Sweet or Sour CHERI: Performance Characterization of the Arm Morello Platform

Xiaoyang Sun[1] Jeremy Singer[2] Zheng Wang[1]

[1]University of Leeds, UK [2]University of Glasgow, UK

Abstract

Capability Hardware Enhanced RISC Instructions (CHERI) is an emerging hardware approach to memory safety that enforces strong spatial and temporal protections. This paper presents the most comprehensive performance evaluation of CHERI to date. Using on-chip performance monitoring counters (PMCs) on the CHERI-enabled Arm Morello platform, we analyze 20 C/C++ applications, including SPEC CPU2017, a SQL database engine, a JavaScript engine, and a large language model inference framework, across three CHERI Application Binary Interfaces (ABIs). We find that CHERI overheads range from negligible to 1.65×, with the highest costs in pointer-intensive and memory-sensitive workloads, largely due to increased memory traffic and higher L1/L2 cache pressure from 128-bit capabilities. Importantly, our projections suggest that modest microarchitectural improvements could significantly reduce these costs, enabling CHERI to deliver memory safety with minimal performance impact. We hope these findings offer timely evidence to guide the development of future architectures that combine strong memory security with high performance.

1. Introduction

Memory safety vulnerabilities, such as buffer overflows and use-after-free errors, remain a major cause of software security flaws, especially in systems written in C and C++. Industry reports estimate that around 70% of critical security bugs are caused by memory issues [11, 22, 24].

Capability Hardware Enhanced RISC Instructions (CHERI) [35, 37, 38] address this problem by extending conventional instruction set architectures with hardware-enforced *capabilities*, a hardware-enforced pointer type that carries metadata about memory bounds and permissions. Capabilities ensure that a pointer can only access authorized memory, enabling fine-grained memory protection. The Arm Morello system [14] is a prototype architecture of CHERI. It integrates a CHERI extended ARMv8-A processor with a GPU, peripherals, and memory system on a single chip. Morello is designed to support industrial evaluation of CHERI, provide evidence for potential adoption, and enable further research. The system is supported by the CheriBSD operating system, a CHERI-enabled version of FreeBSD [4], which allows software to run under different CHERI Application Binary Interfaces (ABIs). This provides a practical platform to explore the trade-offs between memory safety and performance.

While the benefit of CHERI is compelling, understanding its performance impact on a modern and complex architecture like Morello is essential for its broader adoption. An early report of CHERI performance [36] conducted on SPECInt 2006 provides some high-level performance results. It highlighted several sources of capability-associated overhead, including the increased pointer size (128-bit capabilities versus 64-bit native pointers), branch prediction effects from Program Counter Capability (PCC) bounds, and store buffering behavior. Furthermore, as the CHERI ecosystem, including operating systems and compilers, has matured over the past few years, there is a need to revisit the performance results on CHERI-enabled architecture.

Building on [36], we extend the evaluation to a wider range of workloads and more recent benchmark suites. Specifically, we conduct a comprehensive empirical evaluation of CHERI's performance impact on the Morello platform. Using the SPEC CPU2017 benchmark suite together with real-world workloads - including the `LLaMA.cpp` large language model (LLM) inference framework [20], the `QuickJS` JavaScript/ECMAScript engine [27], and the `SQLite` database system [30] - we evaluate performance under three CHERI Application Binary Interfaces (ABIs): hybrid, pure-capability (purecap), and purecap-benchmark. Specifically, the *hybrid* mode uses capabilities only where explicitly annotated in C and C++, allowing conventional and capability-based code to coexist and enabling incremental adoption. The *pure-capability* mode enforces pervasive use of capabilities across both language-level pointers (e.g., variables referencing heap, stack, or function addresses) and sub-language pointers (e.g., stack pointers and return addresses). The *benchmark* mode is a variant of the purecap ABI, designed to work around limitations in the current Morello branch predictor. It retains the same memory layout and nearly identical code generation as purecap code, but relaxes certain protection mechanisms to isolate architectural overheads from software-level effects. By comparing these modes, our study provides detailed insights into the trade-offs between compatibility, security, and performance in capability-based systems, helping inform future hardware and compiler design decisions.

To collect performance data, we leverage the on-chip Performance Monitoring Counters (PMCs). We introduce a practical methodology for interpreting PMC data to characterise workload behavior on capability-based systems, providing a template for future performance analysis. Our study

offers a detailed examination of how CHERI capabilities influence low-level execution dynamics, including cache and TLB behaviour, frontend pipeline activity (instruction fetch and branch prediction), and backend execution stalls. Since Morello is a research prototype, we take care to distinguish effects inherent to the CHERI model from those arising due to implementation-specific artefacts, ensuring our findings are relevant to the design of future memory-safe hardware.

Compared to the most closely related prior work [36], our study offers several new contributions. First, we increased the number of SPEC CPU benchmarks from seven to 17, covering more diverse workloads. Second, we included additional real-world applications, including a JavaScript/ECMAScript engine [27], a SQL database benchmark [30] and an LLM inference workload [20], to ensure the evaluation is more representative. Third, we provide a more detailed analysis for each benchmark and introduce new metrics that give clearer insights into program behavior. Fourth, we find that there are workloads where CHERI capabilities introduce minor or no overhead, especially those with moderate memory intensity. This helps inform more targeted optimization.

This paper makes the following contributions to the understanding of CHERI performance:

- A comprehensive study of application performance under different CHERI configurations;
- Identification and quantified results for key contributors to performance overheads observed in pure-capability mode, including those inherent to CHERI and those related to hardware-specific implementation;
- Actionable insights and recommendations for designing future memory security features at the hardware level.

Online materials: All scripts, code (excluding proprietary benchmarks like SPEC CPU 2017), and data are publicly available at https://github.com/xshaun/iiswc25-ae.

2. Background

2.1. The CHERI Memory Protection Model

CHERI extends conventional ISAs with *capabilities* - hardware-enforced, unforgeable pointer types with embedded metadata to provide fine-grained memory protection. For 64-bit ISAs, capabilities are typically encoded using a 128-bit compressed format [38]. Each CHERI capability encapsulates a memory address along with attributes specifying the bounds of the accessible memory region and the operations permitted within it. This metadata includes a 64-bit base address and a representation of a 64-bit limit address (or length) to define the valid memory range, a set of permission bits that govern allowable actions (e.g., load, store, execute, load/store capability), and an out-of-band single-bit validity tag protected by hardware. The tag ensures the capability has not been forged or corrupted and can only be manipulated through CHERI-defined instructions, providing a strong foundation for memory safety and secure programming.

CHERI capabilities are enforced by the hardware. Every memory access issued through a capability instruction - or within a capability-enabled execution mode - is subject to hardware-enforced checks on the capability's bounds and permissions. If an access attempt falls outside the defined bounds, lacks the required permissions, or uses an invalid (untagged) capability, the hardware raises an exception, typically a capability violation fault. These mechanisms collectively enforce fine-grained spatial memory safety by preventing out-of-bounds accesses. They also support temporal memory safety by enabling the revocation or invalidation of capabilities, thereby mitigating risks from dangling pointers.

2.2. Morello Platform

We target the Arm **Morello** platform [14], an experimental SoC developed to support realistic evaluation of the CHERI architecture [4]. Morello integrates 128-bit CHERI capabilities into a quad-core, 2.5 GHz ARMv8.2-A Neoverse N1 CPU with out-of-order superscalar cores. Each core features a 64KB L1 I-cache and D-cache, both 4-way set-associative with 64B lines. The per-core L2 cache is 1MB, 8-way set-associative with 64B lines, and there is a 1MB shared LL cache for all 4 cores.

Morello leverages the existing Neoverse N1 microarchitecture. This choice enabled rapid integration but limited the scope for CHERI-specific optimization. CPU components such as out-of-order execution, branch prediction, and cache design were preserved with minimal changes, resulting in performance drawbacks [36]. In particular, the branch predictor does not track changes in the Program Counter Capability (PCC) bounds, leading to stalls during interlibrary control transfers. CHERI's heap temporal safety relies on data-dependent exceptions for store operations, which the N1 is not designed to handle efficiently. Store queues and buffers, sized for 64-bit operations, become bottlenecks when handling 128-bit capability stores. Finally, the platform lacks a capability-aware version of the multiply and add (MADD) instruction.

2.3. CheriBSD

Morello runs a modified FreeBSD OS called *CheriBSD*, which manages capabilities at the operating system level. It ensures that user-space memory allocations (e.g., stack, heap, globals) are properly bounded by capabilities. CheriBSD also sets up initial program capabilities, including the PCC for executable regions and the Default Data Capability (DDC) for defining the initial data access scope.

2.4. CHERI ABIs on Morello

CheriBSD on Morello supports three main ABIs, each offering different trade-offs between CHERI protection and performance:

Hybrid (AArch64): This baseline ABI runs standard 64-bit ARMv8-A AArch64 instructions, using conventional integer pointers. Under this hybrid mode, CHERI capabilities are only used when explicitly introduced through language extensions or CHERI-aware libraries. This mode serves as a performance baseline for measuring CHERI overheads.

Pure-capability (purecap): In this mode, all user-space pointers, including those in C/C++ programs (heap, stack,

globals, function pointers) and implicit ABI structures (stack/frame pointers, return addresses, GOT entries), are represented as 128-bit CHERI capabilities. This enables full spatial memory safety by enforcing capability checks on all memory accesses.

Purecap Benchmark (benchmark): This ABI is tailored for performance analysis. It modifies purecap by using a single global PCC and performing function calls and returns using integer jumps instead of capability jumps. This avoids frequent PCC-bound updates and reduces stalls caused by Morello's branch predictor limitations. All other pointers remain 128-bit capabilities, preserving the memory and access profile of purecap code. As a result, this mode isolates the performance cost of PCC handling [36], enabling detailed analysis of CHERI-specific overheads.

3. Performance Analysis Methodology

To characterize workload performance on the CHERI-enabled Morello platform, we combine a hierarchical top-down analysis framework with detailed microarchitectural event monitoring using on-board PMUs.

3.1. Methodology for Performance Analysis

We use PMUs to capture microarchitectural events and analyze them using the top-down performance analysis methodology [2, 42]. This approach, widely applied to evaluate out-of-order superscalar processors [7, 18, 21, 23], provides a hierarchical framework to drill down from high-level pipeline behavior to specific microarchitectural causes. By applying this methodology, we assess how effectively the CPU pipeline is utilized, identify bottlenecks, and characterize workload behavior across different CHERI ABIs.

Specifically, at the top level, the methodology classifies each CPU pipeline slot into one of four categories:

Retiring: micro-operations (μops) are successfully executed and retired. This reflects the amount of useful work completed; a high Retiring percentage indicates good pipeline utilization.

Bad Speculation: Wasted and non-retired work (μops) due to incorrect speculative execution, typically due to branch mispredictions and the resulting pipeline flushes.

Frontend Bound: Slots where the backend is ready but the frontend fails to deliver enough μops (e.g., instruction cache misses, decode bottlenecks).

Backend Bound: Slots where the frontend delivers μops but the backend cannot accept them, usually due to execution or memory bottlenecks (e.g., execution unit contention, memory latency, full buffers).

If a significant portion of the CPU cycles is attributed to Frontend or Backend stalls, further analysis is performed to identify the source of the stalls:

Frontend Analysis: Stalls may arise from instruction fetch latency (such as L1 instruction cache or instruction TLB misses) or limited fetch bandwidth, often linked to memory hierarchy behavior at L2 or L3.

Backend Analysis: Stalls can be categorized as memory bound (for example, due to data cache or DRAM latency) or core bound (due to limited availability of execution resources, captured by events such as `INS_SPEC`, `DP_SPEC`, `ASE_SPEC`, or `BR_IMMED_SPEC`).

This structured methodology helps focus the analysis on dominant bottlenecks, avoiding overinterpretation of isolated event counts. It is especially suitable for CHERI evaluation, where architectural features, such as 128-bit capabilities and hardware checks, can affect both instruction fetch and memory access behavior.

3.2. Performance Monitoring and Metrics

We use PMU events available on the Neoverse N1 CPU core of the Morello platform. These events are collected using the `PMCSTAT` utility on CheriBSD. As the platform only provides up to six configurable PMUs to be used at any time, benchmarks are executed multiple times (nine runs in this work) to collect a larger set of events. Event selection is guided by our top-down evaluation methodology (§3.1) and tailored to analyze CHERI-specific behavior. Table 1 summarizes the key raw PMU events and the derived metrics used in this study. As PMUs do not expose per-instruction counters for CHERI, we infer overheads via increased micro-op activity by comparing ABIs - for instance, purecap vs. purecap benchmark highlights branch prediction stalls from capability branches.

We have specifically considered CHERI-specific events such as `CAP_MEM_ACCESS_RD` and `MEM_ACCESS_RD_CTAG`. These allow for direct quantification of memory operations that are capability-based or involve capability tag checks, providing a unique lens into CHERI's runtime behavior that is not possible with standard PMUs alone. For example, calculating the ratio of `CAP_MEM_ACCESS_RD` to total load instructions (`LD_SPEC`) can reveal the capability load density in different workloads and ABIs, detailed in Table 1.

Generalization. While this study focuses on CHERI, our methodology and metrics should generalize to other memory-safety mechanisms. For instance, `load/store` density and the safe instruction ratio also apply to RISC-V ePMP [26] and related schemes. Since CHERI exemplifies hardware fat-pointer designs, our findings on capability-related overheads are broadly relevant to future architectures.

3.3. Application Workloads

Our evaluation uses a total of 20 different workload applications written in C/C++: 17 benchmarks from SPEC CPU 2017 [6], a JavaScript/ECMAScript engine (QuickJS [27]), an LLM inference engine (LLaMA.cpp [20]), and a SQL database engine (SQLite3 [30]).

SPEC CPU includes both integer and floating-point benchmarks and is an industry standard suite to evaluate the performance of the CPU core and memory subsystem. We use the *train* input set in our evaluation due to the runtime and storage constraints of the Morello platform.

QuickJS is a self-contained interpreter that nearly implements the full ES-2023 specification. It serves as a proxy for

TABLE 1: Key PMU Events and Derived Metrics for Morello Characterization.

Metric Category	Raw PMU Event(s)	Derived Metric	Formula
Cycle Accounting	CPU_CYCLES, INST_RETIRED	IPC (Instructions Per Cycle)	INST_RETIRED / CPU_CYCLES
		CPI (Cycles Per Instruction)	CPU_CYCLES / INST_RETIRED
Top-Level Stalls	STALL_FRONTEND, STALL_BACKEND, BR_MIS_PRED_RETIRED (approx. for bad speculation)	Frontend Bound %	STALL_FRONTEND / CPU_CYCLES
		Backend Bound %	STALL_BACKEND / CPU_CYCLES
		Bad Speculation % (approx.)	1 - Retiring % - Frontend % - Backend %
		Retiring %	INST_SPEC / SUM(*_SPEC)
Branch Prediction	BR_RETIRED, BR_MIS_PRED_RETIRED	Branch Misprediction Rate	BR_MIS_PRED_RETIRED / BR_RETIRED
L1 Instruction	L1I_CACHE_REFILL, INST_RETIRED (for MPKI), L1I_CACHE	L1I Miss Rate (MR)	L1I_CACHE_REFILL / L1I_CACHE
		L1I Misses Per Kilo Inst (MPKI)	L1I_CACHE_REFILL / INST_RETIRED * 1000
L1 Data	L1D_CACHE_REFILL, INST_RETIRED (for MPKI), L1D_CACHE	L1D Miss Rate (MR)	L1D_CACHE_REFILL / L1D_CACHE
		L1D MPKI	L1D_CACHE_REFILL / INST_RETIRED * 1000
L2 Unified	L2D_CACHE_REFILL, INST_RETIRED (for MPKI), L2D_CACHE	L2 Miss Rate (MR)	L2D_CACHE_REFILL / L2D_CACHE
		L2 MPKI	(L2D_CACHE_REFILL / INST_RETIRED) * 1000
Last Level (LLC)	LL_CACHE_MISS_RD, INST_RETIRED (for MPKI), LL_CACHE_RD	LLC Read Miss Rate (MR)	LL_CACHE_MISS_RD / LL_CACHE_RD
		LLC Read MPKI	LL_CACHE_MISS_RD / INST_RETIRED * 1000
Instruction TLB (I-TLB)	ITLB_WALK (page table walks), INST_RETIRED (for MPKI), L1I_TLB (for walk rate)	ITLB Page Walk Rate	ITLB_WALK / L1I_TLB
		ITLB Walks Per Kilo Inst (WPKI)	ITLB_WALK / INST_RETIRED * 1000
Data TLB (D-TLB)	DTLB_WALK (page table walks), INST_RETIRED (for MPKI), L1D_TLB (for walk rate)	DTLB Page Walk Rate	DTLB_WALK / L1D_TLB
		DTLB WPKI	DTLB_WALK / INST_RETIRED * 1000
CHERI-Specific Memory	CAP_MEM_ACCESS_RD, CAP_MEM_ACCESS_WR, MEM_ACCESS_RD_CTAG, MEM_ACCESS_WR_CTAG	Capability Load Density	CAP_MEM_ACCESS_RD / LD_SPEC
		Capability Store Density	CAP_MEM_ACCESS_WR / ST_SPEC
		Capability Traffic Share	(CAP_MEM_ACCESS_RD + CAP_MEM_ACCESS_WR) / (MEM_ACCESS_RD + MEM_ACCESS_WR)
		Capability TAG Overhead	(MEM_ACCESS_RD_CTAG, MEM_ACCESS_WR_CTAG) / (MEM_ACCESS_RD + MEM_ACCESS_WR)
Memory Load	LD_SPEC, ST_SPEC, DP_SPEC, ASE_SPEC, VFP_SPEC	Instruction-mix-based Memory Intensity (MI)	(LD_SPEC + ST_SPEC) / (DP_SPEC + ASE_SPEC + VFP_SPEC)

Note: These events, such as `L1I_CACHE_REFILL`,`L1D_CACHE_REFILL`, `MEM_ACCESS_RD`, `MEM_ACCESS_WR`, etc., are total access counts for the respective units. The D-side Page Table Walk Rate represents total D-side accesses. `*_SPEC` means `INST_SPEC, LD_SPEC, ST_SPEC, DP_SPEC, ASE_SPEC, BR_RETURN_SPEC, BR_INDIRECT_SPEC, BR_IMMED_SPEC, VFP_SPEC, CRYPTO_SPEC`.

TABLE 2: Benchmark memory intensity values

Benchmark	MI	Benchmark	MI
510.parest_r	0.922	519.lbm_r	0.438
520.omnetpp_r	1.164	523.xalancbmk_r	0.860
531.deepsjeng_r	0.489	541.leela_r	0.565
544.nab_r	0.420	557.xz_r	0.514
620.omnetpp_s	1.165	623.xalancbmk_s	0.860
631.deepsjeng_s	0.496	641.leela_s	0.565
644.nab_s	0.424	657.xz_s	0.504
LLaMA.cpp (inference)	0.309	LLaMA.cpp (matmult)	0.432
SQLite	0.816	QuickJS	0.680

We classify programs according to their memory intensity (MI) defined in Table 1: values below ∼0.6 indicate compute-intensive and values between ∼0.6 and 1.0 indicate balanced resource usage, while values above 1.0 correspond to memory-centric workloads.

browser-like workloads, where performance-critical behavior is concentrated in the interpreter loop. This makes it valuable for understanding program behavior that involves capability-enhanced instructions after they have been fetched. Performance was tested on *Test262: ECMAScript Test Suite* [33], that is, for the most recently published ECMA specifications.

LLaMA.cpp is an LLM inference runtime for the execution and deployment of the standalone model. Its performance is primarily constrained by memory bandwidth rather than raw FLOPs, making it an ideal case for examining CHERI's impacts on memory access patterns. *inference* uses `7B/ggml-model-q8_0.gguf` to process a short prompt (`--n-prompt=512`) and generate tokens (`--n-gen=128`). *matmul* pseudorandomly generates two FP32 matrices with dimensions (11008,4096) and (11008,128).

SQLite3 is a widely used embedded SQL engine, known for its compact and reliable architecture. It is typically latency-bound on random I/O and sensitive to cache behavior, offering a compelling use case for evaluating CHERI's compartmentalization capabilities through diverse and frequent SQL queries, herein, using the *speedtest1.c* program [31].

We have adapted the benchmarks with minimal CHERI-specific changes, detailed in Table 6 in the Appendix, to ensure successful compilation and execution with CHERI C/C++ [39] on the Morello platform. These changes do not affect program performance. To support a concrete, non-conjectural analysis of CHERI's impact, we classify the benchmarks based on memory instruction intensity in Table 2. This classification enables a more precise interpretation of CHERI's architectural implications, such as the effect of 128-bit pointers on memory layout and traffic.

3.4. Experimental Setup

All experiments were carried out on the Arm Morello platform running CheriBSD 25.03. The system is configured with 2×8 GB of DDR4 memory running at 2,933 MT / s. The Morello SoC features a quad-core CPU based on the Neoverse N1 microarchitecture with hyperthreading disabled to minimize performance variability during measurements.

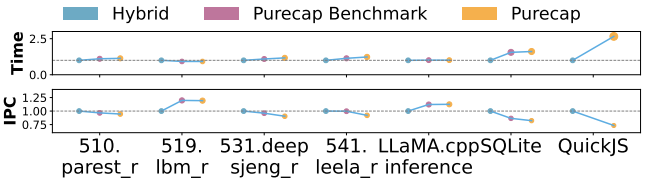

Figure 1: The overall execution performance (normalized to the hybrid mode.)

We *cross-compiled* programs on a Linux development machine using the CHERI LLVM/Clang compiler toolchain (`morello/llvm-project` with `commit 671d6dbe2b7` through [10]), with *-O3* as the optimization flag. No CHERI-specific performance tuning or code modifications were applied, beyond those required to port the programs to CHERI C/C++, to establish a performance baseline.

As explained in §3.2, each benchmark was executed multiple times to capture the full set of desired PMU events (nine runs). We report the arithmetic mean of each metric across ten times of runs. Since each benchmark was run in an isolated and unloaded environment, the variance was consistently below 1%.

4. CHERI Performance Analysis on Morello

This section presents the quantitative results of the workload characterization study on the Morello platform. Our evaluation focuses on how CHERI's architectural features and different ABIs impact performance at macroscopic and microarchitectural levels.

4.1. Runtime Overheads and IPC

The introduction of CHERI capabilities, particularly in the pure-capability (purecap) mode, can lead to noticeable performance overheads compared to the baseline hybrid (AArch64) ABI. For SPEC benchmarks (with the *train* input), purecap code has an overhead ranging from 0% to 165% compared to hybrid ABI. The 60.3% out of the 103% slowdown of `523.xalancbmk_r` could be mitigated by using the purecap benchmark ABI, which specifically addresses a branch prediction limitation in the Morello prototype related to the PCC bounds. The purecap benchmark ABI can typically reduce the purecap overhead by around 0% to 10%. Figure 1 and Table 3 suggest a high variability of runtime overhead in purecap mode, depending on the specific workloads, and in some cases, the *CHERI features introduce zero overhead and even modest performance improvement*, such as `519.lbm_r` and `LLaMA.cpp matmul` benchmarks.

The increase in execution time is generally correlated with a decrease in IPC due to poor instruction-set parallelism. For example, the memory-intensive benchmark `520.omnetpp_r` shows a drop in IPC from 0.578 in hybrid mode to 0.554 in the purecap benchmark and further to 0.516 in purecap mode, corresponding to its significant increase in execution time from 81.73s to 142.30s and 153.21s, respectively. Conversely, compute-intensive benchmarks like `531.deepsjeng_r` show a more modest IPC reduction (1.702

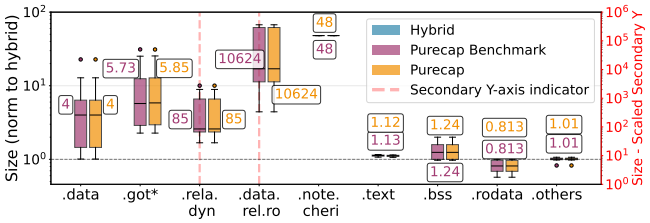

Figure 2: The distribution of program section sizes across benchmarks, normalized to the hybrid mode (numbers indicate median values.)

hybrid to 1.539 purecap) and consequently a smaller runtime increase (67.42 to 78.85 s).

The Branch Misprediction Rate (MR) is a metric that can influence overhead, its impact being contingent upon the specific case under consideration. For example, in `510.parest_r`, there is a decrease in the branch MR from 0. 9% to 0. 76%, but an overhead of 13.81% persists. However, in `523.xalancbmk_r`, the overhead experiences an increase of 105% concomitant with an increase in branch MR 20%.

4.2. Impact on Binary Size

Figure 2 reports the impact of the three ABI modes on binary size across various program sections of the binary. We use the hybrid ABI as the baseline and normalize the sizes of the purecap and purecap benchmark binaries accordingly. Overall, CHERI capability metadata introduces approximately *a 5% increase in total binary size*, though the overhead varies significantly across different sections. Notably, the `.data.rel.ro` and `.note.cheri` sections are absent in the hybrid ABI and are thus shown as absolute sizes. The `.rela.dyn` section shows the most significant growth, approximately 85× larger than the hybrid baseline, reflecting the considerable storage overhead introduced by CHERI capability pointers and the associated metadata required for dynamic linking and relocation. In contrast, sections such as `.text`, `.data`, `.bss`, `.debug`, and `.others` show relatively modest increases, typically around 10%. This indicates that CHERI's binary size overhead is highly section-specific and mainly driven by pointer-intensive structures and relocation information. Interestingly, the `.rodata` section decreases in size by about 19% in both the purecap and purecap benchmark modes, partially offset by the growth of the total binary size. Finally, we observe only a minor difference between the purecap and purecap benchmark modes in the `.got+.got.plt` section.

4.3. Impact of Application Workload Characteristics

The CHERI introduced overhead also varies depending on the characteristics of the application workload. As anticipated, memory-sensitive workloads generally experience higher performance overhead in purecap modes than compute-sensitive ones. However, we found that *some computation-sensitive workloads can also experience significant slowdown*.

Memory-intensive Workloads: For `520.omnetpp_r` (a memory intensive workload), execution time increases by 74% in the purecap benchmark configuration and by 87% in the purecap configuration, relative to hybrid. Similarly, `SQLite` shows a 55.33% (purecap benchmark) and 61.16% (purecap) increase. The impact is even more pronounced for `523.xalanchmk_r`, where the execution time rises by 45.45% (purecap benchmark) and 103.5% (purecap). The primary reason for this degradation is the increased memory footprint caused by CHERI's 128 bit capabilities. The larger size of pointers and pointer-based data structures places greater demand on the memory hierarchy, including the L1D and L2D caches and TLBs, and increases memory bandwidth usage. As a result, workloads that are already constrained by memory access experience more severe slowdowns.

Compute-intensive Workloads: For execution time, the compute-intensive `531.deepsjeng_r` shows an increase of 9.2% in the purecap benchmark and 17% in the purecap. `541.leela_r` increases by 14% (purecap benchmark) and 23.1% (purecap), which aligns with the expectation of low overhead for compute-bound tasks. In contrast, `QuickJS` exhibits somewhat higher overhead for a compute-intensive workload, with execution time rising by 165.9% (purecap). This indicates that even compute-sensitive workloads can incur noticeable overhead when their memory access patterns are affected by CHERI's changes, such as increased L1I, L2D, and TLB misses observed during execution.

Other Findings: An interesting finding is `519.lbm_r` (compute), which shows an acceleration in purecap modes (execution time decreases by 7.73% in the purecap-benchmark and 7.89% in purecap). The similar phenomenon is also shown in the `LLaMA.cpp matmul` benchmark, with around 1.3% speed-up in the purecap benchmark and purecap. Even for the `LLaMA.cpp inference` benchmark (i.e., end-to-end text generation case), only the 1.29% overhead is introduced by the CHERI feature. This implies that the impact of CHERI is not universally detrimental and may involve intricate microarchitectural interactions or advantageous side effects resulting from modified data layouts or compiler behavior, which, in certain instances, can lead to enhancements in performance. This highlights the need for detailed characterization rather than relying on broad generalizations.

4.4. Top-Down Bottleneck Analysis

To understand where CPU cycles are being spent or wasted, we follow the approach described in §3.1, grouping the CPU pipeline slots into *Retiring*, *Bad Speculation*, *Frontend* and *Backend* Bounds. The results are given in Figure 3 and Table 4.

For memory-intensive `520.omnetpp_r`, there is a clear shift: Frontend Bound decreases (5.5% to 3%), while Backend Bound increases (67.8% to 70.7%) from hybrid to purecap. This strongly suggests that the primary performance bottleneck under purecap is exacerbated in the backend due to execution unit stalls related to CHERI operations. The Retiring percentage remains roughly similar, indicating the core is still doing a comparable amount of "useful" work relative to stalls, but the overall execution is slower due to the increased Backend Bound stalls.

TABLE 3: Aggregated key performance metrics for representative benchmarks

Hybrid Benchmark Purecap *ABIs*	**510.parest_r**	**519.lbm_r**	**520.omnetpp_r**	**523.xalancbmk_r**	**531.deepsjeng_r**	**541.leela_r**	**544.nab_r**	**557.xz_r**	**LLaMA inference**	**LLaMA matmul**	**SQLite**	**QuickJS**
Execution Time	37.87 41.94 43.10	38.00 35.06 35.0	81.73 142.30 153.21	53.59 77.95 109.07	67.42 73.64 78.85	97.01 110.59 119.46	99.03 103.39 103.92	46.93 49.65 49.98	477.93 483.79 484.11	126.31 124.57 124.61	18.18 28.24 29.30	22.51 NA 59.87
IPC	1.691 1.634 1.599	.929 1.112 1.108	.578 .554 .516	1.813 1.605 1.188	1.702 1.635 1.539	1.406 1.402 1.295	1.538 1.515 1.516	1.091 1.059 1.037	1.827 2.049 2.055	2.306 2.329 2.324	1.579 1.366 1.299	1.612 NA 1.182
Branch Prediction MR (%)	.90 .77 .76	1.52 1.53 1.46	2.80 1.95 2.20	.44 .39 .53	2.99 3.00 2.99	7.36 7.25 7.22	1.16 1.12 1.14	5.56 5.52 5.48	.04 .05 .05	.04 .03 .04	.91 1.04 1.10	1.79 NA 1.68
L1I Cache MR (%)	.05 .12 .11	.15 .16 .15	.35 .57 .70	.98 1.32 .98	.03 .13 .13	.01 .09 .05	0 0 0	.07 .09 .09	.02 .02 .02	0 0 0	4.29 4.40 4.54	1.17 NA 1.67
L1D Cache MR (%)	2.65 2.72 2.73	19.71 22.75 22.29	4.55 4.65 4.46	.62 1.21 1.04	.43 .49 .47	.55 .61 .61	1.28 1.29 1.31	1.88 1.88 1.88	2.05 1.99 1.97	.96 .97 .96	1.70 2.08 2.11	1.06 NA 1.61
L2D Cache MR (%)	5.75 5.64 5.64	12.32 9.50 9.52	27.74 25.42 25.59	.41 2.06 2.42	22.98 19.15 18.46	1.93 3.31 3.39	1.85 1.88 1.83	22.63 22.24 22.08	6.74 7.15 7.14	.32 .34 .35	2.55 3.77 3.77	2.49 NA 5.39
LLC Read MR (%)	99.19 99.27 99.38	99.17 99.09 98.96	92.88 96.10 95.95	94.52 96.29 96.00	97.77 97.45 97.72	96.24 96.38 96.45	97.58 99.18 97.95	96.95 96.32 96.53	67.21 67.49 66.93	92.56 92.51 92.13	95.16 93.86 94.20	91.31 NA 96.39
Capability Load Density (%)	.10 7.78 7.77	.03 1.63 .06	.07 61.44 60.64	.08 82.58 80.72	.01 28.56 27.88	.25 25.15 24.26	.01 23.23 24.23	.50 12.88 12.32	.05 .29 .18	.04 .12 .10	17.38 49.81 49.74	2.69 NA 56.98
Capability Store Density (%)	1.20 25.12 25.10	.02 1.45 .06	.10 78.89 78.12	.10 114.48 111.66	.02 41.11 40.88	.93 65.63 64.57	.04 14.35 14.85	.81 17.16 16.21	.99 3.55 2.24	1.96 2.84 2.22	23.27 63.58 63.68	4.82 NA 91.43
Capability Traffic Share (%)	.16 8.74 8.73	.03 1.57 .06	.09 64.57 64.64	.08 73.72 72.48	.01 31.02 30.66	.49 35.17 34.51	.02 22.86 23.76	.67 15.44 14.70	.08 .40 .26	.06 .14 .12	18.59 48.73 49.01	3.48 NA 59.76
Capability Tag Overhead (%)	.05 9.87 9.84	.02 1.37 .05	.06 66.75 66.82	.07 73.55 72.22	.01 24.12 23.38	.05 25.81 25.02	.01 29.95 30.84	.54 15.58 15.01	.04 .34 .21	.04 .17 .14	15.03 43.90 44.07	2.26 NA 53.43

TABLE 4: Top-down analysis breakdown for six selected workloads

Hybrid, Benchmark, Purecap *ABIs*	**519.lbm_r**	**520.omnetpp_r**	**541.leela_r**	**LLaMA.cpp inference**	**SQLite**	**QuickJS**
Execution Time	38.00,35.06,35.09	81.73,142.30,153.21	97.01,110.59,119.46	477.93,483.79,484.11	18.18,28.24,29.30	22.51, NA, 59.87
Speedup	1, 1.0838, **1.082**	1, 0.574, **0.533**	1, 0.877, 0.812	1, 0.987, **0.987**	1, 0.643, 0.620	1, NA, **0.375**
IPC	0.929, 1.113, 1.109	0.578, 0.578, 0.516	1.406, 1.402, 1.295	1.827, 2.049, 2.055	1.579, 1.366, 1.299	1.612, NA, 1.182
Retiring	0.546, 0.552, 0.537	0.556, 0.553, 0.556	0.531, 0.527, 0.529	0.503, 0.504, 0.503	0.525, 0.524, 0.527	0.535, NA, 0.536
Bad Spec	0, 0, 0	0.0, 0.0, 0.0	0.142, 0.138, 0.118	0.0, 0.042, 0.044	0.074, 0.028, 0.01	0.088, NA, 0.0
Frontend Bound	0.024, 0.034, 0.033	0.055, 0.034, 0.03	0.088, 0.064, 0.071	0.006, 0.005, 0.006	0.153, 0.12, 0.119	0.094, NA, 0.071
Backend Bound	0.684, 0.615, 0.617	0.678, 0.705, 0.707	0.239, 0.271, 0.282	0.499, 0.449, 0.446	0.249, 0.328, 0.344	0.283, NA, 0.399
—+ Memory Bound	0.562, 0.308, 0.308	0.368, 0.338, 0.347	0.088, 0.092, 0.098	0.331, 0.212, 0.212	0.112, 0.145, 0.154	0.115, NA, 0.157
— — – L1 Bound	0.029, 0.015, 0.015	0.015, 0.014, 0.014	0.015, 0.011, 0.012	0.005, 0.003, 0.003	0.008, 0.011, 0.011	0.011, NA, 0.01
— — – L2 Bound	0.024, 0.01, 0.01	0.027, 0.024, 0.025	0.002, 0.002, 0.003	0.01, 0.007, 0.007	0.004, 0.006, 0.006	0.006, NA, 0.008
— — – ExtMem Bound	0.51, 0.283, 0.283	0.326, 0.3, 0.308	0.072, 0.078, 0.083	0.317, 0.203, 0.202	0.1, 0.128, 0.136	0.097, NA, 0.138
—+ Core Bound	0.121, 0.306, 0.308	0.31, 0.367, 0.36	0.151, 0.179, 0.184	0.168, 0.236, 0.235	0.137, 0.183, 0.19	0.168, NA, 0.242

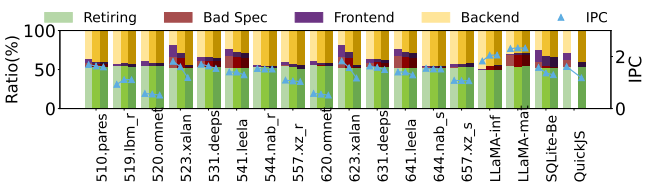

Figure 3: Top-level breakdown analysis
Note: Instructions are classified into four distinct categories: Retiring(%), Bad Speculation(%), Frontend(%), and Backend(%), which collectively sum to one.

For compute-intensive `541.leela_r`, the changes are more subtle. Frontend Bound decreases from 8.8% to 7.1%, while Backend Bound sees a small increase (4.3%). The Retiring percentage sees a slight dip. This indicates a less dramatic shift in bottlenecks compared to memory-intensive workloads. This conforms to the majority of workloads.

In examining the workloads, transitioning from Memory Bound to Core Bound, as illustrated in references `519.lbm_r` and `LLaMA.cpp`, can result in either enhanced performance or no noticeable change. This drives the compiler optimization

for programs executed on the CHERI platform to minimize instructions associated with memory-bound.

`QuickJS` sequentially executed 18,612 JavaScript programs, resulting in a high ExtMemory Bound and a significant performance drop, with execution time increasing from 22.51s to 59.87s. During this period, its memory footprint grew by 36.3%, while *utilized memory* increased by 55.24% from hybrid to purecap mode.

4.5. Frontend Dynamics and Branch Prediction

The frontend of the processor is responsible for fetching and decoding instructions. CHERI features can impact this through changes in code size or by interacting with branch prediction mechanisms, especially concerning the PCC.

Frontend Stall Rates: the Frontend Stall Rate often *decreases* in purecap mode for memory-intensive benchmarks (e.g., from 5.4% to 2.9% in `520.omnetpp_r`) or remains stable for compute-intensive ones (e.g., 4% in `557.xz_r`). This suggests that for many workloads in Morello, the primary bottleneck introduced by CHERI is not in the Frontend.

Branch Misprediction Rates: The Branch Misprediction Rate in Table 3 generally shows little change across ABIs for most benchmarks (e.g., `531.deepsjeng_r`, `541.leela_r`, `557.xz_r`, `LLaMA.cpp`). This implies that the branches themselves are not inherently harder to predict or a core bottleneck in the CHERI-oriented optimization for these workloads.

Morello's PCC Branch Prediction Limitation: A key factor in Morello's performance is its branch predictor's lack of full awareness of PCC bounds changes. This can cause stalls when PCC bounds are modified, such as during interlibrary function calls/returns or virtual method calls. The purecap-benchmark ABI was designed to mitigate this specific issue by using global PCC bounds and integer jumps. The significant performance improvement observed when moving from purecap to purecap-benchmark ABI (e.g., overhead dropping from 28.01% to 14.97% in [36]; also been validated in `520.omnetpp_r`, 523.xalanchmk_r, and 541.leela_r.) strongly indicates that a substantial portion of the overhead in the standard purecap ABI is due to these PCC-related stalls.

For example, in `520.omnetpp_r`, the Frontend Stall Rate is 3.4% for the purecap-benchmark versus 2.9% for purecap, which is a slight decrease, indicating that alternative factors may be contributing. However, `520.omnetpp_r` is overall faster to run in the purecap benchmark than in purecap (142.3s vs 153.2s). This difference highlights the benefit of the purecap benchmark ABI in reducing overheads, part of which is likely due to improved frontend behavior. This analytical separation is crucial: This allows us to estimate that a CHERI implementation with a capability-aware branch predictor would or would not experience overhead for the specific workload.

4.6. Backend Stalls and Execution Characteristics

Backend stalls occur when the execution units or memory subsystem fail to keep up with the Frontend. As shown in Figure 4, the increased Backend stalls in purecap modes

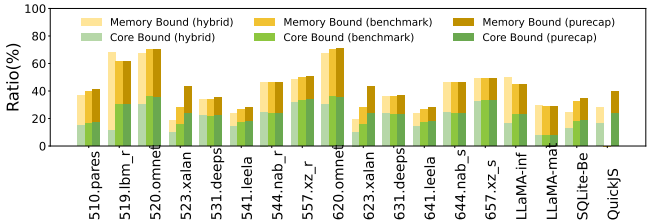

Figure 4: Percentage of counters pertaining to core and memory bounds

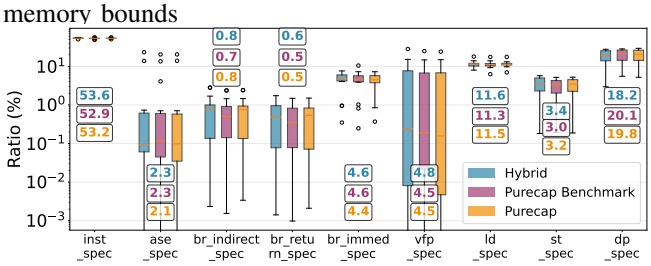

Figure 5: The distribution of speculative instruction ratios across benchmarks by ABIs

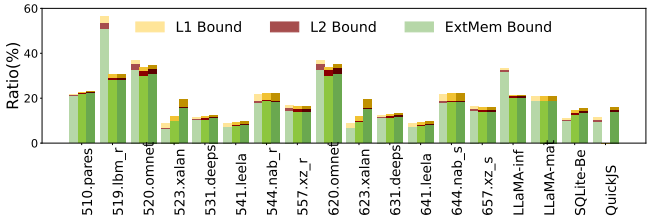

Figure 6: Memory-bound analysis (from cache and DRAM)

are primarily driven by memory hierarchy effects, detailed in §4.7 - notably higher L1I, L1D, L2D, and TLB miss rates.

Our analysis of the dynamic instruction mix (see Figure 5) reveals a notable shift under purecap. In particular, the proportion of data processing instructions (`DP_SPEC`) increases substantially, ranging from 5.21% to 29.31%, due to the extra arithmetic operations required for capability manipulation and bounds checking. In contrast, the proportions of load (`LD_SPEC`) and store (`ST_SPEC`) instructions remain relatively stable, with standard deviations of 2.01% and 1.47%, respectively. This suggests that memory access patterns are less affected than computational demands. The shift in instruction mix underscores the microarchitectural cost of CHERI's security model and identifies potential areas for optimisation.

4.7. Memory Hierarchy under CHERI

The CHERI architecture's 128-bit capabilities fundamentally alter the memory footprint of applications, especially those that are pointer-intensive or manage large pointer-based data structures. This has direct consequences for cache and TLB performance, shown in Figure 6.

Cache Performance: The doubling of pointer size from 64 bit to 128 bit increases the memory footprint of data structures that include pointers, such as arrays or pointer-rich objects on the stack or heap. This expansion reduces spatial locality for a fixed cache size, as fewer logical elements fit within a cache line or cache level, leading to higher

cache miss rates. L1 instruction cache behaviour can be affected by CHERI-induced changes in code generation, including additional instructions for capability handling and modified code layout. For example, in `520.omnetpp_r` and `QuickJS`, the L1I miss rate increases from 0.35% to 0.7% and from 1.17% to 1.67%, respectively. However, benchmarks such as `523.xalancbmk_r` and `SQLite` show negligible change, suggesting limited impact on instruction fetch performance. L1 data cache miss rates show no consistent correlation with overall performance. For instance, `519.lbm_r` sees increased L1D misses yet improved performance, while `520.omnetpp_r` experiences a performance drop despite stable L1D miss rates when moving from hybrid to pure capability mode. The L2 unified cache often absorbs the pressure from increased L1 misses. In `541.leela_r` and `QuickJS`, L2 data miss rates rise by 75.6% and 116.4%, respectively, accompanied by performance reductions of 23.14% and 165.97%. Conversely, `519.lbm_r` and `531.deepsjeng_r` show lower L2 miss rates under purecap, possibly due to changes in memory access behaviour or improved cache replacement effects. Last level cache (LLC) miss rates remain extremely high in almost all cases, typically above 90% in both hybrid and purecap modes—for example, `LLaMA.cpp matmul` records 92.56% (hybrid) and 92.13% (purecap), while `SQLite` shows 95.16% and 94.2%. These results indicate that memory-bound analysis may be more useful for identifying performance bottlenecks than LLC miss rates alone.

TLB Performance: The increased memory footprint introduced by capabilities can place additional pressure on the TLB. When applications access more distinct memory pages due to data expansion or changes in allocation strategies (for example, to satisfy capability alignment), TLB miss rates and page table walks tend to rise. The `DTLB_WALK` event counts data accesses that trigger a page table walk after a miss in the Data TLB (DTLB). Most benchmarks show stable values across the three ABIs. However, `523.xalancbmk_r`, `523.leela_r`, and `544.nab_r` show increases of 1170%, 324%, and 62%, respectively. In contrast, some cases show reductions of 103.52%, 23.4%, and 4.93%. The L1D TLB miss rate shows no consistent pattern with overall performance across the three ABIs. It only shows a notable decrease in `520.omnetpp_r` and an increase in `523.xalancbmk_r`, both linked to performance degradation. The L2D TLB miss rate remains mostly unchanged across the ABIs. However, the total counts of `TLB` and `TLB_REFILL` events increase - and in some cases double - when moving from hybrid to purecap, even though the miss rate stays the same. These TLB and cache effects reflect CHERI's expanded memory footprint and contribute to the increased `Backend Bound` stall percentages discussed in §3.1.

4.8. CHERI-Specific Events

We leverage the CHERI-specific PMU events supported by Morello (§3.2) to examine the CHERI mechanisms during runtime.

Quantifying Capability-Based Memory Traffic: By comparing `CAP_MEM_ACCESS_RD` with `LD_SPEC`, we can calculate

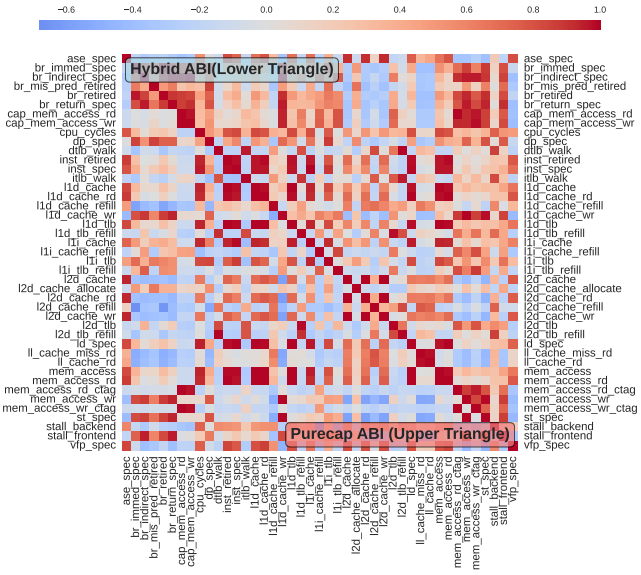

Figure 7: Performance correlation matrix (hybrid vs purecap)

the proportion of read-memory traffic that is explicitly capability-based. Similarly for writes. This ratio provides a direct measure of the capability load density for a given workload and ABI. For instance, this ratio in purecap mode is expected to be higher than that in hybrid mode. A higher proportion of capability accesses increases sensitivity to the overheads of wider pointers and capability checks. For example, in `QuickJS` and `523.xalancbmk_r`, the capability-based load rises from 2.69% (hybrid) to 56.98% (purecap) and 0.08% (hybrid) to 80.72% (purecap), leading to notable performance declines - 165.97% and 103.52%.

CHERI-specific Events and Bottlenecks: The observed increase in `CAP_MEM_ACCESS_RD` aligns with the rise in `L1I MR`, supporting the view that performance degradation is directly influenced by the volume of capability memory operations introduced by CHERI. Although the `MEM_ACCESS_RD_CTAG` event counts tag-dependent memory accesses, it does not directly capture the latency of tag checks, which are likely pipelined and integrated with memory access operations. Nonetheless, a high frequency of these events indicates frequent reliance on CHERI's core memory protection mechanisms. More broadly, the safety guarantees and capability manipulations introduced by CHERI create a tightly coupled execution pattern that links instruction-level events and memory system behaviour. This is reflected in the stronger correlations observed among metrics such as cache refills, TLB walks, and stall cycles, as illustrated in Figure 7. In purecap mode, `CAP_MEM_ACCESS_RD/WR` events show strong associations with `L2D_TLB`, `L1D_CACHE_REFILL`, and `L1I_TLB`, suggesting that effective performance tuning may require a holistic approach, as changes in one metric often propagate to others.

By leveraging CHERI-specific PMC events, the analysis shifts from relying on derived metrics such as cache miss rates to directly observing and quantifying CHERI's operational effects within the memory system. This creates a

clearer link between CHERI's architectural features and their runtime performance impact, offering a novel perspective in this workload characterization.

5. Discussion

Our study highlights a complex performance landscape for CHERI. While CHERI enforces strong memory safety through hardware capabilities, it can introduce non-uniform performance overheads that should be assessed as workloads evolve. A major source of overhead is the increased memory footprint: replacing 64-bit pointers with 128-bit capabilities enlarges data structures, raising L1I and L2D cache miss rates and increasing TLB pressure as applications access more pages or exhibit weaker locality.

The impact is highly application dependent. For example, `LLaMA.cpp`, an LLM inference workload, shows only a 1.29% overhead in purecap mode, with matrix multiplication even gaining a small speed-up. This challenges the expectation that memory-bound workloads suffer most from larger footprints. Our top-down analysis shows that `LLaMA.cpp` becomes less memory-bound (33.1% to 21.2%) and more core-bound, as sequential reads dominate its access patterns. In contrast, QuickJS incurs a 165% overhead in purecap mode. Although classified as compute-intensive, it executes more than 18,000 small JavaScript programs in sequence, with parsing, object allocation, execution, and teardown operations repeatedly stressing the L1I cache and the instruction and data TLBs.

Branch predictor stalls also contribute to overhead. Frequent PCC-bound changes during interprocedural control flow and virtual calls cause frontend inefficiencies. The purecap benchmark ABI reduces this effect for some SPEC benchmarks (e.g., `523.xalancbmk_r`, `541.leela_r`, `520.omnetpp_r`), but has little impact on real-world applications such as `LLaMA.cpp` and `SQLite`, highlighting the need for program-specific tuning. Despite pointer size extensions and permission checks, binary sizes do not increase significantly, supporting CHERI's feasibility for resource-constrained systems.

6. Related Work

Software Solutions for Memory Safety. Languages such as C#, Go, Java [32], Python, Rust [17], and Swift improve memory safety through strong typing, bounds checks, and compile-time enforcement [19]. They prevent common errors (e.g., buffer overflows, use-after-free) but remain vulnerable to logic flaws and insecure coding. To improve safety in unsafe languages, Apple modified its C compiler for iBoot [1], while Microsoft's *Checked C* adds static and dynamic checks [8, 29].

Hardware-Assisted Memory Safety. Vendors have introduced hardware features to enforce safety at runtime. Arm's Memory Tagging Extension (MTE) [3] detects use-after-free and out-of-bounds accesses with memory tags, while Intel's Control-flow Enforcement Technology (CET) [15] enforces control-flow integrity. Isolation-based schemes include Intel SGX [16, 28] and Arm TrustZone [25], though both suffer overhead from context switches. In contrast, CHERI enforces fine-grained spatial and temporal safety using tagged pointers without context-switching costs, and recent work extends it to accelerators [9].

CHERI Capabilities. CHERI introduces capability-based hardware enforcement of memory safety. Tools such as ESBMC-CHERI [5] enable program verification, while designs like *Cheri Concentrate* [40] reduce pointer overhead. Extensions such as Capability Speculation Contracts (CSC) [13] and Cornucopia Reloaded [12] target speculative execution and temporal safety. On the software side, large systems, including the Linux kernel, are being ported to CHERI-enabled RISC-V platforms [34], advancing ecosystem maturity.

CHERI Performance Characterization. Performance evaluations of CHERI remain limited. An early study [36] reported high-level results for SPEC CPU 2006 on Arm Morello. Our work expands this by analyzing a broader set of applications across execution modes, using hardware counters for detailed microarchitectural insights. This large-scale study provides a more complete picture of CHERI's overheads and implications, supporting the design of secure and efficient capability-based systems.

7. Conclusion

We have presented a large-scale empirical study of workload characteristics on the CHERI-enabled Morello platform. Our study leverages the platform's hardware performance monitoring counters (PMUs) to collect a set of microarchitectural event data, from which we derive metrics to analyze the performance of 20 benchmarks across three ABI modes: hybrid, pure-capability, and purecap-benchmark. We take a top-down approach to study the impact of CHERI security features on applications and discuss the potential future directions. Our study provides empirical insights into the performance implications of CHERI's memory safety features and informs the design and optimization of other hardware-assisted memory security mechanisms.

Acknowledgments

This work was supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant agreements EP/X037304/1 and EP/X037525/1.

References

[1] Apple Support, "Memory safe iboot implementation." [Online]. Available: https://support.apple.com/en-gb/guide/security/sec30d8d9ec1/web/

[2] ARM, "Arm neoverse n1 performance analysis methodology." [Online]. Available: https://developer.arm.com/documentation/109198/latest/

[3] ARM, "Introduction to the memory tagging extension." [Online]. Available: https://developer.arm.com/documentation/108035/0100/Introduction-to-the-Memory-Tagging-Extension

[4] T. Bauereiss, B. Campbell, T. Sewell, A. Armstrong, L. Esswood, I. Stark, G. Barnes, R. N. M. Watson, and P. Sewell, "Verified security for the morello capability-enhanced prototype arm architecture," in *Programming Languages and Systems*, I. Sergey, Ed. Cham: Springer International Publishing, 2022, pp. 174–203.

[5] F. Brauße, F. Shmarov, R. Menezes, M. R. Gadelha, K. Korovin, G. Reger, and L. C. Cordeiro, "Esbmc-cheri: towards verification of c programs for cheri platforms with esbmc," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 773–776.

[6] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[7] C. Burth, M. Velten, and R. Schöne, "Introducing the arm-membench throughput benchmark," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski, Eds. Cham: Springer Nature Switzerland, 2025, pp. 99–112.

[8] Checked C, "An extension to c for making existing c code more secure." [Online]. Available: https://github.com/checkedc/

[9] J. Cheng, A. T. Markettos, A. Joannou, P. Metzger, M. Naylor, P. Rugg, and T. M. Jones, "Adaptive cheri compartmentalization for heterogeneous accelerators," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 2002–2016. [Online]. Available: https://doi.org/10.1145/3695053.3731062

[10] Cheribuild.py, "a script to build cheri-related software." [Online]. Available: https://github.com/CTSRD-CHERI/cheribuild

[11] Chromium Security, "Memory safety." [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/

[12] N. W. Filardo, B. F. Gutstein, J. Woodruff, J. Clarke, P. Rugg, B. Davis, M. Johnston, R. Norton, D. Chisnall, S. W. Moore, P. G. Neumann, and R. N. M. Watson, "Cornucopia reloaded: Load barriers for cheri heap temporal safety," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 251–268. [Online]. Available: https://doi.org/10.1145/3620665.3640416

[13] F. A. Fuchs, J. Woodruff, P. Rugg, A. Joannou, J. Clarke, J. Baldwin, B. Davis, P. G. Neumann, R. N. M. Watson, and S. W. Moore, "Safe speculation for cheri," in *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, 2024, pp. 364–372.

[14] R. Grisenthwaite, G. Barnes, R. N. M. Watson, S. W. Moore, P. Sewell, and J. Woodruff, "The arm morello evaluation platform—validating cheri-based security in a high-performance system," *IEEE Micro*, vol. 43, no. 3, pp. 50–57, 2023.

[15] Intel, "Complex shadow-stack updates (intel control-flow enforcement technology." [Online]. Available: https://www.intel.com/content/www/us/en/content-details/785687/complex-shadow-stack-updates-intel-control-flow-enforcement-technology.html

[16] Intel, "Intel software guard extensions (intel sgx)." [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html

[17] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Safe systems programming in rust," *Communications of the ACM*, vol. 64, no. 4, pp. 144–152, 2021.

[18] J. Kalyanasundaram and Y. Simmhan, "Arm wrestling with big data: A study of commodity arm64 server for big data workloads," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, 2017, pp. 203–212.

[19] L. Liu, T. Millstein, and M. Musuvathi, "Safe-by-default concurrency for modern programming languages," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 43, no. 3, pp. 1–50, 2021.

[20] LLaMA.cpp, "Llm inference in c/c++." [Online]. Available: https://github.com/ggml-org/llama.cpp/

[21] F. Mantovani, M. Garcia-Gasulla, J. Gracia, E. Stafford, F. Banchelli, M. Josep-Fabrego, J. Criado-Ledesma, and M. Nachtmann, "Performance and energy consumption of hpc workloads on a cluster based on arm thunderx2 cpu," *Future generation computer systems*, vol. 112, pp. 800–818, 2020.

[22] Microsoft, "A proactive approach to more secure code. microsoft security response center (msrc) blog." [Online]. Available: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

[23] S. Miksits, R. Shi, M. Gokhale, J. Wahlgren, G. Schieffer, and I. Peng, "Multi-level memory-centric profiling on arm processors with arm spe," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2024, pp. 996–1005.

[24] Mozilla, "Implications of rewriting a browser component in rust." [Online]. Available: https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/

[25] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Comput. Surv*, vol. 51, no. 6, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3291047

[26] P. M. Protection, "The risc-v instruction set manual: Volume ii." [Online]. Available: https://riscv.org/specifications/ratified/

[27] QuickJS, "embeddable javascript engine." [Online]. Available: https://github.com/bellard/quickjs

[28] L. Regano and D. Canavese, "Automated intel SGX integration for enhanced application security," *IEEE Access*, vol. 12, pp. 110 312–110 321, 2024. [Online]. Available: https://doi.org/10.1109/ACCESS.2024.3441240

[29] M. S. Simpson and R. K. Barua, "Memsafe: ensuring the spatial and temporal memory safety of c at runtime," *Software: Practice and Experience*, vol. 43, no. 1, pp. 93–128, 2013.

[30] SQLite, "a small, fast, self-contained, high-reliability, full-featured, sql database engine." [Online]. Available: https://sqlite.org/

[31] SQLite Speedtest, "SQLite's speedtest1 benchmark program with an increased problem size of 1000." [Online]. Available: https://openbenchmarking.org/test/pts/sqlite-speedtest

[32] C. Stancu, C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz, "Safe and efficient hybrid memory management for java," *ACM SIGPLAN Notices*, vol. 50, no. 11, pp. 81–92, 2015.

[33] Test262, "Ecmascript test suite (ecma tr/104)." [Online]. Available: https://github.com/tc39/test262

[34] 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*, ser. EuroSec '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 15–21. [Online]. Available: https://doi.org/10.1145/3642974.3652282

[35] R. N. M. Watson, D. Chisnall, D. Brooks, K. Wojciech, W. M. Simon, J. M. Steven, G. N. Peter, and W. Jonathan, "Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-877, Sep. 2015. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-877.pdf

[36] 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," University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, Tech. Rep. UCAM-CL-TR-986, September 2023.

[37] R. N. M. Watson, S. W. Moore, P. Sewell, and P. G. Neumann, "An Introduction to CHERI," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-941, Sep. 2019. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf

[38] R. N. M. Watson, J. W. Peter G. Neumann, M. Roe, H. Almatary, J. B. Jonathan Anderson, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, F. A. Fuchs, R. Grisenthwaite, B. L. Alexandre Joannou, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-987, Sep. 2023. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf

[39] R. N. M. Watson, A. Richardson, B. Davis, J. Baldwin, D. Chisnall, J. Clarke, N. Filardo, S. W. Moore, E. Napierala, P. Sewell, and P. G. Neumann, "CHERI C/C++ Programming Guide," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-947, Jun. 2020. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf

[40] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. M. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, and A. T. Markettos, "Cheri concentrate: Practical compressed capabilities," *IEEE Transactions on Computers*, vol. PP, no. 10, pp. 1–1, 2019.

[41] X.Sun, "The pmcstat tool might cause a crash on cheribsd/morello." [Online]. Available: https://github.com/CTSRD-CHERI/cheribsd/issues/2391

[42] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 35–44.

Appendix-I: The Compilation and Execution Status of Benchmark Suites

This appendix presents a thorough summary of the resolution of compilation issues for each benchmark suite. It offers an in-depth analysis of the underlying causes of compilation failures and errors and details the source code modifications undertaken to enable successful execution.

TABLE 5: Resolution of Compilation Errors in SPECCPU 2017 (I)

Benchmark	Description	Language	Status
520.omnetpp_r	Discrete event simulation of a large 10 gigabit Ethernet network	C++	✔. Compiled. (Add ”stdio.h” header to ”simulator/platdep/platmisc.h” source file for invoking ”sprintf...” function; Set ”-D__LONG_LONG_SUPPORTED” in SPEC compilation configuration). Run successfully under three ABIs.
523.xalancbmk_r	The XSLT processor for transforming XML documents into HTML, text, or other XML document types	C++	✔. Compiled. (Comment ”#include ¡linux/limits.h¿” in ”.../cheribsd/tools/build/cross-build/include/linux/limits.h”; Compile xerces-c v2.7.0 on CheriBSD-Morello and link this library during cross-compilation; Replace ftime() function using gettimeofday(); Congiure ”-D_GNU_SOURCE” in cross-compilation). Run successfully under three ABIs.
525.x264_r	Video compression	C	✔. Compiled. (Configure ’-fcommon’ in SPEC compilation; Fix copying errors when setup ”run” directory through explicit copy operations.)
531.deepsjeng_r	Alpha-beta tree search & pattern recognition	C++	✔. Compiled.
541.leela_r	Monte carlo simulation, game tree search & pattern recognition	C++	✔. Compiled.
557.xz_r	Data compression based on Lasse Collin’s XZ utils 5.0.5	C	✔. Compiled.
510.parest_r	A finite element solver for a biomedical imaging problem	C++	✔. Compiled.
519.lbm_r	Lattice Boltzmann Method (LBM) to simulate incompressible fluids in 3D	C	✔. Compiled.
544.nab_r	A molecular modeling application based on Nucleic Acid Builder (NAB)	C	✔. Compiled.
500.perlbench_r	The cut-down version of Perl v5.22.1	C	Not Compiled. The ’struct_FILE.h’ header file is not compatible/supported.
502.gcc_r	C Language optimizing compiler based on GCC Version 4.5.0	C	Compiled. (An in-address-space security exception was triggered under both the purecap and benchmark ABIs, whereas the hybrid ABI executed without errors.)
505.mcf_r	Combinatorial optimization / single-depot vehicle scheduling	C	Compiled. (An in-address-space security exception was triggered under both the purecap and benchmark ABIs, whereas the hybrid ABI executed without errors.)
548.exchange2_r	One game - sudoku puzzle generator	Fortran	Not Compiled. (Fortran is not supported.)
503.bwaves_r	The numerical simulation on blast waves in three dimensional transonic transient laminar viscous flow	Fortran	Not Compiled. (Fortran is not supported.)
507.cactuBSSN_r	Utilizing the EinsteinToolkit to solve the Einstein equations in vacuum	Fortran, C++, C	Not Compiled. (Fortran is not supported.)
508.namd_r	A parallel program for the simulation of large biomolecular systems	C++	Compiled. (An in-address-space security exception was triggered under both the purecap and benchmark ABIs, whereas the hybrid ABI executed without errors.)
511.povray_r	A free and open source ray-tracing application	C++, C	Compiled. (An in-address-space security exception was triggered under both the purecap and benchmark ABIs, whereas the hybrid ABI executed without errors.)
521.wrf_r	Weather forecasting	Fortran	Not Compiled. (Fortran is not supported.)
526.blender_r	A free and open source 3D creation suite	C++, C	Not Compiled. (The ’alloca.h’ header file is not found/supported.)
527.cam4_r	One atmosphere general circulation model	Fortran, C	Not Compiled. (Fortran is not supported.)
538.imagick_r	Image manipulation	C	Compiled. (An in-address-space security exception was triggered under both the purecap and benchmark ABIs, whereas the hybrid ABI executed without errors.)
549.fotonik3d_r	Computational electromagnetics	Fortran	Not Compiled. (Fortran is not supported.)
554.roms_r	A regional ocean modeling system	Fortran	Not Compiled. (Fortran is not supported.)

Additional Issue Corrections:

- In the event that sampling is paused as referenced in `pmcstat`, the configurations are set *kern.hwpmc.nsamples=”8192”* and *kern.hwpmc.nbuffers_pcpu=”256”*.
- To successfully execute the *speedtest1* SQL benchmark in SQLite, it is necessary to modify the generated benchmark suite by changing *fprintf (g.pScript,”%s”, z)* to *fprintf(g.pScript,”%s;”, z)*.
- To achieve successful cross-compilation of QuickJS, it is necessary to generate the *reply.c* file on the remote device during the compilation stage, utilizing the intermediate generated binaries produced. The binary compiled using the benchmark ABI is unable to execute *test262* successfully due to an in-address security issue.

TABLE 6: Resolution of Compilation Errors in SPECCPU 2017 (II)

Benchmark	Description	Language	Status
620.omnetpp_s	Discrete event simulation of a large 10 gigabit Ethernet network	C++	✔. Compiled. (Add ”¡stdio.h¿” header to ”simulator/platdep/platmisc.h” source file for invoking ”sprintf...” function; Configure ”-D__LONG_LONG_SUPPORTED” in compilation.)
623.xalancbmk_s	The XSLT processor for transforming XML documents into HTML, text, or other XML document types	C++	✔. Compiled. (Comment ”#include ¡linux/limits.h¿” in ”.../cheribsd/tools/build/cross-build/include/linux/limits.h”; Compile xerces-c v2.7.0 on CheriBSD-Morello and link this library during compilation; Replace ftime() function using gettimeofday(); Congiure ”-D_GNU_SOURCE” in compilation.)
631.deepsjeng_s	Alpha-beta tree search & pattern recognition	C++	✔. Compiled.
641.leela_s	Monte carlo simulation, game tree search & pattern recognition	C++	✔. Compiled.
657.xz_s	Data compression based on Lasse Collin’s XZ utils 5.0.5	C	✔. Compiled. (Remove SPEC default configuration ”-fopenmp -DSPEC_OPENMP” and reimplement parallel part using pthreads instead of OpenMP.)
625.x264_s	Video compression	C	✔. Compiled. (Fix copying errors when setup ”run” directory through explicit copy operations.) (Profiling this program found a bug in CheriBSD [41].)
644.nab_s	A molecular modeling application based on Nucleic Acid Builder (NAB)	C	✔. Compiled. Reimplement parallel part using pthreads, instead of OpenMP
600.perlbench_s	The cut-down version of Perl v5.22.1	C	Not Compiled. (The ’struct_FILE.h’ header file is not compatible/supported.)
602.gcc_s	C Language optimizing compiler based on GCC Version 4.5.0	C	Compiled. (An in-address-space security exception was triggered under both the purecap and benchmark ABIs, and ld-elf64 error on hybrid ABI.)
605.mcf_s	Combinatorial optimization / single-depot vehicle scheduling	C	Compiled. (An in-address-space security exception was triggered under both the purecap and benchmark ABIs, and ld-elf64 error on hybrid ABI.)
648.exchange2_s	One game - sudoku puzzle generator	Fortran	Not Compiled. (Fortran is not supported.)
603.bwaves_s	The numerical simulation on blast waves in three dimensional transonic transient laminar viscous flow	Fortran	Not Compiled. (Fortran is not supported.)
607.cactuBSSN_s	Utilizing the EinsteinToolkit to solve the Einstein equations in vacuum	Fortran, C++, C	Not Compiled. (Fortran is not supported.)
619.lbm_s	Lattice Boltzmann Method (LBM) to simulate incompressible fluids in 3D	C	Compiled. Reimplement parallel part using pthreads, instead of OpenMP and configure pthreads settings. (An in-address-space security exception was triggered.)
621.wrf_s	Weather forecasting	Fortran	Not Compiled. (Fortran is not supported.)
627.cam4_s	One atmosphere general circulation model	Fortran, C	Not Compiled. (Fortran is not supported.)
628.pop2_s	Wide-scale ocean modeling (climate level)	Fortran, C	Not Compiled. (Fortran is not supported.)
638.imagick_s	Image manipulation	C	Compiled. Reimplement parallel part using pthreads, instead of OpenMP. (An in-address-space security exception was triggered under both the purecap and benchmark ABIs.)
649.fotonik3d_s	Computational electromagnetics	Fortran	Not Compiled. (Fortran is not supported.)
654.roms_s	A regional ocean modeling system	Fortran	Not Compiled. (Fortran is not supported.)