted to support the CHERI ABI. This thesis utilizes the `LLVM-Morello` compiler (clang-morello) for all code examples and experimental evaluations. `LLVM-Morello` is a modified version of `LLVM` engineered to support the features and requirements of the CHERI architecture, such as capability management, CHERI ABI support, and CHERI-specific optimizations.

The CHERI Morello architecture supports two ABIs: hybrid and pure capabilities. This thesis focuses on pure-capability systems, where all pointers are capabilities, to maximize the security benefits of CHERI.

The pure-capabilities ABI is enabled by passing `-mabi=purecap` to the compiler. The ABI requires the target platform to be further specified to the compiler by using `-march=morello` flag. The `march` option instructs the compiler to generate the program using a specific ISA convention for Morello CHERI ABI.

During the research phase, we encountered a linker-BUG where `LLVM-Morello` presented buggy behavior during the link-time optimization (LTO) phase. This can pose challenges when compiling projects for the CHERI Morello architecture, as observed with the Intel OneTBB library. This issue arises from the linker's inability to resolve references to CHERI capability ABI functions during the linking process when merging linker-optimized object files. LTO challenges can be solved using several strategies: completely disabling the LTO via `-fno-lto` compiler flag or using a less comprehensive LTO variant by specifying `-flto=thin` flag.

Software applications are often composed of multiple subcomponents that are compiled independently. When the *purecap* ABI is enabled, consistent compilation with the appropriate capabilities flag across all subcomponents is important to ensure the program functions correctly. The unified approach guarantees that all application parts respect the exact pointer representation and memory management model, preventing compatibility issues and unexpected behavior.

The data structures relying on C++ and standard libraries, such as cuckoo hash-table and Adaptive Radix Tree (ART), require minimal code modification to successfully port to CHERI capabilities Morello architecture. The commonly used libraries offer full support for the CHERI capabilities pointer. The changes involved updating the compiler and adjusting compiler options with the necessary CHERI-specific flags. This suggests that many existing C++ code bases that rely on standard libraries can readily adapt to CHERI with relatively minor adjustments. This ease of transition has the potential to accelerate the adoption of this promising architecture, enabling developers to leverage its enhanced security and reliability features without extensive code refactoring.

4.1. Base-Address of CHERI pointers

In the CHERI environment, metadata information, such as permissions and boundaries, is integrated within pointers, expanding their size from 64 bits to 128 + 1 bits. The CHERI pointer incorporates metadata information and the memory location's base address (as discussed in the Background section, Pointer Format 2.2.1).

Traditional architectures, such as those from Intel and AMD, permit direct arithmetic operations on pointers, treating them as numerical addresses. In contrast, the CHERI architecture introduces a fundamental shift by incorporating metadata within pointers. The CHERI pointers necessitate dedicated hardware support for pointer arithmetic. The hardware must correctly manage and preserve the metadata during these operations, ensuring the resulting pointer remains valid and secure. This architectural distinction introduces potential complexities when manipulating CHERI pointers. For instance, casting pointers to primitive data types or performing arithmetic requires careful handling of the associated metadata. The metadata must be appropriately propagated or intentionally discarded, depending on the specific operation and desired outcome. Failing to do so could lead to incorrect program behavior or compromise the security benefits provided by CHERI.

The arithmetic operations on the pointer's base address can be further classified into two categories: pointer address operations (sec. 4.1.1) and operations in which the pointer stores an integer value and is used interchangeably (sec. 4.1.2).

Accessing the raw address stored within a CHERI capability pointer requires explicit invocation of the `cheri_getaddress()` function, defined in the `cheri/cheric.h` library. This function extracts the base address from the capability pointer, discarding the associated metadata.

4.1.1. Pointer's Memory Address Operations

We classify the "Pointer's Memory Address" operations as those where the raw numerical address held by a pointer is directly used in computations rather than the pointer itself. In these operations, the CHERI capability metadata is intentionally disregarded, and only the underlying address is used for calculations.

Two primary categories were observed during the porting process: Value Initialization with Base Address and Hash Function Computations. A detailed explanation of each category is provided in the following subsections.

Value Initialization with Base Address

Data structures, primarily used in DBMS, rely on the pointer's base addresses to initialize seeds. The seed is initialized with the base address of a pointer.

For example, the seed for a random number generator is set using the memory address of a pointer. This technique is illustrated by the `FastRandom` function within the Intel Threading Building Blocks (TBB) library, found in the `oneTBB/src/tbb/misc.h` header file.

```
template <typename T>
void init( T seed ) {
    init(seed,int_to_type<sizeof(seed)>());
}

FastRandom( void* unique_ptr ) {
#ifdef __CHERI_PURE_CAPABILITY__
    init(cheri_getaddress(unique_ptr));
#else
    init(uintptr_t(unique_ptr));
#endif
}
```

The `FastRandom` constructor that takes a void* pointer as input. It initializes the seed using the pointer's base address. The `cheri_getaddress` extracts the base address from the capability.

Hash Function Computations

In data structures that rely on hash functions for efficient data retrieval (e.g., hash tables), it is crucial to ensure that the hash function operates on the base address of the pointer rather than the complete CHERI capability pointer. CHERI capabilities include metadata (permissions, bounds) that can change differ, even if the underlying memory location remains the same. If the hash function were to consider the entire capability, metadata changes could lead to different hash values for the exact memory location, disrupting the correctness of the data structure.

For example, the `get_address_waiter` (defined in `include/oneapi/tbb/detail/_utils.h`) returns the corresponding table entry for a given address. It employs a hash function `((tag >> 5) ^ tag) / num_address_waiters` to determine the corresponding slot.

```cpp
static address_waiter& get_address_waiter(void* address) {
#ifdef __CHERI_PURE_CAPABILITY__
    std::uint64_t  tag = cheri_getaddress(address);
#else
    std::uintptr_t tag = reinterpret_cast<std::uintptr_t>(address);
#endif
    return address_waiter_table[((tag >> 5) ^ tag) % num_address_waiters];
}
```

4.1.2. Primitive data is used interchangeably with pointer type

In specific scenarios, primitive data types and pointer types are sometimes used interchangeably. It involves utilizing pointer data types as temporary storage for raw data. This is useful when specific data needs to be reinterpreted as a pointer for subsequent operations.

For this scenario, there are two potential solutions: modifying the `state_type` definition to an integer type or using the `cheri_getaddress()` function to extract the raw numerical address and using it in further computations. This approach works only if the value won't be further used as a pointer.

During the porting phase, it was observed that the Intel TBB project has pointers and primitive data types that are freely interchanged. This is also present in Intel TBB's Spin Mutex implementation (`include/oneapi/tbb/spin_rw_mutex.h`), in which the `state_type` is declared as a `std::intptr_t` to facilitate the `std::atomic` operations. After fetching the atomic value, the pointer must be converted to an int type to apply the state transformations further.

For example, the function `try_lock`, in which the value stored in m_state base address field is periodically checked and updated.

```cpp
bool try_lock() {
    // for a writer: only possible to acquire if no active readers or writers
    state_type s = m_state.load(std::memory_order_relaxed);
#ifdef __CHERI_PURE_CAPABILITY__
    if (!(cheri_getaddress(s) & BUSY)) { // no readers, no writers; mask is 1..1101
#else
    if (!(s & BUSY)) { // no readers, no writers; mask is 1..1101
#endif
        if (m_state.compare_exchange_strong(s, WRITER)) {
            call_itt_notify(acquired, this);
            return true; // successfully stored writer flag
        }
    }
    return false;
}
```

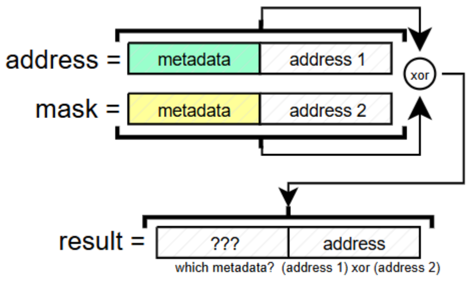
4.2. Source of provenance

Provenance in CHERI capability systems tracks a pointer's origin, ensuring its integrity and validity. It provides a record of the capability's creation and any subsequent transformations. However, when two CHERI capability pointers are involved in a bitwise operation, "source of provenance" ambiguity can arise regarding the provenance of the resulting pointer.

This ambiguity arises at the compiler level, as the compiler cannot definitively determine which pointer's metadata should be propagated to the result of the operation. Consequently, the system cannot guarantee the validity of the resulting pointer's metadata (as shown in Fig. 4.1), potentially leading to the result having the metadata of the unintended operand. The wrong metadata information can introduce security vulnerabilities or unexpected program behavior.

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

(a) The code showcases the problem with the source of provenance.



(b) A visual representation of the CHERI capabilities pointer for the given code.

Figure 4.1.: The source of provenance problem. The compiler does not know which metadata to further propagate to the result.

A typical scenario encountered during the porting process involved applying bitmasks to capability pointers (shown in Fig.4.1). Developers must pay attention when performing bitwise operations on CHERI capabilities, carefully considering the implications for provenance to maintain application integrity and security.

When performing bitwise operations on multiple pointers, it can be ambiguous which operand's capabilities should be inherited by the result. Bitwise operations on bitmasks are prevalent in data structures. They are used to fetch and set flags, check alignments, and perform bitwise manipulations on memory addresses or data within data structures.

The TBB library, when ported to the CHERI architecture, presented several instances where provenance information for capability pointers became ambiguous, particularly within the `task_proxy` class definition (`structures/oneTBB/src/tbb/mailbox.h`). This class utilizes a location_mask variable (`intptr_t`) for pointer manipulation, and methods like `is_shared`, `task_ptr`, and extract_task perform bitwise transformations on pointers. If not carefully managed, these operations can lead to uncertainty about the resulting pointer's provenance, potentially compromising security.

The challenge of maintaining clear provenance information on CHERI capability pointers has several solutions, such as using raw addresses and breaking the operation.

Using raw address

One practical approach to this challenge involves converting one of the operand pointers to its raw address representation using the `__ptraddr_t` type. This conversion effectively strips the pointer of its associated metadata, leaving only the integer value of the address. When the bitwise operation is performed, the resulting pointer inherits the provenance of the unconverted capability pointer.

The result is a new capability pointer with an unambiguous provenance derived from the original capability pointer that wasn't converted to a raw address. For example, the function:

```
void __TBB_EXPORTED_FUNC spawn(d1::task& t, d1::task_group_context& ctx, d1::slot_id id) {
    ...
    if ( id != d1::no_slot && id != tls->my_arena_index && id < a->my_num_slots) {
        ...
    #ifdef __CHERI_PURE_CAPABILITY__
        ptraddr_t pt_location_mask = static_cast<ptraddr_t>(task_proxy::location_mask);
        proxy->task_and_tag = intptr_t(&t) | pt_location_mask;
    #else
        proxy->task_and_tag = intptr_t(&t) | task_proxy::location_mask;
    #endif
        ...
    } else {
        spawn_and_notify(t, slot, a);
    }
#endif
```

Breaking the operation

The second approach involves breaking the operation into simpler operations with combined assignment expressions. The provenance of each intermediate result becomes clear by breaking down the complex operation into smaller, two-operand sub-operations. The use of combined assignment expressions ensures that the result of each sub-operation is stored back into one of the operands, effectively establishing a transparent chain of provenance. This step-by-step approach allows for easier tracing of the provenance flow, enhancing the overall understanding and maintainability of the code. To illustrate these approaches, consider the `task_ptr` function from the Intel TBB library:

```
static task* task_ptr ( uintptr_t tat ) {
#ifdef __CHERI_PURE_CAPABILITY__
    tat &= ~location_mask;
    return (task*) tat;
#else
    return (task*)(tat & ~location_mask);
#endif
}
```

4.3. Pointer Size Assumptions

A fundamental assumption in traditional architectures (like x86 and ARM) is that the ponter size matches 4 bytes for 32-bit systems and 8 bytes for 64-bit systems. This consistency has allowed developers to optimize data structures, such as hash tables, by carefully tuning parameters like the number of buckets per slot based on pointer size. These optimizations are often crucial for achieving optimal performance.

However, this assumption is no longer valid in CHERI architecture. CHERI capability pointers, containing embedded metadata (permissions, bounds), extend the pointer size to 128 + 1 bits. This size difference has two primary implications for software development:

1. The increased pointer size can disrupt existing code and data structures. This may necessitate modifications to code involving pointer arithmetic, type casting, or memory management, where the size of pointers is critical for memory allocation and layout.

2. CHERI introduces a size difference between `size_t`, typically used for object sizes and array indices, and `uintptr_t`, used to represent pointer addresses. While `size_t` remains the size of a standard integer (e.g., 64 bits), `uintptr_t` must accommodate the larger capability pointer size (128 + bits).

For example, the data structures and functions within CLHT and Intel TBB are optimized with the assumption of 64-bit pointers, a standard in traditional architectures. This assumption influences design choices related to memory layout, alignment, and padding, which are crucial for performance. However, when extended to accommodate CHERI (128 + 1)-bit capability pointers, these structure optimizations often become ineffective, potentially leading to performance degradation. This necessitates careful adaptation and re-evaluation of these structures to maintain efficiency in the CHERI environment.

4.3.1. Updating Structures

During the porting phase of the research, two cases were identified when the data structures had to be changed:

1. Structures are often padded to ensure proper alignment of their members. The CHERI increased pointer size, which breaks the alignment.

2. Structures are subjected to memory requirements. The larger pointers in CHERI increase the overall size of structures containing them, breaking the requirement.

Re-padding structures

The increase in pointer size necessitated re-evaluating and adjusting padding within data structures. This adjustment was crucial to maintain proper alignment and ensure efficient memory access in the CHERI environment.

For example, the **Cache-Line Hash Table (CLHT)** adopts a strategy of aligning its hash table entries with the size of cache-lines to minimize expensive cache misses and maximize cache efficiency on traditional architectures. Alignment is achieved by splitting the data into different sectors using padding. The entries are stored in the structure `clht_hashtable_t` (defined under `include/clht.h`).

However, the alignment is disrupted when ported to the CHERI capability pointer with its larger (128 + 1)-bit pointer size. The larger pointers size causes the structure to exceed its original two-cache-line. This leads to cache line splits, where elements previously contiguous within a cache line now span multiple lines. The fragmentation increases the number of memory accesses required to fetch relevant data, potentially degrading performance. The structure must be re-aligned to span three cache lines to maintain optimal performance and minimize cache misses.

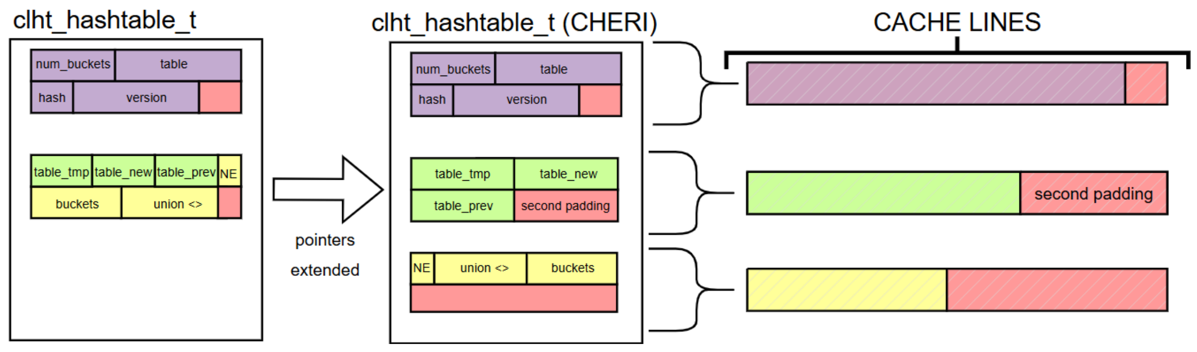


Figure 4.2.: Re-aligned CLHT padding over cache lines. With red representing the padding. The different colors (purple, green, and yellow) represent the newly created three sectors.

To maintain optimal cache line alignment with CHERI's larger capability pointers, the `clht_hashtable_t` structure is adjusted by adding a second padding. This padding effectively divides the structure into three 128-byte sectors, as shown in Fig. 4.2, aligning with the cache line size on Morello ARM processors. The size of the second padding is calculated as $CACHE\_LINE\_SIZE - sizeof(uintptr\_t) * 3$. This calculation ensures that the structure's size is a multiple of the cache line size, maintaining optimal alignment and minimizing potential performance degradation due to cache misses.

This adaptation highlights the need to re-evaluate and adjust data structure design when transitioning to CHERI. While CHERI offers enhanced security with its capability pointers, the larger pointer size necessitates careful consideration of memory layout and alignment to maintain performance efficiency.

Re-sizing structures

Data structures are often carefully designed with specific sizes based on the fixed-size pointers (typically 64 or 32 bits) used in traditional computer architectures. However, CHERI capabilities introduce a challenge by adding metadata to pointers, effectively increasing their size. Since many data structures rely on pointers to connect and organize their elements, this increase in pointer size directly translates to a rise in the overall size of the data structure. Their increase in size may break the optimizations, leading to a big performance overhead when executed within the CHERI capabilities environment.

For example, the structure `cpu_ctl_env` (defined in `include/oneapi/tbb/detai/_machine.h`), is designed to store the FPU settings, limited to `64 bits`. This structure contains only one member: a pointer to the effective FPU settings. However, when operating on CHERI, this pointer expands to 128+1 bits, exceeding the 64-bit storage limit.

```
#if __CHERI_PURE_CAPABILITY__
    /* storage space */
    extern std::vector<fenv_t*> m_cpu_ctl_env;
    extern std::atomic<int>     m_cpu_ctl_index;
#endif

class cpu_ctl_env {
#if __CHERI_PURE_CAPABILITY__
    uint64_t index = 0;      /* the index where the pointer resides in the structure */
#else
    fenv_t *my_fenv_ptr;
#endif
public:
 ...
}
#endif
```

To address this issue, a separate storage space is allocated outside the original data structure. This storage space is dedicated to holding the larger CHERI pointers. Instead of storing the pointers directly within the data structure, their corresponding addresses are moved to this external storage space.

Within the data structure itself, the pointers are replaced with indexes. These indexes act as references to the location of the actual CHERI pointers within the external storage space.

The proposed solution, while addressing the compatibility issue with CHERI's larger pointers, comes with a performance cost. Introducing an extra layer of indirection effectively downgrades memory access efficiency. Accessing the actual data now requires two memory lookups: first, to fetch the pointer from the external storage using the index, and second, to dereference the pointer to reach the data. This can increase memory access latency and potentially reduce performance, especially in performance-critical applications.

4.3.2. Changing function definitions, updating types

The increase in pointer size associated with CHERI capabilities introduced further complexities in managing function definitions and calls. Adapting function definitions to accommodate these changes is crucial to ensure compatibility and maintain the correctness of program behavior in CHERI environments.

The compiler permits casting types to larger sizes without issuing warnings. This masks potential issues, especially when dealing with the pointers of CHERI capabilities. The increased size of CHERI pointers, coupled with their embedded metadata, necessitates careful consideration when performing type casts.

During the porting phase, it was observed that developers utilized pointers in multiple sections of the codebase as function arguments, capitalizing on their representation as one of the largest available data types. However, the introduction of CHERI capabilities necessitates careful consideration of function signatures. When capabilities are enabled, pointer sizes increase to 128 + 1 bits.

In most cases, for performance-oriented data structures, the procedures are implemented using low-level functions, such as `__builtin_clz` or `__builtin_clzl`. In a CHERI capabilities environment, where pointers might be 128 + 1 bits, directly passing a capability pointer to these functions could result in incorrect results or undefined behavior.

To solve this issue, careful modification of the function signature is necessary. This involves updating the function's parameter list to accurately reflect the types and sizes of the passed arguments.

For example, such a problem was observed in Intel TBB, the function `machine_log2()` (defined in `include/oneapi/tbb/detail/_machine.h`):

```
#ifdef __CHERI_PURE_CAPABILITY__
    static inline uint64_t  machine_log2(uint64_t x) {
#else
    static inline uintptr_t machine_log2(uintptr_t x) {
#endif
    // If P is a power of 2 and x<P, then (P-1)-x == (P-1) XOR x
    return (number_of_bits<decltype(x)>() - 1) ^ __builtin_clzl(x);
}
```

This code snippet utilizes built-in compiler functions within a CHERI environment, such as `__builtin_clzl` (count leading zeros).

The difference in size between standard integer types and CHERI capability pointers (128 + 1 bits) leads to unexpected results. The additional metadata embedded within CHERI pointers can interfere with operations designed for standard integer sizes, creating an undefined behavior.

5. Porting to ARM

Porting to the ARM architecture is necessary before adapting the data structures for the CHERI capabilities environment. This porting process was essential because the data structures were initially designed and optimized for the x86 architecture, leveraging its specific instruction set and memory model.

5.1. Synchronization Primitives

Synchronization algorithms are essential for managing concurrent access to shared resources and ensuring data consistency in multi-threaded environments. These algorithms often rely on platform-specific atomic operations and differ significantly between Intel/AMD (x86) and ARM architectures. Consequently, porting data structures that utilize synchronization algorithms to the ARM architecture necessitates translating these atomic operations. The implementation for all functions described below is in Appendix A.

5.1.1. TAS_U8

One such operation for lock-free synchronization is the "Test-and-Set" operation on an 8-bit value (TAS_U8). This atomic operation checks the value of an 8-bit memory location and, if it is zero, sets it to a non-zero value. The original value is returned, indicating whether the operation succeeded in setting the value. This is fundamental for building mutual exclusion primitives and other synchronization mechanisms in a lock-free manner.

5.2. Memory Access

The ARM architecture has a memory model that differs from traditional architectures like x86. This model allows for memory access ordering and behavior variations, which can influence performance and require careful consideration when dealing with concurrency and memory consistency.

5.2.1. Prefetching Data

For example, the SSMEM (used by CLHT) is designed to efficiently allocate a new memory region which is prefetched in the cache. To achieve this, SSMEM utilizes the **PREFETCHW(x)** macro (defined in `ssmem/include/utils.h`) which triggers a prefetch operation on the specified memory address.

```
#if defined(__x86_64__)
    #  define PREFETCHW(x)      asm volatile("prefetchw %0" :: "m" (*(unsigned long *)x))
    #  define PREFETCH(x)       asm volatile("prefetch %0" :: "m" (*(unsigned long *)x))
#elif defined(__arm__) || defined(__aarch64__)
    /* https://developer.arm.com/documentation/dui0489/i/CJADCFDC */
    #  define PREFETCHW(x)      __asm__ volatile("pldw %0" : "Q" (*(const volatile void**)(x)))
    #  define PREFETCH(x)       __asm__ volatile("pld %0"  : "Q" (*(const volatile void**)(x)))
#endif
```

The *pld/pldw* hints to the CPU that the data at the memory address will be needed soon. This allows the CPU to proactively load the data into its cache, potentially reducing memory latency when the data is accessed.

5.3. Concurrent Execution Hint Instructions

Thread parallelism is a common feature in data structures, enabling multiple execution flows to operate simultaneously over shared resources. To manage the concurrency and ensure data consistency, synchronization mechanisms are employed. These mechanisms often utilize custom, platform-specific instructions tailored to the underlying architecture.

Pause/Yield Instructions

These instructions hint to the processor that the current thread is in a spin-wait loop or a similar non-productive state, allowing the processor to optimize its behavior and potentially improve overall system efficiency.

For example, this type of instruction is used by CLHT Lock-Base implementation.

```
#define _mm_pause() __asm__ __volatile__("yield")
```

The macro, *_mm_pause()*, utilizes inline assembly to insert a CPU "pause" instruction, specifically the "yield" instruction, into the code.

Memory Barrier Instructions

To enhance performance, compilers and processors can reorder memory operations. If not carefully managed, this reordering can sometimes lead to unexpected behavior and data inconsistency. Memory barriers enforce ordering constraints on memory operations and ensure data consistency across multiple threads.

For example, the CLHT library utilizes memory barriers to ensure data consistency and proper synchronization across its multi-threaded implementation. The `_mm_mfence()` macro, defined in `external/include/utils.h`, provides a convenient way to insert a full system memory barrier into code.

```
static inline
void arch_atomic_full_barrier(void) {
    /* https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/
    /* memory-access-ordering-part-3---memory-access-ordering-in-the-arm-architecture */
    __asm__ __volatile__ (
        "dmb SY \n\t"  /*  full system barrier   */
    );
}
#  define _mm_mfence()  arch_atomic_full_barrier()
```

The *dmb* instructs the processor to enforce an ordering constraint on memory accesses. The *SY* instruction specifies the scope of the memory barrier as "full system," meaning it affects all processors and I/O devices in the system.

5.4. Update of Built-In Functions

Porting software from traditional architectures to ARM requires the adaption of built-in functions. These functions, often used for performance optimization, are architecture-specific and need careful adaptation to ensure efficient execution on ARM. These modifications were observed across multiple libraries.

6. Evaluation

This chapter presents the evaluation results of the performance implications of enabling CHERI capabilities on the ported data structures.

6.1. Environmental Setup

The benchmarks were conducted on ARM64 Research Morello SoC processor. When writing this thesis, this processor is only an ARM64 processor supporting CHERI capabilities pointers. Table 6.1 further details the processor configuration.

Feature	**Description**
Architecture	Armv8.2-A with CHERI extensions
Cores	4 Cores, grouped in 2 CPU clusters
Cache Line Size	64 bytes
Memory Protection	CHERI capabilities with permissions and bounds

Table 6.1.: The configuration of ARM64 Research Morello Soc processor.

6.2. Framework & Testing Setup

A custom testing framework was developed to facilitate an automated evaluation of CHERI's impact on data structure performance. This framework integrates custom-designed microbenchmarks and standardized workloads, such as Yahoo! Cloud Serving Benchmark (YCSB). The architecture and in-depth technical details are presented in the Appendix **??**.

6.2.1. Testing Setup

All microbenchmarks and workloads were repeated five times to ensure reliability and minimize the impact of random variations. The results presented represent the average values obtained across these repetitions.

Microbenchmark

The number of inserted elements into the data structures is 100.000. The keys are values are represented as `uint64_t`.

The distribution of keys follows a Zipfian distribution for all tests. It describes a relationship where a few elements are accessed or used frequently while the vast majority are accessed far less often.

The evaluation process followed a consistent structure: the insert operation populated the data structure with the specified dataset; query operations were performed on the populated structure, followed by the removal operation.

The benchmark was conducted in two distinct modes: register-based and clock-based. All evaluated data structures utilized the register-based mode, except the Cuckoo hashtable, which used the clock-based mode due to specific implementation constraints.

YCSB

The YCSB workloads used in this study were executed with the following parameters: 100.000 initial insertions (initial elements) and 1 million operations. This approach ensured a standardized evaluation, leveraging widely recognized and representative workloads to assess the performance implications of CHERI capabilities on hashtable implementations.

6.3. Performance Overheads

This thesis presents an in-depth analysis of the performance implications of utilizing CHERI capabilities in the context of high-performance data structures commonly used within DBMS. The data structures analyzed were categorized into two primary groups: hashtables and trees.

6.3.1. Hashtables

The first group of data structures examined in this thesis comprises hashtables, fundamental building blocks in many database systems. This group includes four prominent hashtable implementations: CLHT (Lock-Base and Lock-Free), Cuckoo and oneTBB Concurrent Hashmap.

Microbenchmarks

To assess the performance implications of CHERI capabilities on hashtable, a series of microbenchmarks configurations were created to evaluate key performance characteristics relevant to hashtables used within database systems. These microbenchmarks targeted:

- **Insertion Benchmark**: evaluated performance under a high volume of insertions, with a load factor of 95% (with the hashtable size of 100.000 slots). This scenario simulates database systems handling continuous data ingestion and assesses potential bottlenecks in insertion latency.
- **Lookup Benchmark**: perform a large number of lookup operations, comprising 70% of the database content. Ensure that these lookups include an equal mix of 50% successful and 50% unsuccessful searches.

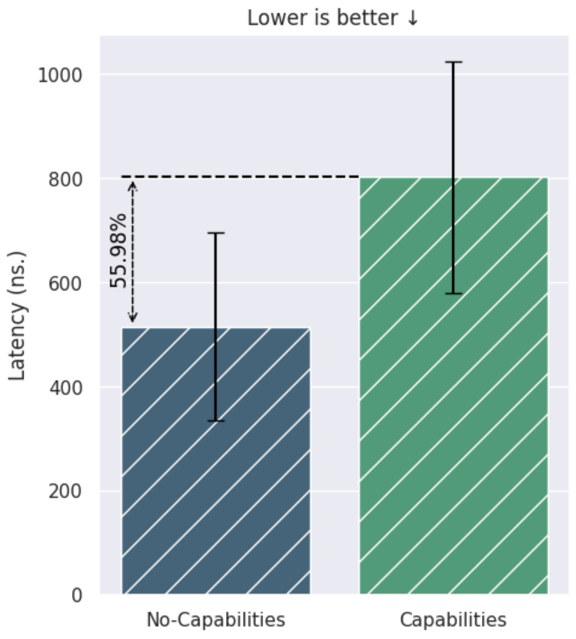
Insertion Benchmark
The observed performance impact of CHERI capabilities on hashtable operations (Fig.6.1) shows a variance across different concurrency control mechanisms. Lock-free implementations, like the CLHT Lock-free shown in Figure 6.1b, exhibit a performance overhead of around 30%. In contrast, lock-based hash tables face a significantly greater impact, averaging 90% and surpassing 130% in the case of the Cuckoo hash table, as illustrated in Figure 6.1c. This disparity can be attributed to the combination of CHERI's memory protection mechanisms and the inherent characteristics of each concurrency control strategy.

CLHT Lock-Based (Figure 6.1a) and oneTBB (Figure 6.1d) have similar performance overheads when utilizing CHERI capabilities, which could be attributed to their shared use of a *chaining* strategy for collision resolution. The shared design choice leads to inherent similarities. These similarities result in a comparable impact of CHERI's memory protection mechanisms on their performance.

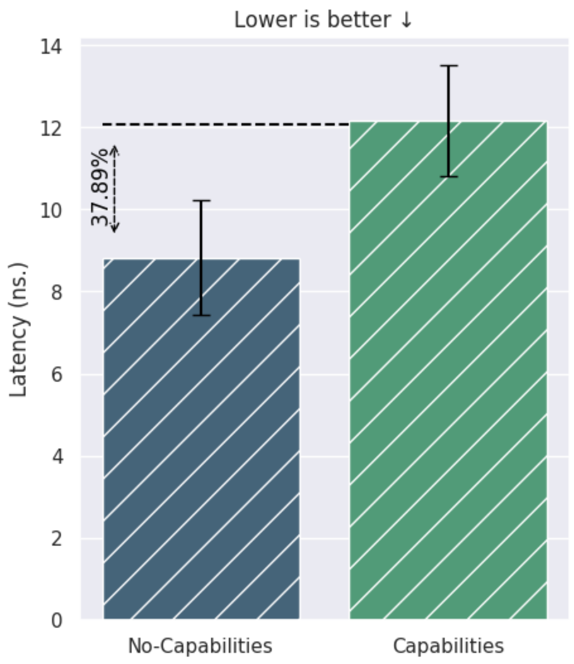
Cuckoo hashing utilizes *open addressing* insertion strategy and dynamic resizing to minimize collisions and maintain efficiency. *Open Address* leads to more memory accesses compared to the *chaining* strategy. This increased memory access necessitates additional checks by CHERI's memory protection mechanism, contributing to a more pronounced performance overhead. More details regarding the algorithms and memory accesses can be found in Appendix **??**.

Lookup Benchmark
The performance overhead observed for lookup operations under CHERI's memory protection mechanisms is consistent across all evaluated data structures, as shown in Fig.6.2. This suggests that the overheads associated with both the *open addressing* and *chaining* hash table strategies are comparable in the context of lookup operations. The performance overhead observed for lock-free mechanisms (CLHT Lock-Free Fig. 6.2b) is comparable to lock-based ones.

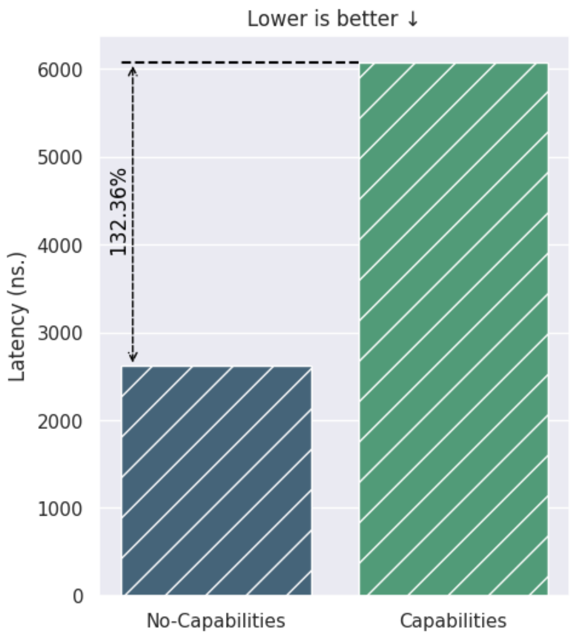
The results of the microbenchmarks highlight that the performance implications of CHERI capabilities are not uniform across all hash table operations and can vary depending on the specific operation and the underlying data structure. While insertions in *open addressing* hash tables may experience a greater impact due to the increased memory accesses involved in relocation and rehashing, lookups exhibit a comparable overhead across different collision resolution strategies.



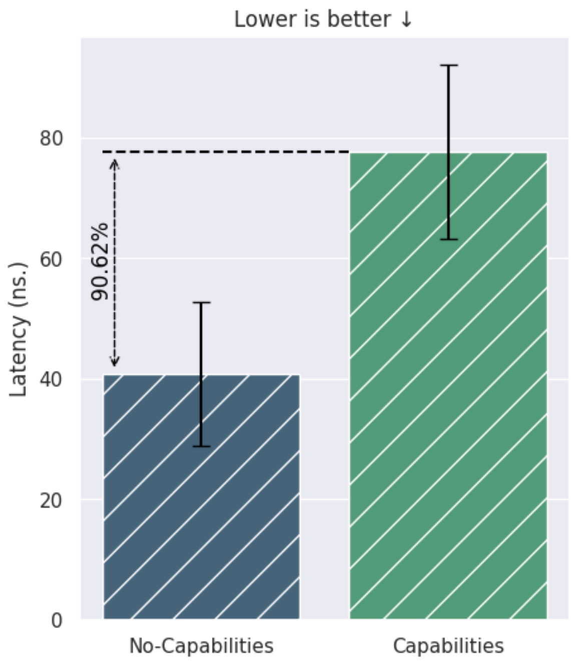
(a) CLHT Lock-Base



(b) CLHT Lock-Free

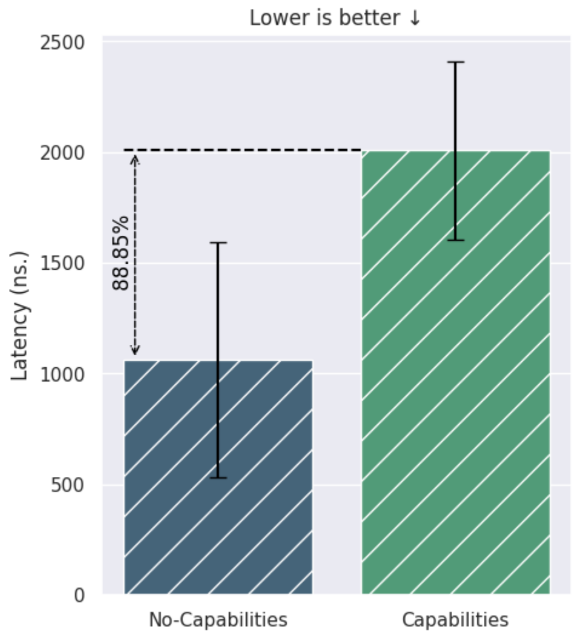


(c) Cuckoo (Libcuckoo)

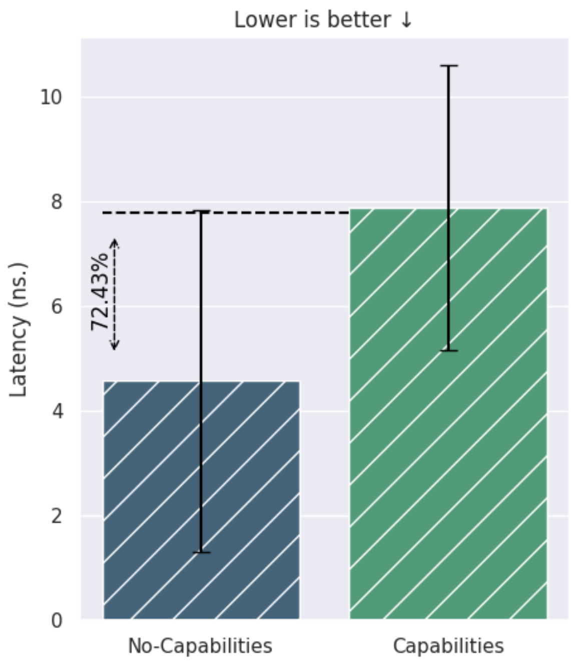


(d) Concurrent Hash Map from OneTBB

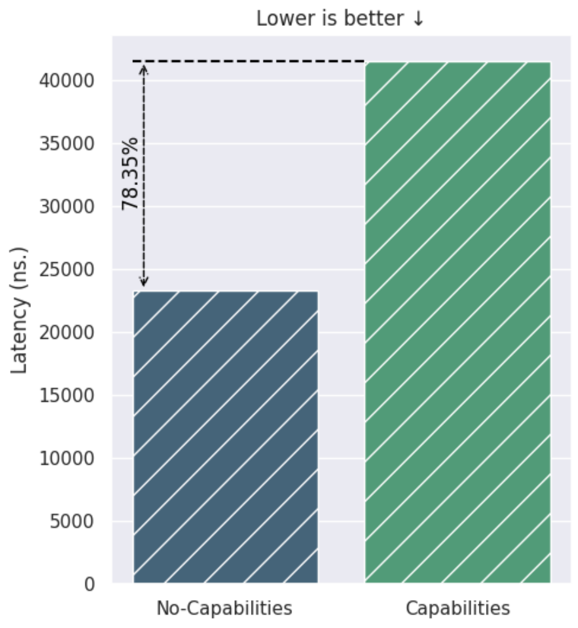
Figure 6.1.: Microbenchmarks performance comparison for "Insertion Benchmark".



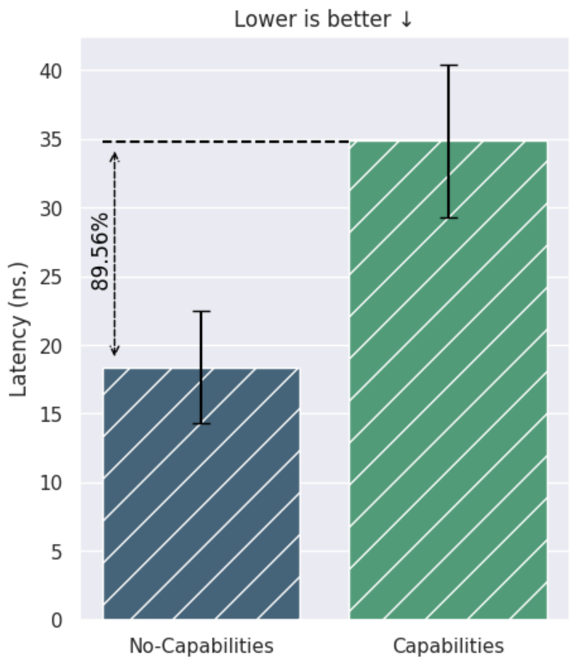
(a) CLHT Lock-Base



(b) CLHT Lock-Free



(c) Cuckoo (Libcuckoo)



(d) Concurrent Hash Map from OneTBB

Figure 6.2.: Microbenchmarks performance comparison for "Lookup Benchmark".

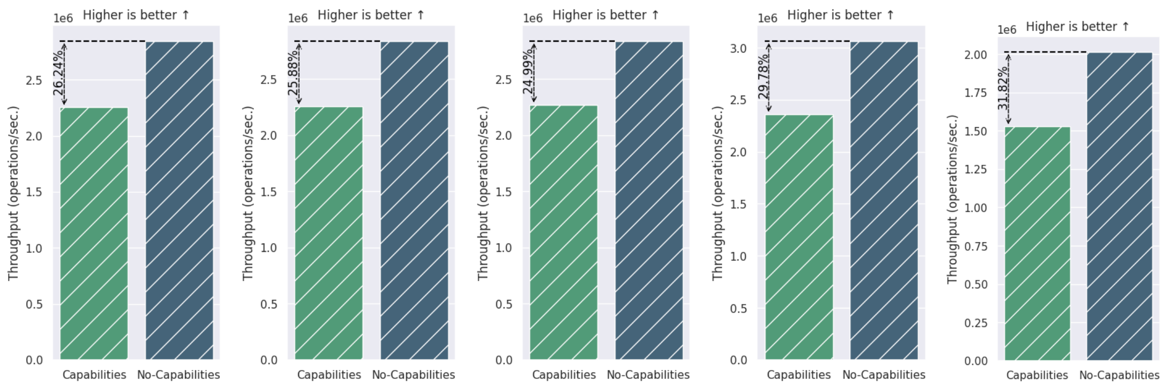
Yahoo! Cloud Serving Benchmark (YCSB)

YCSB framework is used to evaluate the performance implications of CHERI capabilities on hashtables within a realistic database environment.

For each hashtable implementation, the performance overhead measured by YCSB is similar on Workload-A, B, C, and D with the workload F showing a slightly more considerable overhead.

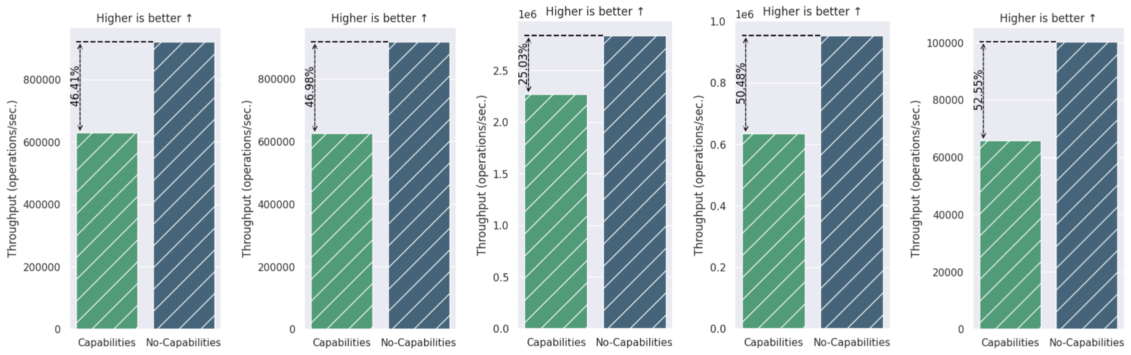
A distinction in throughput was observed between lock-free, CLHT Lock-Free (Fig.6.3) and lock-based hashtable implementations, with lock-free operations generally exhibiting small performance overhead.

For Lock-Based solutions, throughput varies between 35% on OneTBB Concurrent Hashtable (Fig.6.5) to $\tilde{4}$0% on CLHT Lock-Based (Fig.6.4) and 50% on Cuckoo Hashtable (Fig.6.5). The *open addressing* strategy generally under-performs compared to the *chaining* strategy, as previously discussed in 6.3.1. The standard deviation was not plotted because it was too small to be significant.



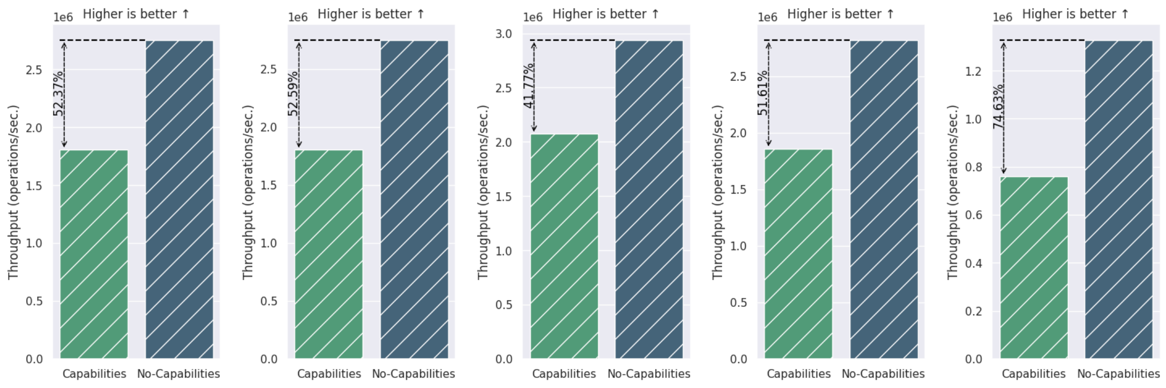
(a) Workload-A (b) Workload-B (c) Workload-C (d) Workload-D (e) Workload-F

Figure 6.3.: Throughput comparison YCSB for CLHT Lock-Free.



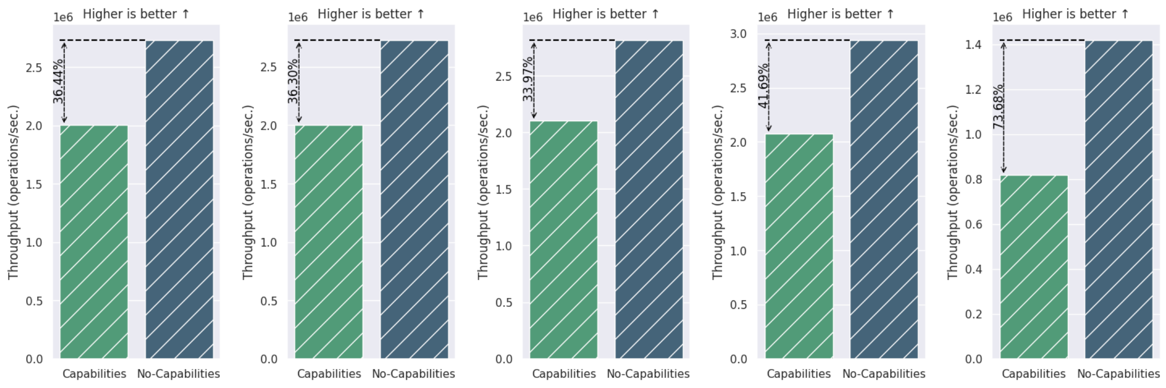
(a) Workload-A (b) Workload-B (c) Workload-C (d) Workload-D (e) Workload-F

Figure 6.4.: Throughput comparison YCSB for CLHT Lock-Based.



(a) Workload-A (b) Workload-B (c) Workload-C (d) Workload-D (e) Workload-F

Figure 6.5.: Throughput comparison YCSB for Cuckoo Hashtable.



(a) Workload-A (b) Workload-B (c) Workload-C (d) Workload-D (e) Workload-F

Figure 6.6.: Throughput comparison YCSB for oneTBB Concurrent Hashtable.

Performance Monitoring Counters

The performance monitoring counter (PMC) stats are captured to gain deeper insights into the performance implications of CHERI capabilities on hashtables. Table 6.3.1 presents a comparative analysis of PMC statistics, averaged across all threads, captured during the microbenchmark insert and query operations (described in 6.3.1).

For a comprehensive overview of the thread-specific performance monitoring counter data, please refer to Appendix B.1. This appendix provides a detailed breakdown of the collected metrics, allowing for a deeper analysis of the performance variations observed across different threads.

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

Table 6.2.: Table showing the average PMC stats for Cache Operations during obtained microbenchmarking.

A general trend observed across the evaluated hashtable implementations was increased cache utilization when operating under CHERI capabilities. This high cache usage can be attributed to the larger size of pointers in the CHERI environment, which directly impacts the memory of the data structures involved in hashtable operations.

The CLHT (Lock-Base and Lock-Free) hashtable implementation has the highest cache utilization among the evaluated data structures. This behavior can be attributed to CLHT's design, which optimizes for efficient storage within cache lines. However, when CHERI capabilities are enabled, the increased size of pointers results in an increased size of relevant data structures. The data structure grows from two to three cache lines, impacting performance and necessitating more frequent eviction and retrieval operations.

Memory Overhead

An analysis of hashtable memory usage was conducted to determine the overheads associated with CHERI capabilities compared to a standard execution environment. The results, shown in Fig.6.7, illustrate the memory usage overhead for the hashtable structure and the elements inserted into it. The experimental setup involved inserting 100,000 key-value pairs into the hashtables, each with a `uint64_t` key and `uint64_t` value. This configuration ensured a substantial dataset for evaluating memory consumption under a realistic load.

The similarity in memory overhead observed for CLHT Lock-Free and Lock-Based implementations can be attributed to their shared underlying structure for hashtables and buckets. Despite using different concurrency control mechanisms, both variants utilize similar data structures to store and organize elements within the hashtable.

The similarity in memory usage between oneTBB Concurrent-Hashtable and CLHT can be attributed to the usage of *chaining* strategy for resolving hash collisions. This strategy, which involves storing colliding elements in a linked list within the corresponding bucket, results in comparable memory consumption patterns for both implementations.

The Cuckoo hashtable shows lower memory utilization than chaining-based hashtables due to its *open-addressing* strategy. This strategy avoids the need for separate chaining, where colliding elements are stored in linked lists, resulting in a more compact memory layout.

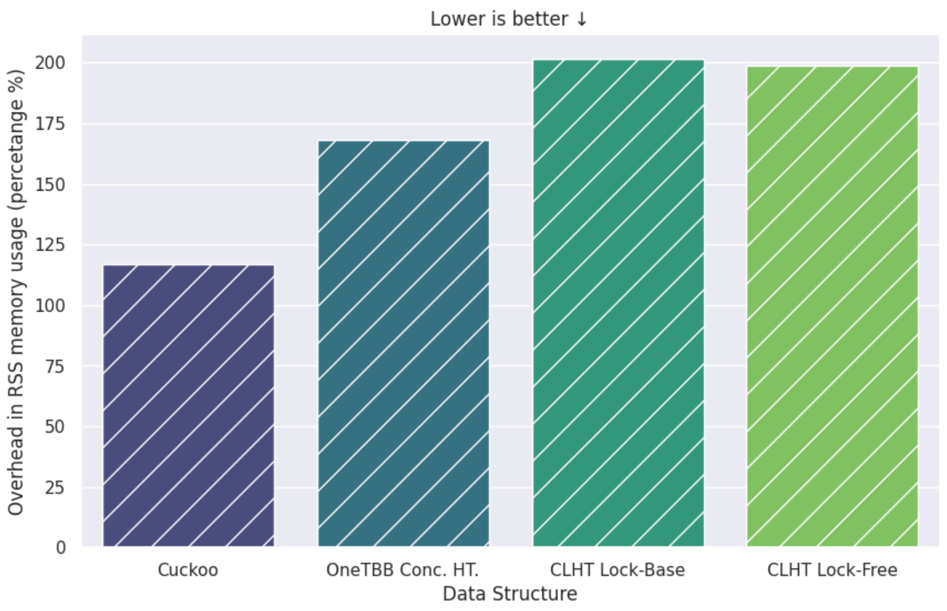


Figure 6.7.: Increase in RSS memory usage between CHERI capabilities and normal mode.

6.3.2. Trees

This study further examines the performance implications of CHERI capabilities on another data structure prevalent in DBMS: trees. Trees are data structures with hierarchical organization. They are efficiently designed for search properties and are commonly used for indexing, query processing, and maintaining sorted data within databases.

Microbenchmarks

A series of custom-tailored benchmark cases were created to further assess the performance implications of CHERI capabilities on tree structures. These microbenchmarks have the following configuration:

- **Insertion Benchmark**: the insertion performance is evaluated while inserting 100.000 elements into the data structure.
- **Lookup Benchmark**: perform 100.000 lookup operations with an equal mix of 50% successful and 50% unsuccessful searches.
- **Deletion Benchmark**: involves performing 100.000 deletion operations, comprising an equal mix of 50% successful deletions (where the key exists in the tree) and 50% unsuccessful deletions (where the key is not present).

Insertion Benchmark

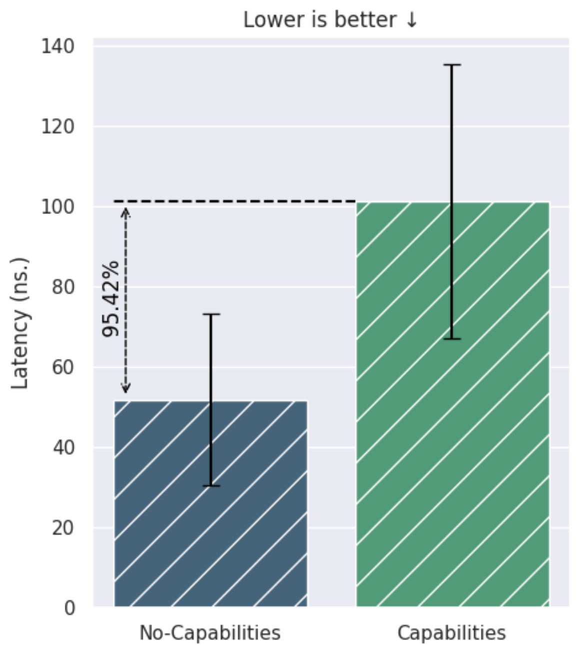
Figure 6.8 illustrates the performance overhead observed when executing tree-based data structures within the CHERI capabilities environment.

In contrast to the hashtable implementations (shown in 6.3.1), the tree-based structures have a more significant performance impact on insertion operation (Fig. 6.8) when operating under CHERI capabilities. This difference in performance can likely be attributed to the differences in the organization and access patterns of these data structures. Trees are hierarchical structures relying on pointers for navigation and may incur more significant overhead from CHERI's pointer manipulations and capability checks.

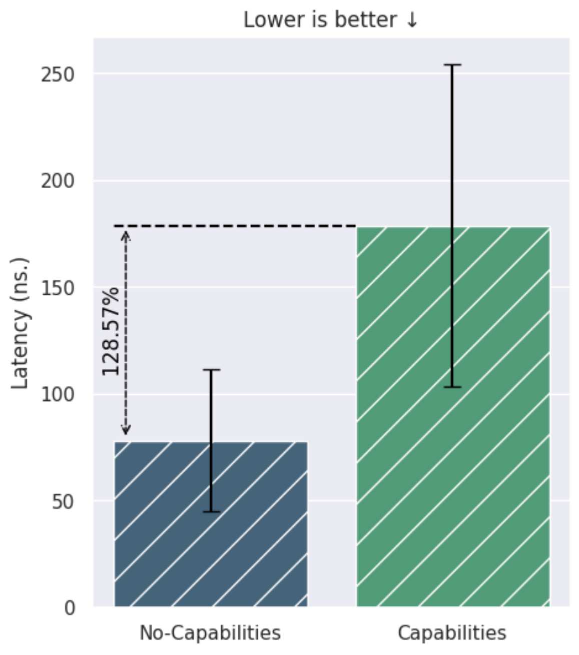
Lookup & Deletion Benchmark

The performance overhead for lookup (Fig.6.9) and deletion (Fig.6.10) operations exhibited similar trends, with a slightly more pronounced impact observed for deletions

The difference in performance overhead between B-tree and ART implementations under CHERI capabilities can be attributed to the distinctions in their structure and memory access patterns. B-trees often require multiple memory accesses to traverse the tree compared with ART.

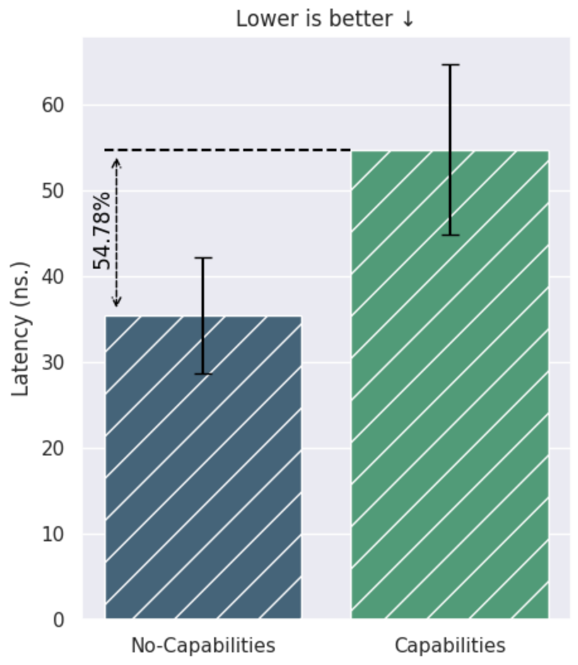


(a) Adaptive Radix Tree

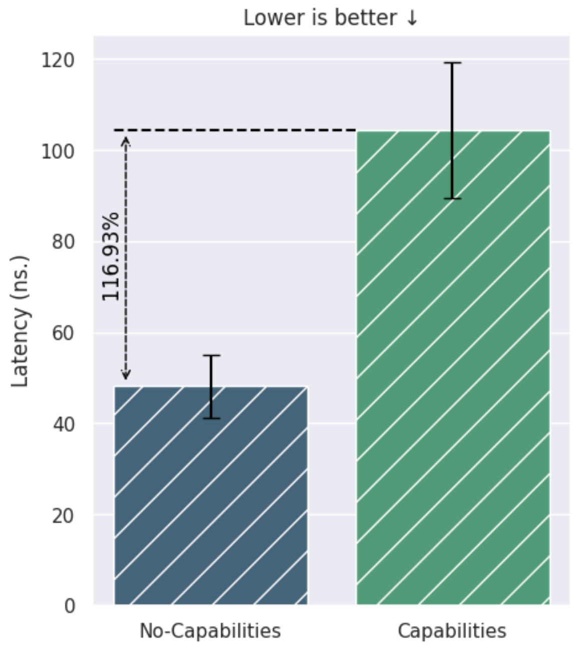


(b) Google BTree

Figure 6.8.: Microbenchmarks performance comparison for "Insertion Benchmark.

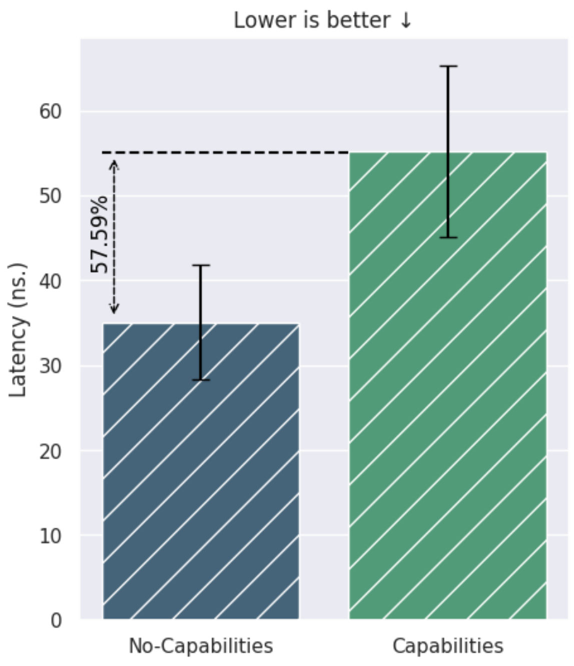


(a) Adaptive Radix Tree

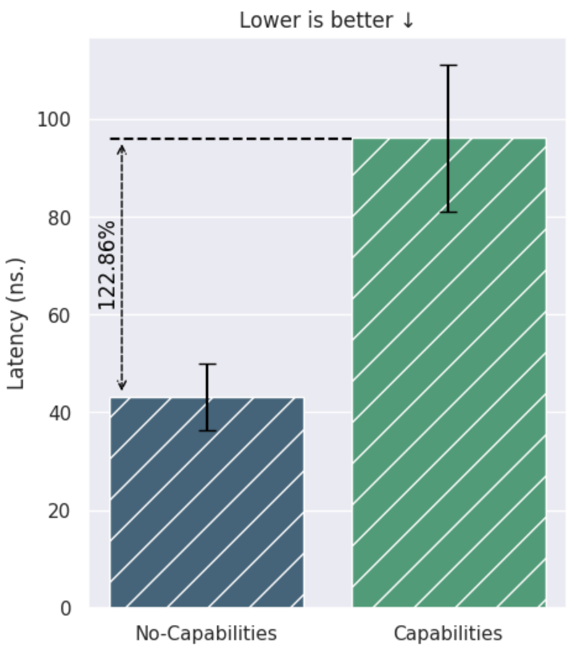


(b) Google BTree

Figure 6.9.: Microbenchmarks performance comparison for "Lookup Benchmark.



(a) Adaptive Radix Tree



(b) Google BTree

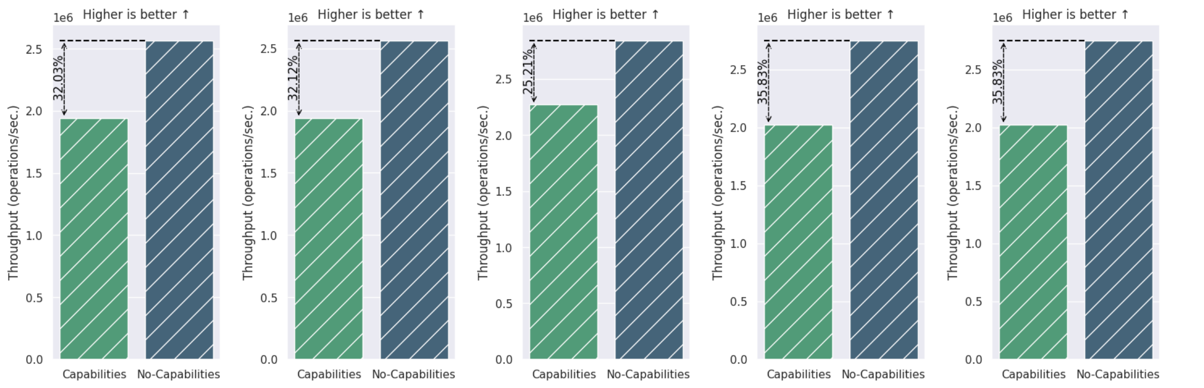
Figure 6.10.: Microbenchmarks performance comparison for "Deletion Benchmark.

Yahoo! Cloud Serving Benchmark (YCSB)

To further assess the performance of tree-based data structures within a realistic database environment, this study used the YCSB framework.

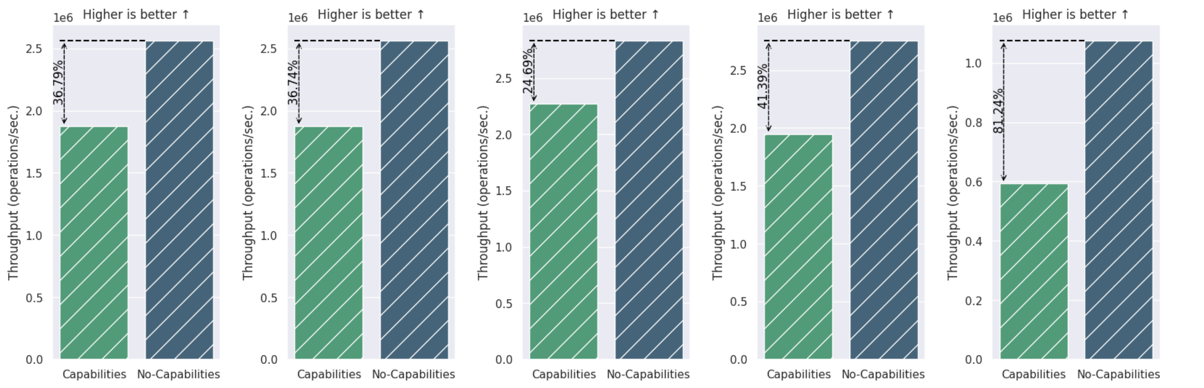
Our findings found a notable difference in performance overhead between these two structures. Specifically, ART (Fig. 6.11) has a performance overhead ranging from 25% to 30%, while B-trees (Fig. 6.12) show a wider range, varying between 25% to 80%.

The Adaptive Radix Tree uses a hierarchical, tree-like structure where each node can store multiple keys and pointers. This design reduces the number of memory accesses required compared to B-trees. Consequently, ART incurs a smaller performance overhead when operating within a CHERI capabilities-based security environment.



(a) Workload-A (b) Workload-B (c) Workload-C (d) Workload-D (e) Workload-F

Figure 6.11.: Throughput comparison YCSB for Adaptive Radix Tree (ART).



(a) Workload-A (b) Workload-B (c) Workload-C (d) Workload-D (e) Workload-F

Figure 6.12.: Throughput comparison YCSB for Google BTree.

Performance Monitoring Counters

The PMC stats are captured to obtain a deeper insight into the performance implications of CHERI capabilities for trees. Table 6.3.2 presents a overview of PMC statistics captured during the microbenchmark inserts and query operations (described in 6.3.2).a

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

Table 6.3.: Table showing the average PMC stats for Cache Operations during obtained microbenchmarking.

The Adaptive Radix Tree experiences a tenfold increase in cache-related operations when executing within a CHERI capabilities environment. This substantial increase in cache activity highlights the impact of CHERI's memory protection mechanisms on the ART's memory access patterns and performance.

The ART nodes are dense, packing in multiple keys and pointers. In typical architectures, this is great for caching. When a node is fetched into the cache, it brings along a lot of relevant data, reducing the need to fetch more data from main memory. However, CHERI capabilities add extra information to each pointer, making them larger. The larger pointers result in fewer of them fitting within each cache line, leading to more frequent data evictions from the cache. As the ART tree is traversed, needed data is constantly being pushed out of the cache, resulting in a surge of cache misses.

In the case of BTrees, nodes are wider, with fewer pointers per node and more. Also, traversals tend to be shorter due to the wider structure. Fewer nodes must be fetched into the cache, reducing eviction pressure. This lower density makes them less vulnerable to cache line limitations imposed by larger pointers.

B-trees have a CACHE advantage in CHERI environments due to their wider, structure and shorter traversals. While they still experience some impact from larger pointers, it's less pronounced compared to ART. This doesn't mean B-trees are always the optimal choice, but it highlights the importance of considering data structure characteristics when designing for CHERI-enabled systems.

Memory Overhead

ART and B-trees show different memory usage patterns (Fig. 6.3.2), with ART having a bigger memory overhead, with an increase of $\tilde{4}0\%$ compared to BTree implementation.

One reason for the bigger overhead in ART is the node density. Each node in an ART can hold many keys and pointers. Larger CHERI pointers mean each node consumes significantly more memory, leading to a bigger overhead. B-tree nodes have a lower density of keys and pointers.

ART nodes often have unused slots, especially when the tree is not fully populated or undergoes frequent modifications. This internal fragmentation contributes to wasted memory within each node. B-trees also experience some internal fragmentation, but it's generally less pronounced due to the smaller node size and how nodes are split and merged during updates. In a CHERI environment, the fragmentation in ART is exacerbated. The larger size of CHERI-enabled pointers amplifies the memory consumption of each node, further increasing the overall overhead.

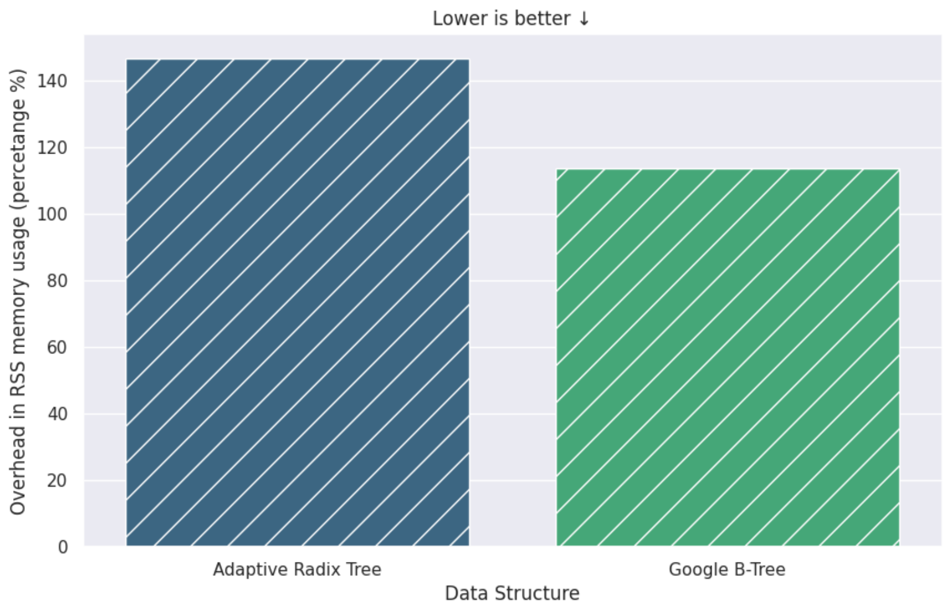


Figure 6.13.: Increase in RSS memory usage between CHERI capabilities and normal mode.

7. Conclusions

This report presents the process of adapting fundamental data structures, commonly used in database management systems, to function within the unique constraints and security enhancements offered by the CHERI capabilities environment. With its fine-grained memory protection capabilities, CHERI presents opportunities and challenges for database system developers.

By adapting and evaluating these core data structures, we have gained invaluable insights into the performance implications associated with increasing data structures' memory security through CHERI's memory protection mechanisms. Specifically, our investigation focused on the performance trade-offs when utilizing the Morello CHERI hardware platform.

To the best of our knowledge, this report presents the first comprehensive analysis of the performance implications of adopting CHERI capabilities within the context of database systems on the Morello platform. This research contributes to understanding how CHERI's advanced security features can be effectively used to enhance the security of database systems.

7.1. Future work

7.1.1. Optimization

An important next step involves developing mechanisms for optimizing data structures to minimize the performance overhead introduced by CHERI capabilities. By reducing the performance impact of the CHERI security capability environment, we can promote its integration into real-world database systems, where efficiency is important. This optimization requires a multi-step approach: careful analysis of existing bottlenecks within data structures, developing optimization techniques, and exploring new data structure designs tailored to the CHERI environment.

7.1.2. Advanced Compiler Support

To ensure both security and efficiency in systems that utilize capabilities for memory protection, advanced compiler optimizations are essential. These optimizations aim to automatically reduce the performance impact often associated with the additional checks and manipulations of capabilities.

The optimizations can be achieved through several strategies, such as identifying and eliminating redundant capability checks, which avoids unnecessary overhead. The compiler can also leverage the inherent properties of capabilities, such as bounds information, to perform additional optimizations like eliminating redundant array bounds checks. Finally, generating code specifically tailored for architectures that employ capabilities allows the compiler to take advantage of specialized instructions and hardware features, minimizing any performance overhead. By employing these techniques, compilers can effectively balance the robust security benefits of capabilities with the need for efficient program execution.

Abbreviations

ASLR	Address Space Layout Randomization
DEP	Data Execution Prevention
ROP	Return-Oriented Programming
CHERI	Capability Hardware Enhanced RISC Instructions
DBMS	Database Management Systems
CLHT	Concurrent Linear Hash Table
oneTBB	oneAPI Threading Building Blocks
ART	Adaptive Radix Tree
ISA	Instruction Set Architecture
JIT	Just-In-Time Compilation
CFI	Control-Flow Integrity
MTE	Memory Tagging Extensions
HWASan	Hardware-Assisted AddressSanitizer
MPX	Intel Memory Protection Extensions

A. ARM functions implementations

TAS_U8

The code implements a Test-and-Set (TAS) operation for an 8-bit value (uint8_t) on ARM processors using inline assembly. The core of the TAS operation lies within the do-while loop and utilizes two ARM instructions:

- **ldxrb**: instruction atomically loads the byte at the memory addressa into a register.
- **stxrb**: instruction attempts to store the value 1 (newval) into the memory address addr.

The instruction returns 0 if the store is successful (meaning the value at addr has not changed since the ldxrb instruction) and a non-zero value otherwise.

The loop repeats until the stxrb instruction succeeds, indicating that the TAS operation has acquired the lock (by successfully setting the value to 1).

```
tas_uint8(volatile uint8_t *addr) {
  uint32_t memval, newval=1, result;
  do {
      __asm__ __volatile__("ldxrb %w0, %1"        : "=&r" (memval)  : "m" (addr));
      __asm__ __volatile__("stxrb %w0, %w1, %2"   : "=&r" (result)  : "r" (newval),  "m" (addr));
  } while (result != 0);

  return memval;  /* extract byte */
}
```

B. PMC Stats

B.1. Hashtable: Cache Operations

Data Structure	No. Threads	PMC Stats			
		Stat Name	No Capability	Capability	Increase
CLHT Lock-Base	1	LL read	65.9 Mil.	198.0 Mil.	3.004x
		LL miss	65.3 Mil.	196.7 Mil.	3.012x
		L1 read	1190.0 Mil.	1286.9 Mil.	1.081x
		L1 write	551.0 Mil.	598.0 Mil.	1.085x
	2	LL read	95.5 Mil.	335.0 Mil.	3.507x
		LL miss	47.6 Mil.	238.4 Mil.	5.008x
		L1 read	2197.0 Mil.	1834.2 Mil.	0.834x
		L1 write	986.2 Mil.	854.3 Mil.	0.866x
	4	LL read	98.6 Mil.	432.6 Mil.	4.387x
		LL miss	49.8 Mil.	311.1 Mil.	6.246x
		L1 read	2196.5 Mil.	2310.9 Mil.	1.052x
		L1 write	986.0 Mil.	1109.7 Mil.	1.125x
CLHT Lock-Free	1	LL read	0.6 Mil.	3.1 Mil.	5.166x
		LL miss	0.6 Mil.	3.0 Mil.	5.0x
		L1 read	114.5 Mil.	1177.4 Mil.	10.282x
		L1 write	68.3 Mil.	901.9 Mil.	13.204x
	2	LL read	2.4 Mil.	3.2 Mil.	1.333x
		LL miss	2.3 Mil.	3.0 Mil.	1.304x
		L1 read	1066.2 Mil.	1176.2 Mil.	1.103x
		L1 write	778.7 Mil.	901.2 Mil.	1.157x
	4	LL read	2.4 Mil.	3.2 Mil.	1.333x
		LL miss	2.3 Mil.	3.1 Mil.	1.347x
		L1 read	1066.6 Mil.	1178.6 Mil.	1.105x
		L1 write	778.9 Mil.	903.4 Mil.	1.159x

Data Structure	No. Threads	PMC Stats			
		Stat Name	No Capability	Capability	Increase
Cuckoo Hashtable	1	LL read	3.5 Mil.	2.3 Mil.	0.657x
		LL miss	3.5 Mil.	2.3 Mil.	0.657x
		L1 read	3385.7 Mil.	3167.4 Mil.	0.935x
		L1 write	2495.9 Mil.	2929.5 Mil.	1.173x
	2	LL read	12.4 Mil.	17.3 Mil.	1.395x
		LL miss	3.2 Mil.	3.6 Mil.	1.125x
		L1 read	3433.6 Mil.	3902.6 Mil.	1.136x
		L1 write	2529.5 Mil.	3453.2 Mil.	1.365x
	4	LL read	71.5 Mil.	62.9 Mil.	0.879x
		LL miss	47.3 Mil.	41.8 Mil.	0.883x
		L1 read	7499.1 Mil.	7109.1 Mil.	0.947x
		L1 write	3380.0 Mil.	4183.9 Mil.	1.237x
OneTBB Conc. HT.	1	LL read	2.2 Mil.	2.7 Mil.	1.227x
		LL miss	2.2 Mil.	2.6 Mil.	1.181x
		L1 read	1160.5 Mil.	1284.6 Mil.	1.106x
		L1 write	867.9 Mil.	1002.8 Mil.	1.155x
	2	LL read	2.5 Mil.	3.0 Mil.	1.2x
		LL miss	2.1 Mil.	2.6 Mil.	1.238x
		L1 read	1159.6 Mil.	1283.4 Mil.	1.106x
		L1 write	867.4 Mil.	1001.2 Mil.	1.154x
	4	LL read	2.7 Mil.	3.1 Mil.	1.148x
		LL miss	2.4 Mil.	2.8 Mil.	1.166x
		L1 read	1160.8 Mil.	1286.2 Mil.	1.108x
		L1 write	868.7 Mil.	1004.0 Mil.	1.155x

List of Figures

List of Tables

Bibliography

[1] N. S. A. (NSA). *Software Memory Safety Report 2022*. https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/. 2022.

[2] PaX Team. *Address Space Layout Randomization*. http://pax.grsecurity.net/docs/aslr.txt. 2001.

[3] N. Bhakar, S. Bhansali, Y.-Y. Chen, R. DuVarney, R. Muth, and Y. Turner. "Data execution prevention". In: *Proceedings of the 10th ACM conference on Computer and communications security*. 2003, pp. 88–97.

[4] M. S. R. Center. *On the Effectiveness of DEP and ASLR*. 2010. URL: https://msrc.microsoft.com/blog/2010/12/on-the-effectiveness-of-dep-and-aslr/.

[5] T. David, R. Guerraoui, T. Che, and V. Trigonakis. *Designing ASCY-compliant Concurrent Search Data Structures*. Tech. rep. EPFL, 2014. URL: https://infoscience.epfl.ch/record/203822/export/mrc.

[6] R. Pagh and F. F. Rodler. "Cuckoo hashing". In: *Journal of Algorithms* 51.2 (2004), pp. 122–144.

[7] I. Corporation. *oneAPI Threading Building Blocks (oneTBB)*. Accessed: 2024-10-03. 2023. URL: https://github.com/oneapi-src/oneTBB.

[8] Google. *BTree - a simple, ordered, in-memory data structure for Go programs*. https://github.com/google/btree. Accessed: October 20, 2024. 2024.

[9] V. Leis, A. Kemper, and T. Neumann. "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 1945–1957. DOI: 10.1145/2463676.2463712. URL: https://db.in.tum.de/~leis/papers/ART.pdf.

[10] MITRE. *CWE/SANS Top 25 Most Dangerous Software Errors*. 2011. URL: http://cwe.mitre.org/top25.

[11] *Memory Unsafety in Apple's Operating Systems*. https://langui.sh/2019/07/23/apple-memory-safety/. Accessed: 2024-09-25. 2019.

[12] *We Need a Safer Systems Programming Language*. Accessed: 2024-09-25. Microsoft, July 2019. URL: https://blogs.microsoft.com/on-the-issues/2019/07/23/we-need-a-safer-systems-programming-language/.

[13] *Queue the Hardening Enhancements*. Accessed: 2024-09-25. Aug. 2023. URL: https://source.android.com/docs/security/test/memory-safety.

[14] Robert N. M. Watson, Peter G. Neumann, Simon W. Moore, Jonathan Woodruff, Michael Roe, Hongyan Xia, David Chisnall, Brooks Davis, Ben Laurie, Khilan Gudka, Peter Sewell, and Robert Norton. "CHERI ABI". In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA '19. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 554–567. ISBN: 9781450366736. DOI: `10.1145/3307650.3322236`. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-945.pdf`.

[15] A. Ltd. *Morello Prototype Architecture Overview User Guide*. https://developer.arm.com/documentation/d 2024.

[16] Arm Ltd. *Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture*. `https://developer.arm.com/documentation/ddi0606/latest`. 2023.

[17] R. N. M. Watson, P. G. Neumann, S. W. Moore, J. Woodruff, and M. Roe. "CHERI Heaps: Security and Performance". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 33–46.

[18] S. Mohurle and P. Patil. "WannaCry Ransomware Attack: Analysis and Countermeasures". In: *International Journal of Computer Science and Information Technologies* 8.4 (2017), pp. 605–609.

[19] L. Szekeres, M. Payer, T. Wei, and D. Song. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. IEEE, Apr. 2013, pp. 48–62.

[20] A. van de Ven and I. Molnar. *Exec shield*. 2004. URL: `http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`.

[21] N. Carlini and D. Wagner. "Control-flow Integrity". In: *2014 IEEE Symposium on Security and Privacy*. IEEE, May 2014, pp. 320–333.

[22] Arm Limited. "Armv8.5-A Memory Tagging Extension". In: *Arm Developer* (2022). URL: `https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf`.

[23] *Hardware-assisted AddressSanitizer*. `https://source.android.com/docs/security/test/hwasan`. Accessed: 2024-09-25. Android Open Source Project.

[24] *Intel MPX Explained*. `https://intel-mpx.github.io/index.html`. Accessed: 2024-10-09.

[25] *PostgreSQL Buffer Overflow Vulnerability*. `https://nvd.nist.gov/vuln/detail/CVE-2022-41853`. Accessed: October 14, 2024. Oct. 2022.

[26] *MySQL Server Use After Free Vulnerability*. `https://nvd.nist.gov/vuln/detail/CVE-2023-24538`. Accessed: October 14, 2024. Feb. 2023.