SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Microarchitectural Analysis of CHERI on the Morello Platform

Yude Jiang

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Microarchitectural Analysis of CHERI on the Morello Platform

Mikroarchitektonische Analyse von CHERI auf der Morello Platform

Author:	Yude Jiang
Examiner:	Prof. Pramod Bhatotia
Supervisor:	Martin Fink, Dr. Masanori Misono
Submission Date:	28.03.2025

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

Munich, 28.03.2025 Yude Jiang

Acknowledgments

I want to thank Prof. Dr. Pramod Bhatotia and the TUM Systems Research Group for allowing me to work on such an interesting and important topic. I thoroughly enjoyed being able to conduct hands-on experiments with innovative technology on prototype hardware.

I also want to sincerely thank my advisors, Martin Fink, and Dr. Masanori Misono. This thesis would not have been possible without their considerate, precise, and knowledgeable guidance.

Finally, I am deeply grateful for my friends and family, who gave me feedback and emotional support. Most notably, Viktoriia, Jonas, Lilly, and my brother Marco, without whom this thesis could not have been written.

Abstract

This thesis investigates the instruction-level performance characteristics of the Arm Morello processor, a hardware implementation of the Capability Hardware Enhanced RISC Instructions (CHERI) architecture. CHERI introduces hardware capabilities for memory safety with potential performance impacts.

Through micro-benchmarking, we analyze three aspects: The effect of capabilities on memory instruction performance, hardware bounds checking overhead, and capability manipulation instruction efficiency. Our findings show minimal overhead for load operations with capabilities, while store operations have reduced throughput with 128-bit capabilities compared to 64-bit pointers. We observe similar performance between capability and regular pointer-based memory access, indicating an efficient hardware bounds-checking implementation. Capability manipulation instructions show varying optimization levels, with some operations having high throughput and low latency while others show limited performance.

These results provide insights for software developers targeting CHERI architectures and for future hardware implementations, contributing to the general understanding of performance-security tradeoffs in memory-safe architectures.

Contents

1 Introduction

Computer security has become a critical concern in the modern digital age, with memory safety issues accounting for approximately 70% of the vulnerabilities disclosed by Microsoft and Google [MSR19; JA24]. These vulnerabilities, such as buffer overflows, use-after-free errors, and uninitialized memory accesses, often stem from conventional computer architectures and programming languages that prioritize performance over security.

Traditional approaches to addressing memory safety rely primarily on software-based solutions like Rust's ownership model, static analysis tools, and runtime checks. While valuable, these approaches often introduce performance overhead, lower coding productivity, or require extensive code rewrites. They also typically cannot address memory safety across the entire software stack, leaving critical components vulnerable.

Hardware-based memory protection mechanisms offer a promising alternative. By implementing memory safety at the architectural level, these solutions can provide stronger security guarantees with minimal performance impact. The Capability Hardware Enhanced RISC Instructions (CHERI) [Wat+19] architecture represents one of the most promising hardware-based approaches. CHERI extends conventional instruction sets with capabilities: Unforgeable tokens that combine memory addresses with bounds, permissions, and provenance information—to enforce fine-grained memory protection.

The Arm Morello program, a collaboration between Arm, the University of Cambridge, and other partners, has produced the first commercial-scale implementation of CHERI principles [ARM23]. The Morello processor incorporates CHERI capabilities into its architecture and provides a platform for evaluating capability-based memory protection in real-world scenarios.

While CHERI's security benefits have been well documented, its performance characteristics remain less explored, particularly at the instruction level. This understanding is crucial as it informs compiler optimizations, guides software development, identifies potential bottlenecks, and helps assess CHERI's viability for widespread adoption.

This thesis addresses this gap through a detailed instruction-level performance analysis of the Morello processor, focusing on memory operations and capability manipulations. We measure instruction latency and throughput through carefully designed microbenchmarks to characterize the performance implications of CHERI capabilities. Our analysis reveals critical insights into how capabilities affect memory access patterns,

bounds-checking implementations, and capability manipulation instructions.

The findings contribute to a deeper understanding of CHERI's performance characteristics and guide software developers, compiler engineers, and hardware designers. By identifying specific performance patterns and optimization opportunities, this work aims to facilitate more efficient software for CHERI-based systems and inform future architectural refinements.

In the following chapters, we provide background on CHERI and Morello, outline our research questions and methodology, present our findings, and discuss their implications. Finally, we compare our results with related work and offer conclusions and directions for future research.

2 Background

In this chapter, we describe the core principles of CHERI and how they are applied within the Morello prototype architecture. We explain key terminology and concepts which are utilized for the analysis presented in this thesis.

2.1 CHERI

Capability Hardware Enhanced RISC Instructions (CHERI) is a research project that originated from the University of Cambridge, aimed at enhancing the security features of traditional computer architectures.

On conventional Instruction-Set Architectures (ISA)s, the lack of distinction between pointer and integers, at least at the hardware level, where they are both usually represented as machine words, has been identified as one of the main causes of software vulnerabilities [MSR19; VZ; Pro]. As pointers can be manipulated using the same operations as integers, there are no fine-grained restrictions on which parts of memory they can reference. While operating systems provide some protection through virtual memory and the Memory Management Unit (MMU), this coarse-grained isolation at the process level is insufficient to prevent all exploits. This weakness can lead to data leaks and, in the worst case, to arbitrary code execution.

To mitigate this, CHERI introduces a new hardware datatype: the **Capability**. This data couples the traditional pointer with additional metadata, which not only serves to differentiate it from regular integer data, but also provides information on which parts of memory the pointer is allowed to access, what kind of Access is permitted (i.e. read, write, execute) if the capability itself is still valid. Moreover, the CHERI architecture enforces crucial security properties like provenance and monotonicity, which ensures that capabilities can only be derived from others in memory-safe ways.

To ease adoption and the conversion of existing codebases, the CHERI **Software Model** defines two different compilation modes: **pure-capability mode** and **hybrid mode**. In pure-capability mode, memory access is only possible via capabilities, while in hybrid mode, regular pointer access is still permitted, allowing software to adopt capabilities gradually by selectively replacing conventional pointers.

By restricting memory access to occur only via capabilities in pure-capability mode or selectively in hybrid mode, checking accesses against capability metadata, enforcing

capability provenance and monotonicity, CHERI can enforce fine-grained memory protection and compartmentalization across the entire software stack.

2.1.1 Capabilities

The foundation of CHERI is the introduction of capabilities. Capabilities are a new architectural primitive, meant to replace pointers as the only way for programs to address data in memory while at the same time introducing a new layer of hardware-level memory safety. This is achieved using their unique construction, which widens the conventional pointer to double its size + 1 bit, so to $64 + 1$ bits for 32-bit and $128 + 1$ bits for 64-bit architecture and utilizes the additional bits as special metadata.

A 128+1 bit CHERI capability, as used in Morello, consists of the following components (see Figure 2.1):

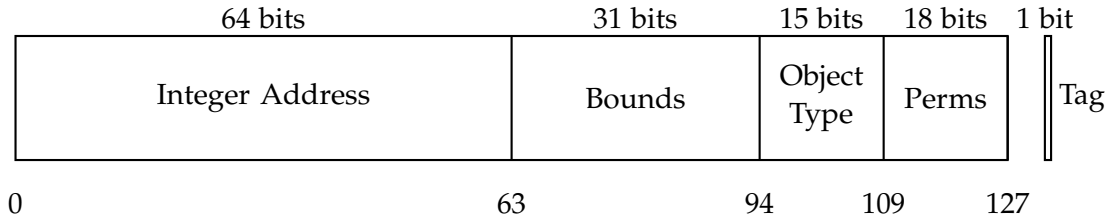

Figure 2.1: Structure of a 129-bit CHERI capability as implemented in Morello.

- **Integer address** (bits 0-63): A conventional 64-bit integer address.
- **Bounds** (bits 63-94): The lower and upper bounds describe which part of the address space can be accessed using the capability. As storing bounds in full 64-bit resolution would require capabilities to be $4\times$ the size of a pointer instead of double, which would be unfeasible due to performance and memory overhead concerns, CHERI uses a bespoke compression scheme called CHERI Concentrate. This compression scheme utilizes redundancies between the lower/upper bound addresses and the integer address. It represents the bounds using a floating-point format, allowing high-precision bounds checking for smaller allocations while still allowing efficient encoding of larger regions.
- **Object type** (bits 94-109): The Object type allows a capability to be "sealed", preventing it from being manipulated or dereferenced. This can be used for opaque pointers types and in conjunction with instructions such as `CCALL` to implement secure calling conventions and controlled invocation of privileged

code. This mechanism is crucial for enabling fine-grained compartmentalization and enforcing strict access control within the system.

- **Permissions** (bits 109-127): A mask controlling how the capability can be used. It can enforce the conventional properties of read/write/execute by restricting load-/store instructions and instruction fetch, and it further extends those properties by the ability to regulate the loading and storing capabilities.
- **Validity tag** (bit 128): A tag that tracks if the capability is valid. When a capability stored in memory is manipulated using non-capability-aware instructions, such as integer arithmetic or data copies, the validity tag is cleared, preventing the creation of capabilities outside the safety constraints imposed by CHERIs architectural rules. After a capability is invalidated, extracting data, such as the integer address, from its components is still possible, but using the capability for any memory access operation is prohibited. When a capability is stored in a register, it is stored at its full 129-bit size, which includes the validity tag. In contrast, when stored in memory, the tag is not stored inline but is instead in a separate hierarchical tag table retained in Dynamic Random-Access Memory (DRAM).

2.1.2 Architectural Rules

CHERI enforces several crucial security properties governing how capabilities can be manipulated and used.

Two core properties are enforced:

- **Provenance validity**: Valid capabilities can only be created through explicit capability-aware instructions and must be derived from other valid capabilities. This applies to capabilities both in registers and memory, preventing the creation of capabilities through arbitrary data manipulation.
- **Capability monotonicity**: When constructing new capabilities (except during sealed capability manipulation and exception handling), the resulting capability cannot have greater permissions or bounds than its source capability.

The capability system is initialized at boot time, where firmware receives initial capabilities that permit full address space access and instruction fetch. Memory tags are cleared, and subsequent capabilities are derived following the monotonicity principle as they propagate through the boot chain: firmware $\rightarrow$ boot loader $\rightarrow$ hypervisor $\rightarrow$ operating system $\rightarrow$ application. Each stage can only further restrict the bounds and permissions of derived capabilities. This architecture enables precise control over memory access intentions and prevents “confused deputy” problems, where a program

with elevated permissions is tricked into performing operations on behalf of an attacker, through careful capability delegation, providing the foundation for implementing memory-safe programs and enabling fine-grained compartmentalization.

2.1.3 Software Model

The CHERI software model is designed to facilitate the adoption of CHERI capabilities by offering two distinct compilation modes. These modes allow developers to choose between comprehensive fine-grained memory safety and only selective coarse-grained adoption based on their specific requirements.

The two modes are:

- **Pure-capability mode**: For the compiled software to benefit from the full fine-grained memory safety guarantees provided by CHERI, in this mode, every pointer type, including return addresses, stack pointer, etc., are replaced with capabilities. This changes the Application Binary Interface (ABI) significantly, as the size of pointers doubles, necessitating different memory layouts for data structures. Due to these significant changes to the ABI, code compiled in pure-capability mode cannot maintain binary compatibility with non-capability code. Therefore, it cannot be directly linked or interfered with.
- **Hybrid mode**: To ease code bridging between legacy and capability-enhanced software, the hybrid mode keeps pointers implemented as integers. Instead of being checked against included capability metadata as in pure-capability mode, or not at all as in legacy software, in hybrid code, the integer pointers are checked against and interpreted based on the global Default Data Capability (DDC), which is stored in a dedicated CPU register. This approach provides a basic level of memory safety and compartmentalization. For scenarios requiring more fine-grained safety or interoperability with pure-capability mode, capabilities can be introduced through language-specific annotations.

2.2 Morello

In collaboration with the University of Cambridge, the University of Edinburgh and Linaro, Arm developed the Morello architecture, an extension of Armv8-A meant to incorporate principles from the CHERI ISA [ARM]. This development culminated in the release of a hardware prototype in 2022 called the Morello System Development Platform (SDP), a board containing the Morello System on a Chip (SoC). This SoC is the first physical implementation of the Morello architecture based on the preexisting Arm

Neoverse N1, a multicore and out-of-order executing high-performance CPU designed for infrastructure applications.

Morello implements several key architectural characteristics:

- **Capability format**: Morello's capability representation is a slightly modified version of the CHERI Concentrate format
- **Merged register file**: Morello extends the Armv8-A register file with capability registers and preserves all existing architectural registers
- **Instruction encoding**: The architecture adds capability-aware variants of existing Arm instructions while maintaining backward compatibility
- **Exception handling**: Morello adapts CHERI's exception model to align with Arm's existing exception levels and privilege modes

3 Motivation

While enhanced security and correctness are the driving features of CHERIs and, by extension, Morellos development, more widespread adoption of the technology will more likely be achieved if concerns about its performance can be addressed adequately. To allow a more informed evaluation of the architecture's security performance tradeoff, more concrete data on CHERIs/Morellos performance characteristics needs to be obtained, particularly around memory access overheads and capability manipulation costs.

Most of the information that exists about CHERIs/Morellos performance has been derived from research that ports legacy software to the novel memory-safe architecture. This research tends to primarily focus on the correctness of the port. If performance is evaluated, the causes for overhead are rarely discussed or analyzed in detail, especially not at the instruction level.

Some of the early analyses published after the release of the Morello hardware prototype, heavily suggests that both inefficient compiler output [LJS23] and microarchitectural limitations [Wat+23] significantly impact the overall execution speed of capability-enhanced binaries. While these studies provide first insights into potential performance bottlenecks, more detailed performance metrics are needed. Specifically, understanding instruction latencies and throughput is crucial for improving compiler optimizations and instruction level micro-optimizations.

To this end, this thesis performs extensive performance analysis of Morellos Central Processing Unit (CPU) instructions using a microbenchmark framework. Through careful measurement of instruction timing characteristics like load/store throughput and capability manipulation latencies, we aim to provide deeper insights into microarchitectural behavior and instruction-level characteristics that can inform future software optimizations and compiler improvements.

4 Research Objectives

We determine three main research objectives, which we address in the following analysis.

4.1 RO1: Effect of Capabilities on Memory Instructions

We investigate how the increased size of the pointers, in particular when doing memory I/O with capabilities, affect the overall latency and throughput of the operations. The specific scaling between pointer size and throughput, when executing either load or store instructions might indicate the size of the processor's load and store buffers. This provides insight on the CPUs ability to pipeline and execute memory operations in parallel, unlocking potential micro-optimizations. Given that one of the primary performance concerns with the CHERI architecture is its increased memory footprint due to $128 + 1$-bit capabilities replacing 64-bit pointers, we investigate whether this size increase also directly impacts the performance of memory instructions.

4.2 RO2: Bounds Checking Overhead

Secondly, we want to assess the overhead incurred by performing bounds and validity checks on capabilities. These checks are performed for each memory instruction on CHERI. Therefore, even a small increase in latency could accumulate into a non-negligible performance hit in the context of a larger application. Furthermore, bounds checks are performed differently when accessing memory from a capability or a regular pointer in hybrid mode. As the former checks the memory address against its capability bounds and the latter against the bounds provided in the systems Default Data Capability (DDC), we also need to find out if there is a performance discrepancy between the two methods.

4.3 RO3: Latency and Throughput of Morello Instructions

Finally, we want to observe the performance metrics for instructions, which operate directly on capabilities, modifying their bounds or converting them to and from

regular 64-bit pointers. The latency and throughput of said operations provide useful information when evaluating performance costs, especially for optimizing software they are often used in, such as allocators, operating systems, and programs that partially adopt capabilities using hybrid mode. They can also indicate if there are special execution units for capability manipulation, how many of them there are, and how effectively those operations can be pipelined and executed in parallel.

5 Evaluation Setup

In this chapter, we describe the Hardware and Software setup used for this thesis.

5.1 Hardware Setup

The hardware setup is identical between all experiments. We test on the Arm Morello SoC, with the following specifications (also visualized in Figure 5.1):

- Two CPU clusters, each containing two out-of-order Armv8.2-A cores, all implementing CHERI, with a target clock frequency of 2.5 GHz [Wat].
- 64 KiB of L1d and L1i cache each per core, 256 KiB each in total.
- 1 MiB of L2 cache per core, 4 MiB in total.
- 1 MiB of L3 cache shared between the two cores in one cluster, 2 MiB in total.
- 8 MiB of L4 shared between all cores and clusters.
- 16 GB of external DDR4 RAM

5.2 Software Setup

We run all experiments on the same operating system setup:

- NixOS 24.11 (Vicuna)
- Morello fork version 1.8.1 of mainline Linux Kernel Version 6.7.0

The compilation configuration and runtime are differentiated into three distinct modes that represent varying levels of CHERI capability adoption:

- *NOCAP*
- *PURECAP*

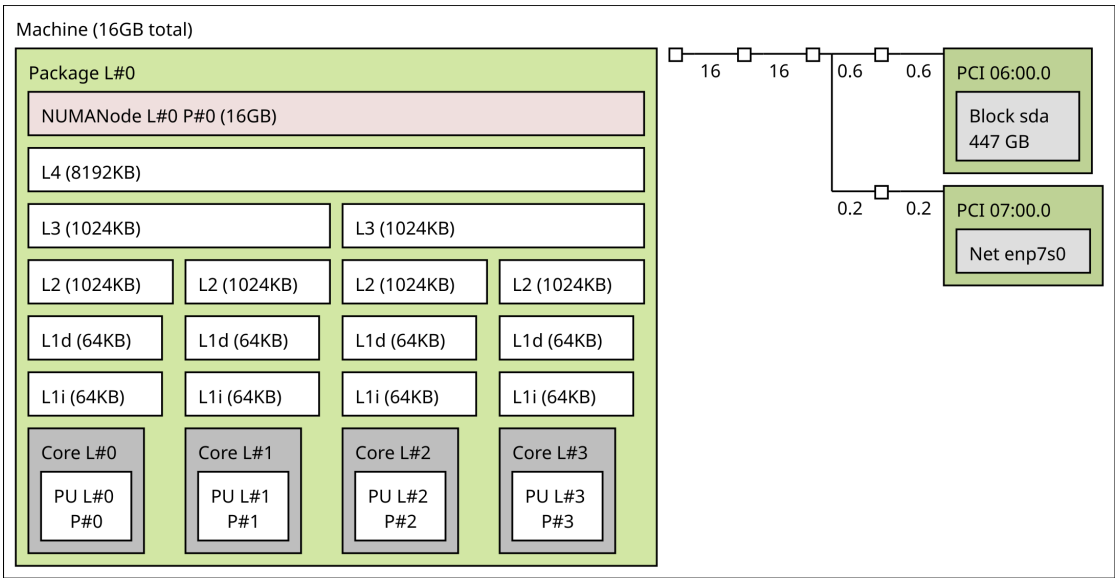

Figure 5.1: Hardware topology of the evaluation system

We replicate a traditional non-CHERI environment in *NOCAP* mode. Tests are compiled using the hybrid compiler from the Morello LLVM toolchain release version 1.8 (based on Clang 14) and dynamically linked against the system's glibc 2.40.

In *PURECAP* mode, we enforce exclusive use of capabilities for all pointers. Tests are compiled using the purecap compiler (also based on Clang 14) from the same toolchain, with the `-mabi=purecap` flag set, and statically linked against an experimental purecap build of musl libc based on its 1.1 release. This static linking allows our test programs to interface with the operating system while maintaining pure capability-based memory access throughout.

All tests are compiled at an optimization level of O2.

6 Architectural Analysis

In this chapter, we investigate RO1, RO2, and RO3. We analyze the hardware characteristics of the Morello processor with a focus on instruction performance. By utilizing a micro-benchmarking framework to measure instruction latency and throughput, we identify key architectural properties, bottlenecks and optimizations specific to Morello's microarchitectural design. The findings allow us to draw conclusions about Morello's performance characteristics and suggest optimization strategies for software running on the platform.

6.1 Methodology

To analyze their performance characteristics, we measure two main metrics for each instruction:

- Throughput (in instructions per cycle)
- Latency (in cycles per instruction)

We measure both by running the same instruction repeatedly in a loop and reading the number of elapsed cycles from the CPUs performance counters. The loop is unrolled, with 16 duplicate instructions executed per iteration, which amortizes the overhead of branching and counter-management across multiple repetitions.

For throughput, our goal when writing the loops is to eliminate any data dependencies between consecutive instructions (see lines 5-6 in Listing 6.1). This enables the processor to pipeline and execute them in parallel as effectively as possible, giving us accurate instructions per cycle readings.

For latency, the exact opposite is the case, as we can only gain accurate readings by guaranteeing that instructions run sequentially. Therefore, we deliberately introduce data dependencies into the loops (see lines 8-9 in Listing 6.2). For instructions where this was not achievable, latency measurements are omitted.

For store instructions specifically, we measure throughput using two distinct memory addressing patterns. In the default pattern, each store instruction writes to a memory location that is offset by the size of the data being stored (e.g., 8 bytes for 64-bit stores,

16 bytes for 128-bit capability stores). This sequential access pattern allows us to measure maximum store throughput by avoiding address conflicts between consecutive stores. For comparison, we also measure "in-place" stores, where all store instructions repeatedly target the same memory address.

```
ldr_throughput_loop:
    cbz x0, ldrt_end
    sub x0, x0, #1
        ...
        ldr x11, [c1] // This only affects the value stored in x11,
        ldr x12, [c1] // which is not an input for the next instruction.
        ...
    b ldr_throughput_loop

    ldrt_end:
        ret
```

Listing 6.1: Example throughput loop

In addition to the aforementioned loop unrolling, we use through-the-loop measurement, which means we first perform a benchmark multiple times using zero loop iterations to obtain the average amount of cycles that can be accounted to loop, test orchestration and runtime overhead. These "empty" runs also serve as warm-up for the main test, where a high number of loop iterations are executed to minimize the effect of outliers on the averaged result, from which the previously determined overhead is then subtracted for the final value. The result is the average cycles per instruction, used directly when measuring latency and inverted when determining instructions per cycle for throughput.

The calculations and orchestration of the tests are managed using a Python test framework, which systematically executes a predefined set of benchmarks, each defined by a unique combination of instruction, measurement metric, and binary type. The binary type parameter can be either *PURECAP* or *NOCAP* Section 5.2, as certain instructions are only available when the executable binary is compiled with capabilities enabled or disabled. Based on the benchmark configuration, the framework selects the appropriate binary to run. To gather statistics on the elapsed CPU cycles, the framework utilizes the `perf` utility, a Linux profiling tool that accesses the CPU hardware performance counters. Cycle counts are collected by running `perf stat -e cycles` and parsing the command line output.

```
// The capability stored in c1 is set to point to the value 0
ldr_latency_loop:
    mov x9, #0
    ldrl_loop:
    cbz x0, ldrl_end
    sub x0, x0, #1
        ...
        ldr x9, [c1, x9] // This affects the value stored in x9,
        ldr x9, [c1, x9] // which *is* an input for the next instruction.
        ...
    b ldrl_loop

    ldrl_end:
        ret
```

Listing 6.2: Example latency loop

6.2 Results

All following measurements are gathered through the benchmarking framework described in Section 6.1. We perform 100 warm-up runs and 10 test runs at 1 000 000 loop iterations.

6.2.1 Memory Instructions

First, we look at the throughput and latency measurements of base and capability-enhanced memory instructions.

The measured instructions are divided into two groups. The first group (compiled in *PURECAP* mode) performs 128-bit or 64-bit memory accesses through capabilities with bounds checking, and the second (compiled in *NOCAP* mode) through regular 64-bit pointers:

- `LDR`: loads 64 bit from memory
- `LDR_CAP`: loads a 128-bit capability from memory
- `LDP`: two 64-bit loads from memory into two registers
- `LDP_CAP`: two 128-bit capability loads from memory into two capability registers
- `STR`: stores 64 bit to memory

Table 6.1: Performance of memory instructions addressed through capabilities

Instruction	Throughput (i/c)	Latency (c/i)
LDR	1.990	4.009
LDP	0.996	-
LDR_CAP	1.990	4.005
LDP_CAP	0.995	4.009
STR	1.981	-
STP	0.996	-
STR_CAP	0.997	-
STP_CAP	0.499	-
STR (in-place)	1.984	-
STP (in-place)	0.333	-
STR_CAP (in-place)	0.333	-
STP_CAP (in-place)	0.166	-
LDAR	1.993	-
STLR	1.990	-

- STR_CAP: stores a 128-bit capability to memory
- STP: two 64-bit stores to memory from two registers
- STP_CAP: two 128-bit capability stores to memory from two capability registers
- LDAR: Load-acquire ensures memory accesses are not reordered before this load
- STLR: Store-release ensures memory accesses are not reordered after this store

We choose the instructions as we expect their examination to yield valuable information needed to answer RQ1 (Section 4.1) and RQ2 (Section 4.2). The load and store instructions, with their varying data size and addressing modes, allow us to gather precise information on data size throughput scaling and addressing mode overhead. Their memory access order enforcing counterparts provide data on if and how capability-enhanced accesses influence multithreaded execution.

We make the following key observations about the data presented in Table 6.1 and Table 6.2:

1. **There is no measurable performance difference between pointer or capability addressed memory access.** When we compare base instructions with their capability addressed counterparts, i.e. instructions that perform the equivalent

Table 6.2: Performance of memory instructions addressed through pointers

Instruction	Throughput (i/c)	Latency (c/i)
LDR	1.986	4.010
LDP	0.996	4.008
STR	1.972	-
STP	0.994	-
STR (in-place)	1.980	-
STP (in-place)	0.333	-
LDAR	1.990	-
STLR	1.985	-

load or store action with the same data size, we clearly see almost identical numbers across both metrics.

2. **The latency is consistent across all load instructions.** Independent of loaded data size and addressing mode, the latency of the operations stay at 4 cycles per instruction.

3. **Load instruction throughput does not scale with data size, only with instruction type.** Single memory access instructions like LDR and LDR_CAP achieve approximately 2 instructions per cycle, regardless of whether the data being loaded is 64-bit or 128-bit (see Figure 6.1). In contrast, paired memory access instructions like LDP and LDP_CAP consistently show throughput of around 1 instruction per cycle, again independent of data size.

4. **Store instruction throughput scales inversely with data size.** Store instruction throughput drops proportionally as data size increases (see Figure 6.1). Single stores like STR reach about 2 instructions per cycle with 64-bit data, while 128-bit stores like STR_CAP achieve only 1 instruction per cycle. This pattern continues with paired operations, where STP_CAP ($2 * 128$ bit) operates at half the throughput of STP ($2 * 64$ bit).

5. **Store instruction throughput is significantly reduced when targeting the same memory address.** Except for the base 64-bit STR, repeatedly storing to the same memory location drastically reduces throughput compared to sequential addressing. In-place stores show approximately $3\times$ lower throughput across all tested store instructions, which store more than 64 bits of data (e.g., STP: $0.996 \rightarrow 0.333$, STR_CAP: $0.997 \rightarrow 0.333$, STP_CAP: $0.499 \rightarrow 0.166$), suggesting a throughput penalty proportional to data size.

6. **Memory ordering instructions show no performance penalty.** The load-acquire (`LDAR`) and store-release (`STLR`) instructions achieve nearly identical throughput, approximately 2 instructions per cycle, as their regular counterparts (`LDR` and `STR`).

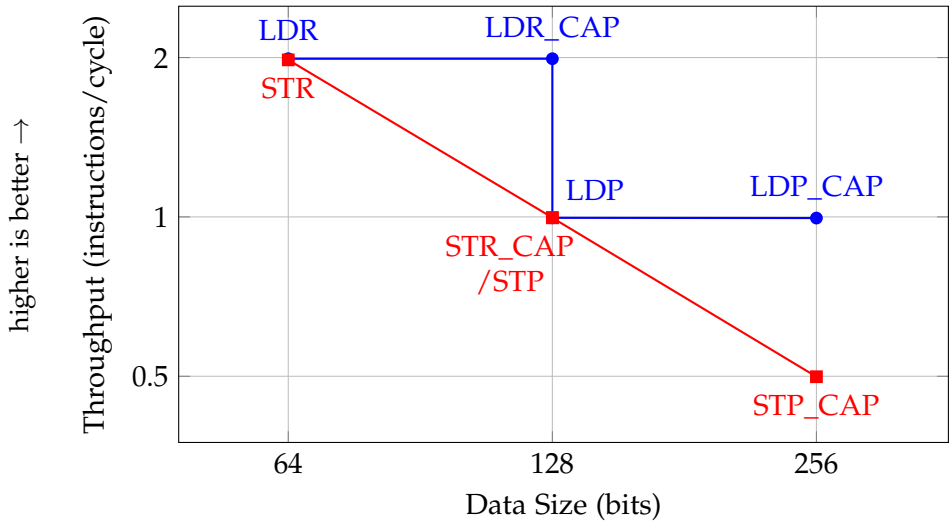

Figure 6.1: Memory instruction throughput versus data size

6.2.2 Capability Instructions

Next, we look at the throughput and latency of instructions that operate on the metadata of capabilities. All the tests conducted here are compiled in *PURECAP* mode.

The measured instructions perform the following operations:

- `CFHI`: copies the upper 64 bits of capability metadata to a 64-bit register
- `CTHI`: copies a 64-bit register to the upper 64-bit of a capability register
- `CVTD_ TOCAP/TOPTR`: converts a pointer into a capability using the capability metadata provided by the DDC and vice versa
- `CVTP_ TOCAP/TOPTR`: converts a pointer into a capability using the capability metadata provided by the Program Counter Capability (PCC) and vice versa
- `CVT_ TOCAP/TOPTR`: converts a pointer into a capability using the metadata provided by a capability parameter and vice versa

Table 6.3: Performance of capability manipulation instructions

Instruction	Throughput (i/c)	Latency (c/i)
CFHI	2.800	1.005
CTHI	2.813	1.007
CVTD_TOCAP	0.998	2.006
CVTD_TOPTR	0.997	1.004
CVTP_TOCAP	0.997	2.004
CVTP_TOPTR	0.998	2.005
CVT_TOCAP	0.998	2.006
CVT_TOPTR	0.997	2.007

We choose the instructions to gather data for addressing RQ3 (Section 4.3)., focusing particularly on the instructions which allow us to extract, set and manipulate capability metadata. The variety of instructions tested, some using the DDC, PCC or an arbitrary capability as a source of metadata allows us to determine if there is a performance difference between these methods.

We make the following key observations about the data presented in Table 6.3:

1. **The performance of CFHI and CTHI significantly exceed that of conversion instructions.** With both having a throughput of around 2.8 instructions per cycle and single-cycle latency, these capability manipulation operations have almost three times higher throughput and half the latency of most conversion instructions.

2. **There is almost no difference in throughput or latency when comparing capability conversion instructions.** Except for CVTD_TOPTR, neither the source of the capability metadata used during conversion nor the direction of the conversion, meaning from capability to pointer or vice versa, appear to cause a measurable difference in instruction performance. They all consistently measure in at a throughput of about 1 instruction per cycle and a latency of roughly 2 cycles per instruction.

3. CVTD_TOPTR **has half the latency of the other conversion instructions.** Compared to its counterpart CVTD_TOCAP and other conversion instructions, the single-cycle latency determined for CVTD_TOPTR represents a clear anomaly.

6.3 Analysis

In the following section we address RO1, RO2, and RO3, by analyzing the findings we made in Section 6.2. We further utilize our results to infer information on how certain operations are implemented on the Morello processor, and how said hardware implementation details may affect the performance of software running on it.

6.3.1 RO1: Effect of Capabilities on Memory Instructions

Our results indicate that the size of capabilities has very little impact on the performance of memory load instructions, while memory store instructions are more negatively affected by the increased data size.

We found no performance difference between base load instructions and their capability loading equivalents. This suggests that the CPUs load buffer entry size and load bandwidth are tuned to process 128-bit capabilities as efficiently as their 64-bit counterparts. As shown in Figure 6.1, throughput scales consistently between single and paired loads. This indicates that pair instructions are handled as separate memory operations, with overall throughput constrained by the processor's memory transfer limitations. This would also explain the relative inefficiency of the `LDP` instruction, which has only half the throughput of `LDR_CAP`, even though they both load 128 bits of data.

In contrast, store instructions show apparent throughput scaling with data size (see Figure 6.1). This indicates that storing capabilities present an immediate performance overhead over storing 64-bit pointers. Even more pronounced is the penalty when repeated stores target the same memory location. When comparing offset-based stores with in-place stores, we observe dramatic throughput reductions for all instructions that store more than 64 bits of data. For example, the throughput of `STP` drops from 0.996 to 0.333 instructions per cycle, while `STR_CAP` similarly falls from 0.997 to 0.333 instructions per cycle. The most severe degradation occurs with `STP_CAP`, which reduces from 0.499 to just 0.166 instructions per cycle when storing to the same location.

Interestingly, the base 64-bit `STR` instruction shows almost no performance difference between offset and in-place stores (1.981 vs. 1.984), suggesting that Morello's store buffer can efficiently handle repeated single-word stores but struggles with larger data sizes or paired instructions. These findings align with the architectural limitations of Morello stated in [Wat+23], which explains that the Morello processor retains the same store throughput and buffer size as the Neoverse N1 it was based on, becoming particularly congested when storing capability pairs. Our results further quantify this limitation, showing that repeated stores to the same location exacerbate the congestion by approximately 3× for instructions handling larger data sizes.

These findings have significant implications for the performance characteristics of software running on Morello. While we find that the per-instruction overhead of capabilities compared to regular pointers is relatively small for load operations, the reduced throughput could impact long sequences of stores. For example, software that needs to bulk copy large blocks of capabilities, such as compacting garbage collectors or capability-aware memory allocators, should expect lower performance than copying regular pointers. This contrast is even more pronounced when repeatedly updating the same memory locations, such as in data structures with frequent in-place modifications. In comparison, load-dominated data structures like trees and linked lists, where capability overhead should be less noticeable, should perform more efficiently on Morello hardware.

6.3.2 RO2: Bounds Checking Overhead

We find neither a throughput nor a latency difference when comparing instructions that utilize pointers to their equivalents, which utilize capabilities for addressing memory. This is expected, as in both cases, the bound checking of the memory access still occurs, in the case of pointers against the DDC and in the case of capabilities against the bounds stored in the capability itself, with the bound metadata in both instances being stored in CPU registers. Our findings confirm that no special architectural optimizations have been made to make either of the cases more performant than the other.

These results indicate that software running on Morello incurs minimal direct performance overhead when switching from regular pointers in hybrid mode to capabilities, at least at the instruction level. While there is also no immediate performance advantage, the software can leverage built-in bounds-checking capabilities to potentially improve performance by eliminating redundant software bounds checks that would otherwise be needed. This presents an interesting optimization opportunity when porting software to hybrid or pure-capability modes, as the hardware bounds-checking infrastructure could replace explicit bounds checking in the code, reducing the total number of instructions executed.

6.3.3 RO3: Latency and Throughput of Morello Instructions

We measured the capability manipulation instruction `CFHI` and `CTHI` at a high throughput of about 2.8 instructions per cycle and low single-cycle latency. This implies that dedicated execution units exist for manipulating capability metadata fields. In contrast to this, most capability conversion instructions show a maximum throughput of 1 instruction per cycle and a latency of 2 cycles, indicating that these operations require multiple steps in the processor pipeline and can not be pipelined as efficiently. One

notable exception to this is CVTD_TOPTR, which achieves single cycle latency, suggesting potential hardware optimization for converting capabilities derived from the DDC back to pointers.

These hardware characteristics have essential implications for Morello software performance. Applications could leverage the high throughput capability manipulation instructions (CFHI and CTHI) to efficiently inspect and modify capability metadata when necessary. However, the lower throughput of conversion instructions suggests that frequent switching between capabilities and pointers should be minimized to avoid performance bottlenecks. This is particularly relevant for hybrid mode software, where capability and pointer representations might need to be converted frequently at interface boundaries. The optimization for CVTD_TOPTR could be exploited in scenarios where hybrid mode software needs to frequently pass pointers derived from the DDC back to non-capability code.

7 Related Work

In this chapter, we look at previous work analyzing the performance impact and microarchitectural limitations of CHERI/Morello.

7.1 Evaluation of Morellos microarchitectural limitations

In "Early performance results from the prototype Morello microarchitecture", the authors, who are also, developers of the Morello hardware conduct performance tests using the SPECint 2006 benchmark suite on a Field-Programmable Gate Array (FPGA) synthesizing variations of the Morello architecture. To determine how key microarchitectural limitations affect CPU performance, they conduct their tests using different software and hardware configurations, varying ABIs and hardware implementations.

One of the architectural constraints they identify is **untuned store throughput and buffer sizes**, constraints whose performance implications we were able to replicate and analyze in Subsection 6.3.1. After modifying the microarchitecture to include a larger store queue and applying several other software and hardware patches, they observe an 18.7% overhead reduction from the store queue modification.

7.2 Performance Evaluation of CHERI Linux

In "Cherifying Linux: A Practical View on using CHERI", the authors present a full CHERI port of the Linux kernel and user space [Wan+24]. They evaluate the overhead of the CHERI port and its spatial memory safety on a 5-stage in-order RISC-V core with CHERI support, synthesized on a FPGA. Throughout the range of their benchmarks, which tests integer performance, algorithms and commercially representative workloads, they observe a maximum overhead of 49.1% and minimum of 1.7%, with all results averaging out an overhead of about 15% compared to executing the binaries with no CHERI instrumentation.

They make the following hypotheses to explain the overhead:

- **The CHERI variants execute additional instructions**, caused by the removal of optimizations in glibc, which was incompatible with CHERI. They speculate that

some of the associated performance loss could be recovered by implementing new CHERI-compatible optimizations.

- **The memory subsystem is under a higher load**, as cache access patterns and memory usage are different due to altered code and the use of 128-bit capabilities.

Our results regarding the reduced store throughput for 128-bit values could potentially support their speculation on the increased memory subsystem load, as slower store operations could lead to more congestion in the pipeline and memory subsystem. However, as our analysis is mainly concerned with instruction level performance analysis, not large-scale programs such as an operating system, and is furthermore based on tests conducted on a different microarchitecture (Arm Morello vs. RISC-V CHERI simulated on FPGA), we do not believe our findings can be used to reinforce their hypotheses meaningfully.

8 Conclusion

We have presented an analysis of the instruction level performance characteristics in the Morello processor. Our research was motivated by the observation that while the security benefits of CHERI and Morello are well documented, their performance impacts are less well understood, particularly at the instruction level. Through careful micro-benchmarking and analysis, we have gained valuable insights into how capabilities affect instruction execution and memory operations.

Our investigation into memory instruction performance (RO1, Section 4.1) reveals that load operations show minimal overhead when using capabilities compared to regular pointers. Load instructions achieve consistent throughput regardless of whether they are accessing 64-bit or 128-bit values, suggesting the processor is well-optimized for capability loads. However, store operations show clear performance scaling with data size, with capability stores achieving only half the throughput of regular pointer stores. This asymmetry between load and store performance has important implications for software design on Morello, particularly for applications that perform extensive memory copies or updates.

Regarding bounds checking overhead (RO2, Section 4.2), we find no significant performance difference between memory accesses using capabilities versus accesses using regular pointers checked against the DDC. This suggests that the hardware implementation of bounds checking is equally efficient for both modes and that the choice between hybrid and pure-capability modes can be made based on security requirements rather than performance concerns. The results also indicate potential opportunities for performance optimization by leveraging hardware bounds checking to eliminate redundant software checks.

Our analysis of capability manipulation instructions (RO3, Section 4.3) reveals varying levels of hardware optimization. Some operations, particularly `CFHI` and `CTHI`, show high throughput and low latency, suggesting dedicated execution units for capability metadata manipulation. Conversion instructions generally show lower performance, though with some exceptions like `CVTD_TOPTR`, indicating selective hardware optimization of frequently used operations. These findings provide valuable guidance for compiler optimizations and software design patterns on Morello.

We believe our results contribute important insights for both software developers and hardware designers working with CHERI-based architectures. For software developers,

our findings highlight performance characteristics that should influence design decisions, such as minimizing capability stores and conversion operations where possible. For hardware designers, our results identify areas where additional optimization could improve performance, particularly in store operations and capability conversions.

9 Future Work

The instruction level performance analysis of the Morello processor conducted in this thesis provides the basis for several future research directions.

9.1 Methodology Extension

The current measuring methodology could be expanded to inspect more instructions and metrics. While we thoroughly observe the performance of memory instructions, a more comprehensive collection of capability-related instructions, including instructions that manipulate permission masks, object types, or perform direct arithmetic on the integer address could be included in future testing. Furthermore, more complex test cases, where specific sequences of different dependent or independent instructions are executed, could be insightful for analyzing processor pipelining behavior and execution unit utilization patterns.

Additionally, the metrics could be expanded to include measurements of memory subsystem effects like cache misses and branch prediction accuracy, as earlier research suggests that branch prediction and inefficient cache behavior are key bottlenecks for capability-enhanced code [Wat+23; Wan+24].

9.2 Application Level Analysis

Building upon the micro benchmarks this thesis conducted, larger-scale benchmarks of applications could produce deeper insight into real world performance implications of CHERI/Morello. This could include compiling and running standard benchmarking suites in both pure-capability and hybrid mode, while gathering detailed instruction execution statistics. Such an approach would allow the assessment of how the instruction-level characteristics documented in this thesis manifest in complete application workloads and whether the performance bottlenecks identified are significant in practice.

This analysis could also be applied to pre-existing porting work, where their performance and overhead results are reevaluated using instruction statistics and our

microbenchmark results, potentially providing more concrete explanations for observed performance differences.

Abbreviations

CHERI Capability Hardware Enhanced RISC Instructions

DDC Default Data Capability

PCC Program Counter Capability

ISA Instruction-Set Architectures

MMU Memory Management Unit

DRAM Dynamic Random-Access Memory

ABI Application Binary Interface

SDP System Development Platform

SoC System on a Chip

FPGA Field-Programmable Gate Array

CPU Central Processing Unit

List of Figures

List of Tables

Listings

Bibliography

[ARM] ARM Ltd. *Arm Architecture Reference Manual for A-profile architecture*. White Paper. Accessed: 2024-03-28.

[ARM23] ARM Ltd. *Morello Prototype Architecture Overview User Guide*. Oct. 24, 2023.

[JA24] Jeff Vander Stoep and Alex Rebert. *Eliminating Memory Safety Vulnerabilities at the Source*. Google Online Security Blog. Sept. 25, 2024. URL: https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html (visited on 03/27/2025).

[LJS23] D. Lowther, D. Jacob, and J. Singer. *CHERI Performance Enhancement for a Bytecode Interpreter*. Sept. 12, 2023. DOI: 10.48550/arXiv.2308.05076. arXiv: 2308.05076[cs].

[MSR19] MSRC. *A proactive approach to more secure code | MSRC Blog | Microsoft Security Response Center*. July 16, 2019. URL: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (visited on 03/27/2025).

[Pro] T. C. Project. *Memory Safety*. Accessed on March 14, 2024.

[VZ] J. Vander Stoep and C. Zhang. *Queue the Hardening Enhancements*. Accessed on March 14, 2024.

[Wan+24] K. Wang, D. Kasatkin, V. Ahlrichs, L. Auer, K. Hohentanner, J. Horsch, and J.-E. Ekberg. "Cherifying Linux: A Practical View on using CHERI." In: *Proceedings of the 17th European Workshop on Systems Security*. EuroSys '24: Nineteenth European Conference on Computer Systems. Athens Greece: ACM, Apr. 22, 2024, pp. 15–21. ISBN: 979-8-4007-0542-7. DOI: 10.1145/3642974.3652282.

[Wat] R. Watson. *Department of Computer Science and Technology – CHERI: The Arm Morello Board*. URL: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-morello.html (visited on 03/20/2025).

[Wat+19] R. N. M. Watson, S. W. Moore, P. Sewell, and P. G. Neumann. "An Introduction to CHERI." In: (Sept. 2019).

[Wat+23] R. N. M. Watson, J. Clarke, P. Sewell, J. Woodruff, S. W. Moore, G. Barnes, R. Grisenthwaite, K. Stacer, S. Baranga, and A. Richardson. *Early performance results from the prototype Morello microarchitecture*. Tech. rep. UCAM-CL-TR-986. 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500: University of Cambridge, Computer Laboratory, Sept. 2023.