Technical Report

UCAM-CL-TR-984
ISSN 1476-2986

Number 984

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Efficient spatial and temporal safety for microcontrollers and application-class processors

Peter David Rugg

July 2023

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 2022 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Efficient spatial and temporal safety for microcontrollers and application-class processors

Peter David Rugg

Abstract

This thesis discusses the implementation of Capability Hardware Enhanced RISC Instructions (CHERI) secure capabilities for RISC-V microarchitectures. This includes implementations for three different scales of core, including microcontrollers and the first open application of CHERI to a superscalar processor. Tradeoffs in developing the architecture and performant microarchitecture are investigated. The processors are then used as a platform to conduct research in reducing the overheads for achieving temporal safety with CHERI.

CHERI offers a contemporary cross-architecture description of capabilities. The initial design was previously carried out in a single MIPS processor. Based on its success in this context, this thesis investigates the microarchitectural implications across a wider range of processors. To improve adoption, this work is performed on the more contemporary RISC-V architecture. The thesis also explores the microarchitectural implications of architectural decisions arising from the adaptation of CHERI to this new context.

The first implementations are to the Piccolo and Flute microcontrollers. They present new tradeoffs, for example being the first CHERI implementations supporting a merged register file and capability mode bit. The area and frequency implications are evaluated on FPGA, and the performance and power overheads are investigated across a range of benchmarks. To validate correctness, the processors are integrated into a new TestRIG infrastructure.

This thesis also develops the first open instantiation of CHERI for a superscalar out-of-order application-class core: RiscyOO. This presents new questions due to the very different design of the more sophisticated microarchitecture, and highlights more architectural tradeoffs. Again, the processor is evaluated on FPGA, investigating area, frequency, power, and performance. This allows the first analysis of how the overheads scale differently across different sizes of core.

Finally, the augmented processors are used as a platform to refine the use of CHERI for temporal safety. Significant improvements are made to the architecture-neutral model used for revocation sweeps. In addition, processor-specific acceleration of revocation is performed, including new approaches for caching capability tags.

Acknowledgements

I would like to thank my supervisor—Prof. Simon W. Moore—for his patient supervision and invaluable guidance. Thanks also to Prof. Robert N. M. Watson for his advice and for driving the overall CHERI project to make this work possible.

It has been a privilege to be a part of the CHERI team and its exceptional collaborative environment. Jonathan Woodruff, Alexandre Joannou, Jessica Clarke, Franz Fuchs, Ivan Ribeiro, Nathaniel W. Filardo, Hesham Almatary, Marno van der Maas, Lawrence Esswood, A. Theodore Markettos, and everyone else: thank you for always making time for thought-provoking discussions, and for your contributions to the CHERI effort that made my work possible.

To my parents: I do not know what I would have done without your support, particularly since the pandemic. I hope you know how much this means to me.

Thank you also to my friends for being there for me along the way.

As well as my supervisor, thanks in particular to Sarah, Franz, Alexandre, Lawrence, and my parents for help with proofreading that significantly improved this thesis.

Thank you also to my examiners—Prof. David Oswald and Prof. Robert Mullins—for their time and suggestions.

Approved for public release; distribution is unlimited. This work was supported by the Defense Advanced Research Projects Agency (DARPA) under contract HR0011-18-C-0016 ("ECATS"). The views, opinions, and/or findings contained in this dissertation are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Contents

List of Figures

Glossary

Application Binary Interface (ABI)

Interface between interacting code, specifying e.g. calling conventions and register allocations. 54, 56

Arithmetic Logic Unit (ALU)

Part of the processor responsible for executing the arithmetic operations. 13, 52, 54, 60, 62, 64, 65, 68, 70, 85, 86, 87, 98, 106, 107, 108, 113, 114, 115, 118, 124, 125

Application-Specific Integrated Circuit (ASIC)

Synthesised chip for a specialised hardware design. 13, 27, 28, 79, 81, 84, 85, 96, 102, 119, 121, 126, 134, 159

Address Space Layout Randomisation (ASLR)

Security measure that randomises where programs are loaded to prevent static addresses being used to access control data [17]. 35, 37, 101

Advanced eXtensible Interface (AXI)

Arm-specified open on-chip interconnect [7]. 25, 53, 66, 68, 88, 89, 90, 96, 106, 127, 147

BESSPIN

Galois SoC and tools for the Piccolo, Flute, and Toooba cores, provided as part of the System Security Integration Through Hardware and Firmware (SSITH) program [143]. 76, 79, 84, 121

Block Random Access Memory (BRAM)

Specialised FPGA block for random access memory, allowing registers to be synthesised in a much smaller area, but with more restrictive access requirements. 9, 28, 53, 56, 68, 81, 84, 85, 87, 88, 96, 124, 126

Branch Target Buffer (BTB)

Prediction structure that caches the target Program Counters (PCs) of jump or branch instructions so that they can be predicted. 54, 89, 104

Complex Instruction Set Computing (CISC)

Architecture design principle favouring compound instructions to execute common sequences of operations, as an alternative to Reduced Instruction Set Computing (RISC). 15

Core-Local Interrupt Controller (CLINT)

RISC-V-specified module to manage timer interrupts within a core. This is instantiated once per core. 85

Control and Status Register (CSR)

RISC-V registers for controlling system-level functionality, for example the exception vector and exception cause register. 15, 29, 33, 58, 67, 70, 79, 86, 105, 106, 109, 110, 141, 151

Common Weakness Enumeration (CWE)

An open categorisation of common software and hardware errors that often lead to vulnerabilities [30]. 98, 133

Default Data Capability (DDC)

A capability that is implicitly used as the authority for legacy memory accesses. 57, 62, 86, 89, 108, 110

Direct Instruction Injection (DII)

Protocol for injecting instructions directly into a RISC-V processor. 16, 23, 24, 72, 74, 158

Direct Memory Access (DMA)

Peripherals given direct access to main memory to allow them to operate independently from the CPU. 13, 16, 28, 53, 66, 84, 124, 143, 144, 156

Dynamic Random Access Memory (DRAM)

Current standard main memory for application-class cores. 10, 12, 28, 33, 48, 51, 53, 54, 79, 84, 88, 96, 98, 99, 113, 121, 130, 132, 134, 144, 145, 146, 155, 156, 159

Design Under Test (DUT)

The design being tested in a testbench. 72

Error Correction Code (ECC)

Additional bits stored by some (typically server-class) DRAM components to allow detection and recovery from bit-errors. 48

Flip-flop (FF)

A basic storage element that provides single-cycle access at the expense of high area. 81, 84, 85, 122, 124, 125, 126

First In, First Out (FIFO)

A hardware structure describing a queue, where data is consumed in the same order as it was provided. 9, 30, 63, 67, 74

Field-Programmable Gate Array (FPGA)

Device to emulate hardware designs, often used for prototyping as an intermediate between simulation and ASIC synthesis. 9, 11, 13, 14, 22, 27, 28, 51, 68, 79, 81, 84, 85, 87, 96, 99, 102, 119, 121, 124, 126, 134, 149, 156, 159

Floating-Point Unit (FPU)

Module for handling floating-point arithmetic, analogous to the Arithmetic Logic Unit (ALU), but with most operations multi-cycle due to their complexity. 86, 99, 124, 134, 157

Global Data Protection Regulation (GDPR)

EU regulation mandating industry robustness against data breaches, laying out severe penalties for violations [4]. 158

Hardware Thread (Hart)

RISC-V's unit of execution: typically a single core, or single Simultaneous Multi Threading (SMT) thread within a core, having its own register file. 119

Hardware Description Language (HDL)

Language used for hardware design, allowing description of circuit behaviour that can target simulation, FPGA, or ASIC. 27, 30, 31

Input/Output Memory Management Unit (IOMMU)

Device that virtualises and protects memory as seen by Direct Memory Access (DMA) devices. 29

Internet of Things (IoT)

Networking appliances to enable convenient telemetry and control, e.g. from a smartphone. 51, 165

Intellectual Property (IP)

Designs of hardware components, typically described at the Verilog level. 52

Instructions Per Clock (IPC)

The average number of instructions executed per clock cycle. 52, 70, 91, 121, 127, 130

Instruction Set Architecture (ISA)

The specified interface between hardware and software, consisting of a description of the instructions and any other guarantees that must be upheld. 16, 19, 27, 28, 29, 30, 34, 36, 44, 45, 47, 48, 51, 55, 57, 58, 59, 63, 64, 77, 89, 101

Just-In-Time compilation (JIT)

Interpreting a program by dynamically compiling blocks to native instructions as they are run. 33

Load/Store Queue

A structure preserving the outstanding memory accesses in program order. 20, 105, 108, 110, 112, 121, 124, 125, 126, 130, 144, 145

Lookup Table (LUT)

The unit of reconfigurable logic within an FPGA, able to implement any six-input binary function. Used as the basic unit of consumed logic area. 27, 38, 79, 81, 84, 85, 86, 96, 99, 122, 124, 125, 134, 155

Macro-Op Fusion

Combining of common sequences of instructions together into a single operation in decode. 28

Memory-Mapped Input/Output (MMIO)

Peripherals or control registers that are mapped into the address space of the processor, having effects beyond changing the value to be read back later. 104

Memory Management Unit (MMU)

Hardware support for address translation, enabling virtualisation and security as different processes can observe different virtual address spaces. 14, 32, 33, 40, 51, 54, 101

Memory Protection Unit (MPU)

Hardware support for memory protection, typically lighter weight than a Memory Management Unit (MMU), so more suitable for microcontrollers, and focused only on security rather than virtualisation. 33, 40, 51

Operating System (OS)

Software responsible for leveraging hardware into a usable application interface. 33, 48, 49, 76, 77, 79, 102, 119, 139, 143

Program Counter (PC)

Address of the instruction currently being executed by a processor. 11, 15, 29, 52, 54, 60, 62, 70, 72, 74, 88, 104, 105, 108, 109, 125, 164

Program Counter Capability (PCC)

Extension of the PC to include the capability metadata for the processor's current execution. 10, 58, 60, 63, 67, 68, 70, 86, 89, 108, 109, 110, 125, 161, 162, 164

Platform-Level Interrupt Controller (PLIC)

RISC-V-specified module to convert interrupt wires into interrupt traps within the cores. Since it is platform-level, it is instantiated once, regardless of the number of cores. 84, 85, 124

Physical Memory Protection (PMP)

RISC-V specification for a component controlled using Control and Status Registers (CSRs) to restrict access to memory regions on a coarse-grained basis. 29, 32, 33, 34, 51, 84, 138

Page-Table Entry (PTE)

Section of memory that describes a level of translation that must be followed to convert a virtual address to a physical address. Includes description of the target address, as well as permission bits guarding use of the entry. 33, 141

Return Address Stack (RAS)

Prediction structure that detects calls into subroutines, pushing the caller address to a stack so that it can be predicted when the corresponding return (which is otherwise a difficult to predict indirect jump) is detected later. 54, 88, 104

Rivest Cipher 4 (RC4)

A stream cipher on which Wired Equivalent Privacy (WEP) was based, forming part of the MiBench benchmarking suite [62]. 166

Return-Oriented Programming

Attack technique that bypasses many existing stack security measures, overwriting return addresses to use existing legitimate code as exploit gadgets [108]. 37, 45

Reduced Instruction Set Computing (RISC)

Architecture design principle favouring simple single-purpose instructions to make the common case fast, as an alternative to Complex Instruction Set Computing (CISC). 11, 28

Real-Time Operating System (RTOS)

Lightweight operating system, typically for microcontrollers, with emphasis on low or predictable event-handling latency. 51, 77

RISC-V Formal Interface (RVFI)

Tracing protocol for RISC-V processors to specify results of running instructions [134]. 16, 72, 73, 74

RISC-V Formal Interface with Direct Instruction Injection (RVFI-DII)

Use of Direct Instruction Injection (DII) and RISC-V Formal Interface (RVFI) together. 51, 71, 72, 74, 118

Sail

A language produced by the University of Cambridge's formal group used to produce executable and verifiable Instruction Set Architecture (ISA) specifications [10]. 71, 73, 75, 76, 116

Special Capability Register (SCR)

CHERI RISC-V control registers that contain a full capability. 22, 67, 70, 109, 110

seL4

A formally verified L4 microkernel [69]. 51

Secure Entry (Sentry)

Capability that can only be unsealed by jumping to it, produced by default on linking from a jump. 22, 58, 59, 64, 77, 106, 117, 158, 161

Simultaneous Multi Threading (SMT)

Support for multiple execution threads within a core supported in hardware, whereby a core has registers for multiple threads and instructions from each can be interleaved. 13

System on Chip (SoC)

Chip-level design of a system, often including a processor and supporting peripherals, some of which may be DMA capable. 9, 10, 11, 53, 79, 84, 85, 102, 121, 156

SPEC

Corporation designing benchmarks designed to cover a range of applications [29]. 10, 43, 44, 88, 119, 121, 126, 127, 129, 130, 132, 134, 149, 156, 157, 165, 167

System Security Integration Through Hardware and Firmware (SSITH)

DARPA program aiming to develop hardware security architectures to protect against classes of vulnerabilities exploited in software [143]. 11, 155, 156

Tightly-Coupled Memory (TCM)

A memory subsystem favoured by microcontrollers, in which the processor has an on-chip memory providing quick access, as an alternative to caches [12]. 79, 84

Translation Lookaside Buffer (TLB)

Cache of virtual to physical address mappings used to accelerate translation. 28, 54, 79, 104, 105, 108, 121, 140, 141, 143, 151, 152, 156

Wired Equivalent Privacy (WEP)

A defunct protocol for encrypting traffic sent over 802.11 wireless networks, so was widely implemented by microcontrollers for devices requiring wireless internet access. 15, 166

Chapter 1

Introduction

1.1 Architectural security

Computer systems have become ubiquitous very quickly, changing the world beyond recognition. They are integrated into the most important areas of our lives—in control of safety-critical processes and handling our data—while being accessible from anywhere in the world via networking. Security has not kept up. Throughout their development, this has been realised through one crisis after another as attackers exploit the same common software mistakes to completely take over systems. Cybercrime due to malware (including the resulting prevention) costs the global economy a significant amount each year [6]. Two recent examples—Heartbleed [41] and WannaCry [46]—exemplify the problem well. In both cases, a small error in a complicated piece of software allowed the security properties of their respective systems to be completely violated. Industry incentives discourage programmers from producing robust software, as the competitor who ships first receives the sales, with the security violations following much later [42]. The status quo is begging for change.

Increasingly, programmers express their ideas at higher levels of abstraction, for example through high-level managed languages. This allows them to express intended behaviour in a safe way, with opaque memory management preventing confusion over the objects the programmer intended to access. However, systems programmers—who must write the software underlying these languages or other low-level software—are just as susceptible to the same problems and cannot benefit from managed languages. Capabilities offer a potential solution: by preserving the programmer's intended limits on their code's execution, opportunity for code to be misused outside of its intended purpose is limited. Capability systems have long been proposed (as early as 1966 [37]), but past proposals have seen little success, partly due to high overheads and poor compatibility.

CHERI [128] is a contemporary capability-based architecture extension aimed to improve security for C and C++ code. It does this by enforcing the principles of least privilege and intentionality, with hardware enforcing properties inferred from the programmer's code.

Initial promise for CHERI was shown augmenting a MIPS processor [136]. However, MIPS has seen declining usage, limiting this implementation's applicability to modern processors. RISC-V is an open architecture, seeing increasing popularity, especially in research [131]. This offers a unique opportunity to shape the security landscape, investigating how capabilities can be incorporated into the Instruction Set Architecture (ISA) before legacy code accumulates and the ISA solidifies. Arm is also investigating CHERI, with the Morello program producing a research prototype of a CHERI-augmented contemporary high-performance core [54].

Adapting CHERI to RISC-V gives a good opportunity to investigate its implications in different microarchitectural contexts. For instance, the original CHERI processor was in-order [136], leaving many questions unanswered about composition with more advanced microarchitectural techniques used in contemporary processors. Conversely, there is also an opportunity to investigate scaling in the opposite direction to area-constrained microcontrollers. Finally, the new implementations will allow further development of temporal safety techniques, a vulnerability class less naturally resolved by capabilities.

1.2 Hypotheses

This thesis aims to answer some key hypotheses about CHERI microarchitecture implementations.

Hypothesis H.1

CHERI can be implemented to provide spatial safety for RISC-V microcontrollers, with a small area, power, clock frequency, and performance impact.

RISC-V is a new target for CHERI, which was initially applied to cores implementing the MIPS architecture. Since RISC-V shares a design philosophy and many features with MIPS, many of the architectural and microarchitectural capability features may be expected to transfer across. In addition, the particular focus on small-scale microcontrollers may reveal new challenges, particularly around the amount of additional logic required to support capabilities. The definition of "small" for the various overheads will depend on the application and associated acceptable tradeoffs.

Hypothesis H.2

CHERI can be implemented to provide spatial safety for RISC-V out-of-order superscalar application-class cores, with a small area, power, clock frequency, and performance impact.

There is no pre-existing (open[1]) implementation of CHERI for an out-of-order, superscalar, application-class core. I hypothesise that the model applies well to such processors. This

[1]Morello [54] is an instantiation of CHERI for Arm that was developed concurrently with this PhD,

will require considering interactions with more complex microarchitectures, including Load/Store Queues, reorder buffers, and multiple pipelines. New challenges, such as deep speculation, may diminish CHERI's effectiveness.

Hypothesis H.3

The CHERI area, power and performance impact becomes less significant for larger cores.

There is little current evidence as to how the CHERI overheads scale with the size of the host core. Adding CHERI to a core involves adding additional logic and scaling some structures, such as the register file and datapaths. However, some structures are largely unmodified, such as floating point, caches, and branch prediction. Therefore, we might expect larger cores to see a smaller fractional area overhead, as the structures grown for capabilities represent a smaller fraction of the core. I expect power overheads to mirror area overheads and see a similar trend. Additionally, the performance overhead comes in the form of added instructions to manipulate capabilities and increased memory bandwidth from capability metadata. It may be the case that an out-of-order design allows these overheads to be hidden. By applying CHERI in an similar way across the Piccolo, Flute, and Toooba cores, I hope to investigate these effects.

Hypothesis H.4

Temporal safety can be implemented efficiently atop CHERI for RISC-V processors.

Following the direct spatial safety and compartmentalisation applications of CHERI, attempts have already been made to achieve temporal safety. It is not clear how these apply to new RISC-V cores, especially application-class. I conjecture that there is further opportunity to optimise the sweeping approach. Moreover, I aim to show that RISC-V cores are amenable to supporting temporal safety using these approaches via microarchitectural and architectural acceleration.

1.3 Contributions

The following are contributions of this thesis:

- The first CHERI RISC-V microarchitectural implementations, both as proofs-of-concept and platforms for further capability research:
  - CHERI extensions of the existing Piccolo and Flute microcontrollers,

but does not have an open microarchitecture. Note that the Morello prototype used microarchitectural research from the CHERI team, including *CHERI-Concentrate* [135], the tag controller, and the merged register file.

  - The first open superscalar CHERI implementation, as an extension of the existing Toooba processor, completed in collaboration with Jonathan Woodruff and Alexandre Joannou, with my personal contributions including:
    * Contributions to the overall design approach, such as which pipelines to modify,
    * Modifying the pipelines,
    * Decoding and implementing the capability manipulation instructions,
    * Adding the capability checks,
    * Implementing the Special Capability Registers (SCRs) and exception logic,
    * Implementing the Secure Entry (Sentry) mechanism,
    * Implementing the changes to support capability-aware compressed instructions;

- Evaluation of the microarchitectures, including FPGA area overhead, effect on timing, and impact on performance and power;
- A study of the scaling of CHERI overheads across various scales of microarchitectures;
- Exploration of microarchitectural implications of various architectural improvements for CHERI RISC-V, including:
  - The merged capability register file,
  - Tag clearing instead of exceptions on monotonicity violations,
  - The Sentry mechanism,
  - A dynamically switched capability encoding mode;
- Identification of significant tradeoffs in the implementation of CHERI for application-class cores;
- Protection against a class of speculative execution attacks, including an audit of the microarchitectural possibilities for related attacks against the capability model in Toooba;
- New additions to *CHERI-Concentrate* [135] for single-cycle bounds checking;
- The shadow bitmap high-level approach for representing revoked capabilities during a revocation sweep;
- The `MESTI` cache coherence approach for capability tags;
- An investigation of various cache optimisations for finding tags;
- Contributions to the TestRIG RISC-V testing framework.

1.4 Publications

I have co-authored the following publications throughout my PhD:

CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety (*52nd IEEE/ACM symposium on Microarchitecture, Columbus 2019*). Co-authored paper: my contributions include development of the sweeping algorithm applied, including the idea to keep a shadow-bitmap of freed capabilities. Paper: `https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/201910micro-cheri-temporal-safety.pdf`

Four CHERI RISC-V Microarchitectures (*RISC-V spring week, Paris 2022*). Co-authored talk: my contributions include work on three of the microarchitectures discussed, assembling the slides, and writing and delivering the talk. Slides: `https://open-src-soc.org/2022-05/media/slides/4th-RISC-V-Meeting-2022-05-04-16h00-CHERI-Cambridge.pdf#page=42`

TestRIG: Using RVFI-DII to eliminate the "Test gap" between specification and implementation (*RISC-V Workshop, Zurich 2019*). Co-authored talk: my contributions include significant improvements to the verification engine, allowing more complex templates for deeper pipeline testing and smarter counter-example shrinking. I also performed processor instrumentation to enable Direct Instruction Injection (DII). Video: `https://youtu.be/kB7SHlJtfC4`, Slides: `https://content.riscv.org/wp-content/uploads/2019/06/13.20-TestRIG.pdf`

Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8) (*Technical report, Cambridge 2020*). Co-authored technical report: my contributions include discussion of tag clearing on monotonicity violations, the capability encoding mode switching mechanism, and elaboration of various other aspects of the CHERI architecture. Technical report: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf`

The following are pending publication:

CompartOS: CHERI Compartmentalization for Embedded Systems Co-authored paper: my contributions include CHERI adaptations of the underlying hardware, evaluation of area and frequency, and support with bringing up software. Paper: `https://arxiv.org/abs/2206.02852`

Random Testing of RISC-V CPUs Using Direct Instruction Injection Co-authored paper: my contributions include contributing to the framework, augmentation of Piccolo and Flute with DII, adding features including smart shrinking and

recursive templates to the verification engine, and help with evaluation including test coverage measurement. Paper: See Appendix C.

Architectural Contracts for Safe Speculation in CHERI Processors Co-authored paper: my contributions include the CHERI augmentation of Toooba that is evaluated, auditing the microarchitecture for potential side-channel vulnerabilities, and development of a more side-channel resistant implementation.

The following publications are planned, in collaboration with others in the CHERI team:

- A summary of the combined software and hardware effort on CHERI RISC-V, including the architecture, hardware implementations, compiler, and operating system, providing an updated CHERI performance analysis beyond that provided by CHERI MIPS.
- An introduction to superscalar CHERI, as applied to Toooba, discussing how to optimise the extensions for the microarchitecture and evaluating the impact of the CHERI extensions to the power, performance, and area of the Toooba core.
- Further work on revocation, following up on our *CHERIVoke* [138] and *Cornucopia* [132] papers by discussing further optimisation, particularly optimised sweeping hardware.

1.5 Open-source contributions

The implementation work required to enable the research in the thesis is all open-source, primarily contributing to the following repositories:

Piccolo A fork of the Bluespec microcontroller, extended with CHERI modifications. My contributions include adding support for DII and adding support for CHERI. `www.github.com/CTSRD-CHERI/Piccolo`

Flute A fork of Bluespec's larger microcontroller, extended with CHERI modifications. As with Piccolo, I added DII and CHERI support. `www.github.com/CTSRD-CHERI/Flute`

Toooba A fork of Bluespec's out-of-order processor, extended with CHERI modifications. The baseline processor is itself a fork of MIT's RiscyOO processor. I performed much of the CHERI modification, including the capability checks, treatment of capabilities in the pipelines, and various changes to support temporal safety. `www.github.com/CTSRD-CHERI/Toooba`

CHERI Capability Library A standard interface for capability compressed format operations and several implementations that operate at various levels of decompression. My contributions include the encapsulation approach and various additional algorithms for manipulating the compressed format. `www.github.com/CTSRD-CHERI/cheri-cap-lib`

Tag Controller AXI component for providing tagged memory using a shadow-space. I made various improvements, including adaptations required for the RISC-V implementations and temporal safety. `www.github.com/CTSRD-CHERI/TagController`

TestRIG Infrastructure for connecting RISC-V implementations with models and verification engines. I made various infrastructural improvements, such as the ability to run multiple instances in parallel. `www.github.com/CTSRD-CHERI/TestRIG`

QuickCheckVEngine Verification engine that uses Haskell's QuickCheck library [28] as a basis for generating tests and comparing implementation traces. I performed significant design of almost every aspect, particularly smarter shrinking and templates. `www.github.com/CTSRD-CHERI/QuickCheckVengine`

Sail CHERI-RISC-V CHERI RISC-V model implementation using the Sail architecture description language [10]. My contributions include implementation of various architectural features, most notably tag-clearing as opposed to exception semantics. `www.github.com/CTSRD-CHERI/sail-cheri-riscv`

BlueStuff Bluespec libraries providing interconnect utilities, including for AXI. I implemented and improved many of the interconnect components required for the memory subsystems of the CHERI implementations. `www.github.com/CTSRD-CHERI/BlueStuff`

1.6 Thesis overview

The thesis is arranged into a background chapter (Chapter 2), three implementation chapters (Chapters 3, 5, and 7), two evaluation chapters (Chapters 4 and 6), and a conclusion (Chapter 8).

- Chapter 2 covers existing research and relevant context for the work.
- Chapter 3 discusses the implementation of CHERI for the Piccolo and Flute microcontrollers.
- Chapter 4 evaluates this work, investigating the CHERI overheads.
- Chapter 5 discusses the implementation of CHERI for Toooba, the first open implementation of CHERI in a superscalar, out-of-order core.

- Chapter 6 evaluates the CHERI Toooba implementation in a similar manner to Chapter 4.
- Chapter 7 describes the investigation into architectural and microarchitectural optimisation of temporal safety using CHERI, mostly again focused on the Toooba core.
- Chapter 8 draws overall conclusions from the work done and evaluations performed.

Chapter 2

Background

2.1 Processor design

This section introduces background aspects of processor design, including the RISC-V ISA and Bluespec Hardware Description Language (HDL) used.

Processors have seen very rapid performance improvements since the first silicon chips in the 1960s [82]. This has been facilitated by transistor scaling, following Moore's Law, and by improvements in microarchitectural techniques. Contemporary application-class processors maximise extraction of instruction-level parallelism using superscalarity and out-of-order execution [57]. Superscalarity allows multiple instructions to be issued and retired per clock cycle. Out-of-order execution prevents a small number of slow instructions (typically memory loads) from halting progress. These features are in addition to achieving a high clock frequency by heavily pipelining every part of the execution. The increasing complexity and transistor count required for application-class performance has led to separate categories of microarchitectures emerging: in particular, microcontrollers avoid this complexity for applications that are not performance-critical. Since they present different tradeoffs, this thesis talks separately about application-class cores (defined as those offering these advanced features) and microcontrollers (defined as simple, in-order, scalar cores with short pipelines).

Processor design has benefited from the development of FPGAs for rapid prototyping. These use Lookup Tables (LUTs) and reconfigurable interconnects to implement arbitrary logic operations that are dynamically reprogrammable, at the expense of an order-of-magnitude penalty to clock frequency and area for a similar design. However, since the gates can still operate in parallel, the performance of running a design on FPGA exceeds that of simulating it by several orders of magnitude. In addition, implementing a design for FPGA provides an initial sense of its area and timing characteristics, as well as providing some level of performance realism in terms of pipelining and cache effects. This allows investigations and proofs-of-concept to be carried out without enormous financial and time costs. Since the design is described at the register-transfer level, the FPGA behaviour will

match that on ASIC, avoiding duplicated design and verification work and adding further evaluation realism when compared to a simulator. We therefore use FPGAs to prototype and evaluate the CHERI extensions, comparing to the baseline designs synthesised on the same FPGA.

Note that FPGA timing, area, and performance results have limited applicability to a real ASIC synthesis. Timing is not simply a scaling of that for ASIC, in part due to differing FPGA hard blocks (such as to accelerate carry chains) and ASIC standard cell libraries. The area of structures differs significantly for similar reasons. Most significantly, FPGAs offer BRAMs, allowing single-cycle access memories with a much smaller relative overhead than registers. Finally, performance may differ as DRAM frequencies are proportionally much higher for FPGA systems. However, due to the difficulty of ASIC flows, the scope of this thesis is restricted to FPGA metrics, leaving optimisation and measurement of the designs for ASIC flows as future work. As an intermediate step, a cycle-accurate FPGA simulation framework such as *FireSim* [67] could be used to provide more accurate DRAM timings. In terms of the design insights given, FPGA designs are still much closer to ASIC designs than using an instruction set simulator (even combined with a pipeline model). For example, critical paths force consideration of the work done per cycle and interactions between the pipeline and caches are realistic. I attempt to mitigate FPGA evaluation artefacts and explicitly highlight them where present.

2.1.1 RISC-V

All processors investigated in this thesis use the RISC-V ISA. This section introduces the relevant features of RISC-V and explains why it is a good target for CHERI.

RISC-V [126] is a relatively new Reduced Instruction Set Computing (RISC) ISA governed by RISC-V International. The ISA is open for use (but not modification), encouraging open implementations. It is seeing increasing interest and adoption, both in the microprocessor market and in prototypes of application-class processors, with the development of the superscalar RISC-V Boom [27] and Riscy-OO cores [145], among others [39]. It is estimated that over 10 billion RISC-V processors have shipped to date [131]. However, these are mostly small microcontrollers; it remains to be seen whether RISC-V will penetrate the application-class processor space commercially.

The full ISA is described in the user-mode [126] and privileged [127] manuals. Familiarity with the ISA, including instruction mnemonics, is assumed throughout the thesis.

Considerable effort has been invested to make RISC-V a good target for embedded and application-class cores alike. To optimise for microcontrollers, the instruction encoding is kept simple. For example, register indices are always encoded in the same position in the instruction. Side-effects of instructions are avoided: RISC-V does not feature carry flags and arithmetic cannot raise exceptions. The ISA endeavours to avoid making assumptions

about the microarchitecture, such as by avoiding assumptions about caches and Translation Lookaside Buffers (TLBs). This approach has its drawbacks: for example, instructions for cache manipulation (invalidates and flushes), which are useful for practical Direct Memory Access (DMA), have only recently been specified [47]. The ISA also avoids conditional instructions in the base ISA. For larger processors, it is anticipated that Macro-Op Fusion can recover some of the performance lost by having a minimal instruction set.

The privileged specification describes the interaction between the different privilege modes: Machine and (optionally) Supervisor and User. This includes specification of Control and Status Registers (CSRs) that allow software to manipulate and query system state. For example, the `mtvec` CSR controls the address taken in the event of an exception or interrupt, with the previous Program Counter (PC) installed in `mepc` to allow recovery once the trap is handled. Memory translation between virtual addresses seen by Supervisor and User modes and physical addresses seen by Machine mode is controlled via the `satp` CSR, including pointing at the root of the page table. A Physical Memory Protection (PMP) mechanism may also be provided, allowing software to specify address ranges to protect in CSRs. This mechanism is discussed further in Section 2.2.1.2.

One of the key attractions of RISC-V is the freedom to implement a processor with arbitrary implementer-specific instructions to accelerate the task the processor is intended for. This is something that is forbidden by a standard Arm license. The risk to this approach is that the RISC-V ecosystem may become fragmented, with all implementations supporting vastly different instructions, preventing cross-platform software compatibility. As such, the RISC-V standardisation process is intended to prevent multiple independent feature interfaces from arising, with mixed success. For example, RISC-V International is attempting to standardise the interface to the Input/Output Memory Management Unit (IOMMU) and has been presented with five independent proposals from implementers [94]. The RISC-V architecture is organised as: a relatively minimal base ISA, which must be implemented; additional optional standard extensions; plus non-standard extensions.

Some key standard extensions include:

Atomic (A) extension Adds atomic memory instructions, including load-reserved and store-conditional (which are preferred over compare-and-swap), and operations to perform in memory, returning the old value.

Compressed (C) extension Adds a 16-bit encoding space for common operations to improve instruction density, reducing instruction cache pressure. 16-bit instructions can be freely interleaved with full 32-bit instructions, unlike Arm's Thumb [113], which requires a mode switch.

Floating-point (F) and double-precision floating-point (D) extensions Add support for IEEE floating point of the relevant widths, with an additional register file added, and CSRs to specify global state such as rounding modes.

Multiply/Divide (M) extension Adds instructions for integer multiply and divide, again avoiding exceptions in all cases.

Vector (V) extension Recently specified, adds instructions to operate on vectors of registers, with a wide variety of vector sizes and counts supported (but not mandated) for implementing processors.

The openness and continuing development of RISC-V make it ideal for the CHERI team's capability research. The open ISA encourages open-source implementations, helping with availability of baseline implementations (most notably RiscyOO) and potentially allowing others to build on our modified processors. In addition, the ISA is still being actively extended, with security features currently being discussed and specified. This may allow a capability extension to be ratified, cementing it within the ISA and allowing other extensions to be adapted around it. CHERI fits in well with the modular design of the ISA: initially a non-standard extension, we hope to standardise CHERI RISC-V as an optional standard extension. A lack of legacy software will also reduce adoption difficulties. The similarities with MIPS also improve transferability of the existing CHERI research.

2.1.2 Bluespec

Software tools for hardware development fall short in usability compared with those for software, perhaps due to the large barrier to entry for companies to design and fabricate their own hardware. The most widely-used HDLs today are VHDL and (System)Verilog. These lack some of the basic type-safety and expressibility features expected in software languages, even C, making them more akin to assembly-level languages. However, some alternative higher-level HDLs are currently gaining traction: for example Berkeley's Chisel, built atop Scala [11]. The work in this thesis uses the Bluespec HDL, designed by Bluespec Inc. [95].

The Bluespec HDL is built atop Haskell, providing clear distinction between evaluation-time computation and descriptions of hardware logic wrapped in the `module` monad. It provides strong type-safety as well as guarded atomic actions (`rules`): descriptions of behaviour that must be run together or not at all, based on programmer-specified and implicit conditions. As shown in Figure 2.1, this allows high-level reasoning about designs, leaving fiddly and error-prone control signals to be generated by the compiler.

There are several advantages to using Bluespec, including:

Baseline processor availability The requisite range of processors are available in the source language, including the Piccolo and Flute microcontrollers from Bluespec Inc. and the superscalar RiscyOO from MIT.

```
interface FIFO #(type t);
  method Action put (t x);
  method ActionValue#(t) get ();
endinterface

module mkFIFO (FIFO#(t)) provisos (Bits#(t,_));
  Reg#(Maybe#(t)) state <- mkReg(Invalid);

  method Action put (t x) if (!isValid(state));
    state <= Valid (x);
  endmethod

  method ActionValue#(t) get() if (isValid(state));
    state <= Invalid;
    return state.Valid;
  endmethod
endmodule
```

Figure 2.1: A simple single-element First In, First Out (FIFO) implementation in Bluespec. The FIFO is parametrised on the type `t` provided it has a bit representation, allowing the FIFO to be reused in different contexts in a type-safe way. Users of this module can freely call `put` and `get`, knowing that back-pressure will be automatic if the FIFO is not ready due to being full or empty, with no risk of data being lost. This code is only illustrative as FIFOs are provided as standard library components, including variants with different numbers of stored elements and variants that remove the cycle of latency between `get` and `put`.

Modularity Bluespec is explicitly designed to maximise code reuse and modularity, with interfaces and type parametrisability being integral to the language [95]. RiscyOO's design is a good example of this [145].

Correctness By providing a higher level of abstraction and type-safety, designs avoid many common bugs, such as around control signals and losing data.

Existing CHERI work The existing CHERI MIPS processor was written in Bluespec, allowing some of the more intricate and already-optimised capability compression logic to be factored out and reused.

Tool compatibility By compiling to a small, commonly-supported subset of Verilog, the same source Bluespec can be used for synthesis with either Vivado or Quartus and for simulation with common simulation tools, as well as Bluespec's own.

The use of the language does carry some disadvantages. The language was previously very expensive to use in industry, severely limiting its adoption. Partly as a result of this history, the Bluespec community is very small, even compared to other high-level HDLs like Chisel. However, the language was recently made free-to-use and the compiler was open-sourced [58], removing this as a limitation. Finally, the higher-level description can

sometimes make it harder to express the desired behaviour for pieces of circuitry critical for area or timing.

2.2 Spatial safety

Programmers write code that deal with abstract objects: named constructs that have limited size and lifetime. The compiler must map objects onto actual memory locations. Spatial safety violations confuse objects at two different locations: code written intending to access one object instead accesses another. This may allow privileged state to be leaked or corrupted via accesses intended for a user-accessible structure. A common example is a buffer overflow, where an index is used that exceeds the length of an array, causing whatever follows in memory to be accessed instead [77]. As a prominent example, this was the vulnerability that enabled the Heartbleed attack [41], which allowed arbitrary memory reads. More concerning, a user may provide more input to a buffer than it is allocated to contain, causing the overflowing data to be written over whatever happens to follow in memory. This may allow them to write arbitrary code into memory and overwrite the function return address to jump to it once the function returns.

Szekeres et al. provide a good summary of the state of memory safety [122]. The paper examines contemporary memory safety techniques used on top of virtual memory to mitigate spatial safety attacks, concluding that they do not sufficiently mitigate this threat. The failure of current protection mechanisms motivates a rethink of approaches to memory safety, to provide security-by-design through enforcement of robust security principles [109, 112, 93].

2.2.1 Deployed protections

Hardware and software are already adapted to partially mitigate spatial safety issues due to their prominence and severity. This section discusses these protections. Most significantly, Memory Management Units (MMUs) and PMPs provide access control, which can confine spatial safety violations to within a process. Safe languages guarantee safety properties, but at the expense of performance and low-level expressivity. These mitigations all have their own costs; capabilities may allow them to be removed or disabled, recovering some performance.

2.2.1.1 Memory Management Units

Contemporary application-class processors provide memory translation: each process has a virtual view of memory that is translated by a hardware MMU into physical addresses that are actually used to interface with DRAM. This allows processes to act as if they have exclusive ownership over memory, rather than having to negotiate memory management with other arbitrary programs. It also enables over-provisioning, where the Operating System (OS) can offer large amounts of virtual memory that is only mapped on use. Memory translation has the side-effect of providing isolation between processes, as the page table root is switched on context switch. Programs therefore have no way of expressing an access to the address space of another process. In addition, the Page-Table Entrys (PTEs) that manage the translation are augmented with permission bits, restricting access types at a coarse granularity. However, beyond this MMUs provide little to no intra-process protection and limit the granularity of safe sharing to page boundaries. Protection is provided based on the accessed address and does not take into account the call-site, providing no intentionality.

Various MMU permission conventions help to make attacks more difficult. Guard pages can be left unmapped, or mapped with no permissions, to detect egregious buffer overruns, for example between the stack and the heap [66]. In addition, non-executable stacks prevent stack buffer overflows from allowing an attacker to write arbitrary code that they can jump into with a control-flow gadget. This extends in general to the principle of $W \oplus X$: avoiding memory being both writable and executable. Sometimes it is impossible to achieve this: Just-In-Time compilation (JIT) code generation, for example, needs both write and execute access to the same memory. To maintain $W \oplus X$ in this case, the permissions must either be split across two different mappings or switched dynamically.

2.2.1.2 Physical Memory Protection

On embedded microcontrollers lacking MMUs, by default, spatial safety violations could allow corruption between processes. To enable some isolation between processes and the kernel, such cores may offer a Memory Protection Unit (MPU) that has smaller area and power cost by just implementing a subset of the MMU privilege protection mechanisms[1], without offering translation [114]. MPUs may also be composed with MMUs to protect machine-mode code and guard against memory management mistakes made by the kernel. Some RISC-V cores offer a standardised MPU, which RISC-V dubs a PMP [127]. Here, the core offers a fixed number of CSRs to configure memory regions that have restrictions on their access, principally offering read, write, and execute permissions. These configurations can only be changed while the core is in machine mode.

Uses of the PMP are very coarse-grained. Bootloaders will often make their memory

[1] Although, in principle, MPU regions can be more granular than MMU pages.

unwritable, preventing corruption from within the loaded OS. Lindemer, Midéus, and Raza discuss using the PMP to enforce thread isolation on embedded cores, performing a reconfiguration on every context switch to allow threads access only to their memory [79]. However, they note that the PMP specification is currently insufficient to support their goals, needing an additional privilege level.

The security offered by the PMP mechanism is a subset of that offered by capabilities: regions can have their access restricted, but only on a coarse granularity, since only few regions are available (16 or 64 depending on implementation). There is no intentionality: no effort is made to ensure the executing code intended to access a particular region. In addition, different access rights to the same physical region cannot be delegated to different compartments without reconfiguration on every context switch, since access control is performed by global configuration. Reconfiguration can only be performed in machine mode, preventing user code from protecting itself. Finally, the PMP must perform an associative lookup to ensure each access is not prohibited by any of the PMP registers. This implies a relatively large power and area cost of the scheme, growing linearly with the number of PMP registers supported. Capabilities avoid associative lookups by requiring the intended capability to be quoted on access. However, capabilities require code to be at least recompiled to provide their full protections, whereas the PMP can be used to protect existing binaries. By decentralising the authority in the system, capabilities can also make it harder to reason about security and perform revocation. For example, access to a particular region can be forbidden simply by adding a PMP entry, while forbidding it in a capability system requires ensuring no capability to that region exists, either by scanning or maintaining an invariant.

2.2.1.3 Safe languages

The CHERI ISA is focused on adding safety to the C and C++ languages, which are vulnerable by default to spatial safety attacks. An alternative might be to move away from these languages in favour of languages with safety built-in, such as via compiler-added bounds checks and more restrictive treatment of pointers. This section discusses possible alternative languages. However, it is worth noting that, even if the ideal safe language emerged and became universally adopted, billions of lines of critical C and C++ code are already written and will continue to be relied upon until replaced [122]. Therefore, there is still value in adding spatial safety to C and C++ code.

Most current high-level languages ensure spatial safety by performing dynamic bounds-checks, for example Java, C#, and Python. This is supported by maintaining length information and performing run-time bounds checks whenever an array is accessed. This is made possible by use of a virtual machine that tracks object metadata. In essence, capabilities try to achieve a similar effect, although implementing the bounds checks in hardware reduces performance overhead. In addition, safe languages make memory

references opaque and ban pointer arithmetic. This avoids confusion between objects, but is obstructive to low-level programming.

Rust [83] is a relatively new language, aiming to replace C, that offers C-like performance but enforces programming that guarantees spatial and temporal memory safety (as well as data races), using the notion of ownership. However, Rust offers an "unsafe" mode which forgoes these guarantees. Many low-level Rust programs and the language's libraries use this mode when the type system is too restrictive, making these security-critical pieces of software more vulnerable to attack. Combining Rust with CHERI is active research [119]. For example, CHERI may be usable to accelerate bounds checks inserted when static checks cannot guarantee the access is in-bounds. CHERI may also be able to improve the safety of unsafe Rust, making it at least analogous to C compiled with CHERI.

Even for safe languages, CHERI capabilities allow bounds checks to be performed in parallel with memory access without additional instructions, unlike dynamic checks added by the compiler on a regular processor. Modifications to the compilers and runtimes to exploit CHERI could therefore bring significant performance improvements.

2.2.1.4 Other protections

Stack canaries place known values between variables and compiler-managed critical state on the stack, checking before return that the canary is unmodified [13]. The intention is to detect buffer overflows as they overwrite the canary on their way to the intended value, typically the return address. This generally makes vulnerabilities less convenient to exploit, as the attacker needs to also find a way to expose the stack canary.

Address Space Layout Randomisation (ASLR) randomises stack and code addresses to hinder spatial safety attacks by preventing the attacker from knowing where critical structures are, relative to the memory they can control [17]. As with stack canaries, this makes attacks trickier in practice, but certainly does not prevent them, typically just requiring an additional step to extract the required addresses [116].

Compilers provide further protections against stack-based buffer overflows. For example, LLVM can split the stack into a safe region and an unsafe region [74]. The safe region includes code pointers and values statically determined to only be accessed safely. This prevents some of the most exploitable buffer overflows, but still allows overruns between objects in the unsafe region, and can be bypassed [53].

2.2.2 Related research

This section lists research aiming to address spatial safety issues in C, but that has not seen universal adoption. Woodruff et al. [136] and Szekeres et al. [122] give further comparisons between existing approaches. Approaches primarily aimed at temporal safety are deferred to Section 2.3.1.

CCured **[90]** A transformation on C that statically analyses bounds accesses, inserting dynamic bounds checks for those that cannot be shown to always be in-bounds. This does guarantee spatial memory safety, but breaks a lot of existing C code, as it exploits the C specification's undefined behaviour when a pointer goes more than one byte out of bounds in an address calculation. It also changes the memory layout of the program in an unpredictable way to store bounds information. Finally, it has a large run-time overhead, with a 38% overhead in the SPEC INT benchmarks, but a worst case overhead of over 800% in their pointer-heavy `Ptrdist` benchmarks.

Cyclone **[61]** A safe dialect of C, which, much like *CCured*, combines static checks with dynamic bounds checking. However, the language has much less of a focus on supporting legacy code. For example, traditional pointers cannot be used in pointer arithmetic. An alternative fat pointer type with dynamic bounds checks must be used instead. Rewriting programs in Cyclone requires editing approximately 10% of the lines of code and run-time overheads of around 100% are seen in many programs.

Fail-Safe C **[98]** A compiler-only modification that aims to support the full C standard as well as common tricks. The compiler augments all generated memory operations with bounds checks and doubles the size of pointers, storing bounds information in the additional bits. They report up to 700% run-time overhead.

HardBound **[38]** Like CHERI, this approach augments the ISA with instructions to manipulate bounds, and checks these bounds on pointer dereference. However, the bounds information is entirely stored in a shadow space, both in registers and in memory, with compression optimisations to reduce the 3× space overhead this would incur. This approach achieved a worst-case run-time overhead of below 25% for all the benchmarks tested. However, there is no guarantee of monotonicity in bounds, meaning an attacker with access to a bounds-setting gadget can violate memory safety and malicious code cannot be sandboxed.

SoftBound **[88]** Operates as *Hardbound*, with disjoint metadata, but adds software checks rather than adding hardware. Incurs an average 67% run-time overhead.

M-Machine **[26]** All pointers are "guarded", with a permissions field and segment length stashed in the top 10 bits and an out-of-band tag field. Pointers can then only be manipulated such that they remain in the same segment and also only dereferenced

within their segment. This is similar to CHERI, but the reluctance to extend the size of pointers severely limits the precision guarded by a pointer and the expressiveness of the permissions.

Arm Memory Tagging Extension **[9]** All memory locations are tagged with a "colour" and pointers have a corresponding colour stashed in the top bits, where they are ignored when determining the address. Spatial safety violations dereference a pointer outside of its original allocation. Hopefully, this memory is coloured with a different colour than the original allocation, triggering a hardware exception. This is a fairly invasive change, requiring full tagged memory support. The protection is only probabilistic and the small number of colours (16) make circumvention a matter of retries for the attacker, making this most useful as a debugging feature. Oracle's Sparc processor had a similar feature: *Sparc ADI* [1].

Arm Pointer Authentication **[103]** Uses cryptography to authenticate and verify pointers to detect corruption, for example to defend return addresses against Return-Oriented Programming attacks. The authentication code is stored in the top bits of the pointer. This significantly limits the number of bits, making this primarily a debugging feature unable to provide strong protection in the face of an attacker who can retry.

Califorms **[110]** Inserts poisoned bytes into memory between valid allocations, adding an instruction to poison bytes and metadata in cachelines to record this. This means attempts to overflow buffers are likely to trigger an exception, especially when combined with ASLR. However, this would prevent common C mechanics such as type-agnostic copying using `memcpy`, so they add a mechanism for it to ignore the exception. Requires changes throughout the processor and reports a 2% to 16% performance overhead.

Low Fat Pointers **[75]** Uses bounds compression (inspiring *CHERI-Concentrate*) to store bounds in the top bits of pointers. Also has integrity tags and different instructions to operate on pointers, making them capabilities. Unlike CHERI, privileged software is allowed to set tags. The extent of pointer compression imposes severe alignment constraints on capabilities.

Mondrian **[133]** Allows memory to be split into arbitrary-length segments down to word-granularity, with user-specific permissions for each. Various optimisations are suggested to allow rapid checking of dereference, including a sorted segment table to allow binary search, and sidecar registers to cache previously obtained permissions. This is good for inter-process protection, but does not give intra-process data protection, as no check is performed on dereference that a pointer is used to access the intended object.

Valgrind **[91]** A debugging framework that takes compiled binaries and converts them into an intermediate representation. This can then be instrumented with checks in a modular way: tools can be written as plugins that transform the intermediate representation. The `memcheck` tool can provide memory safety guarantees, including checking for buffer overflows. However, this approach is very expensive (around a 22× run-time overhead), so is used only for debugging purposes.

AddressSanitizer **[115]** Instruments code and the allocator to detect spatial safety violations by maintaining a map of inaccessible memory, allowing regions to be poisoned. Code is added to check this map on every memory access. Again intended as a debugging tool, it sacrifices the generality of *Valgrind* to reduce the run-time overhead to 73% on average. Zhang et al. performed additional optimisations to reduce this overhead to 63% [147] (although their baseline *AddressSanitizer* had a run-time overhead of 108%). *HWAsan* [72] uses *Arm MTE* to significantly reduce the *AddressSanitizer* memory overhead, but does not improve the run-time overhead.

PACMem **[78]** By using *Arm Pointer Authentication*, the authors are able to produce a memory sanitiser with a 69% geometric mean run-time overhead. This provides only probabilistic protection, so does not defend against an attacker who can retry.

Intel MPX **[99]** A now deprecated attempt to enforce memory safety as an extension to the x86 instruction set. It provided four bounds registers, manipulated with explicit bounds instructions, that registers could be checked against before being dereferenced. It incurred an average 50% run-time overhead, with 300% in the worst cases. It also had poor compatibility for C idioms and was bypassable, with an easy option to ignore bounds errors.

In-Fat Pointer **[142]** Like CHERI, uses hardware support to restrict and check bounds. However, uses the top bits of pointers (at least 16 bits) to index into a lookaside bounds table. This is to preserve interoperability with unmodified binaries, such as libraries. Spatial safety errors in these binaries can violate the safety guarantees. The approach sees significant hardware area overhead (60% LUT increase for CVA6) and a 12% *geometric* mean performance overhead across a range of benchmarks. Overall, the approach is very similar to CHERI, but sacrifices guarantees in the presence of unprotected code to avoid perturbing memory layouts.

AOS **[68]** Uses *Arm Pointer Authentication* for pointers on the heap (both spatially and temporally), using the resulting MAC to index into a lookaside bounds table. They see a SPEC run-time overhead of 8.4% in Gem5. However, their approach protects only the heap, does not allow bounds to be restricted, and relies on probabilistic protection due to the limited space for authentication bits in the top of pointers.

CODOMs **[124]** Tags instruction pages with the tags of data pages it can access. This does not protect spatial safety within an application, but can be used to

compartmentalise applications in the same address space. To allow sharing, code can create capabilities to delegate access to a region. Negligible performance impact was extrapolated from Gem5 microbenchmarks.

2.2.2.1 Summary of approaches

This section gives a summary of the key features of these approaches, highlighting their fundamental properties and limitations. For example, approaches that only change the compiler see easier adoption and no hardware overhead, at the expense of the security being bypassable, and increased software runtime overheads. In addition, protection of intentionality is omitted in some techniques. The different approaches are summarised in Figure 2.2. The following qualitative properties are used to categorise the schemes:

Software-only Schemes that avoid requiring hardware support may see easier adoption. However, such schemes require use of a particular compiler and so do not allow sandboxing of binaries from unknown sources.

Guaranteed Schemes that offer deterministic protection can provide strong runtime guarantees. Depending on the amount of entropy, attacking probabilistic schemes may be simply a matter of retries, making them primarily debugging features.

Mandatory Schemes can allow running untrusted binaries by being non-circumventable. This requires some mechanism for restricting access to arbitrary machine code in a way that cannot be reversed without some additional privilege.

Intentional Schemes can enforce intentionality to provide additional security: not only does the scheme check that an access is to an allowed region, but also the software must specify the region it was intending to access alongside the actual address.

C Compatibility Schemes vary in how much software change is required to support the scheme. This can range from being able to use the same binary, to requiring a recompile, to a complete source-code rewrite. It is worth noting that the purpose of any protection is to prevent certain unsafe behaviours: software that relies on these behaviours will therefore no longer work. As such, some C idioms may be forbidden, some of which rely on undefined behaviour.

One additional point is that schemes can be combined or enhanced to overcome their limitations and performance overheads. For example, adding hardware support for fast bounds checking or to track pointers may significantly accelerate a software-only approach. An example of this can be seen in the difference between *SoftBound* and *HardBound*. As well as being faster, moving from software- to hardware-enforced checks can allow code to be sandboxed. To a large extent, CHERI can be seen as a combination of these approaches.

	Software-only	Guaranteed	Mandatory	Intent.	C Compat.
CCured	✓	✓	✗	✓	Recompile
Cyclone	✓	✓	✗	✓	Rewrite
Fail-Safe C	✓	✓	✗	✓	Recompile
HardBound	✗	✓	✗	✓	Recompile
SoftBound	✓	✓	✗	✓	Recompile
M-Machine	✗	✓	✓	✓	Recompile
Arm MTE	✗	✗	✓	✓	Recompile
Arm PAC	✗	✗	✗	✓	Recompile
Califorms	✗	✗	✓	✗	Recompile
Low Fat Pointers	✗	✓	✓	✓	Recompile
Mondrian	✗	✓	✓	✗	Recompile
Valgrind	✓	✓	✗	✗	Binary
AddressSanitizer	✓	✗	✗	✗	Recompile
PACMem	✗	✗	✗	✓	Recompile
Intel MPX	✗	✓	✗	✓	Recompile
In-Fat Pointer	✗	✓	✗	✓	Recompile
AOS	✗	✗	✗	✓	Recompile
CODOMs	✗	✓	✓	✗	Rewrite
CHERI	✗	✓	✓	✓	Recompile
Rust	✓	✓	✗	✓	Rewrite

Figure 2.2: A summary of the key qualitative features of the C safety mechanisms discussed.

2.3 Temporal safety

Temporal safety violations confuse objects at the same memory locations but with different lifetimes. Computer systems generally have to reuse resources over time: most notably, virtual address space must be reused as objects are rapidly created and discarded. Temporal confusion can occur whenever objects are dynamically allocated, most commonly for C programs on the stack or the heap. On the stack, a violation consists of dereferencing a pointer to stack-allocated memory that has since been repurposed. This can occur when the pointer is passed outside of the context that allocated it. On the heap, the most common vulnerability is a use-after-free, where a pointer is used to access an object that has since been freed and the backing memory has possibly been repurposed [32]. This can leak or corrupt privileged state. As lifetime characteristics of stack and heap objects are usually very different, performant solutions to stack- and heap-based temporal safety can differ significantly. When discussing temporal safety, this thesis focuses on the heap. Like spatial safety, temporal safety violations can cause accidental bugs and exploitable vulnerabilities.

Some of the deployed protections discussed in the context of spatial safety in Section 2.2.1 also protect temporal safety. MMUs and MPUs again provide protection on a process

granularity, since mappings are destroyed when a process terminates. Safe languages also ensure temporal safety by never freeing objects with remaining reachable references.

Capabilities naturally solve spatial safety issues, as bounds within pointers prevent confusion between objects at different memory locations. However, temporal safety issues pose more difficulty. Traditional approaches to revocation in other computer science contexts involve maintaining a centralised list of revoked references. An example is certificate revocation for cryptography systems [70]. This works well when references are indirected through a central resource, so the reference can be rendered useless by deleting the backing entry in the central database. Capabilities do not naturally lend themselves to this approach, as they can be freely copied and are not indirected: holding the capability should be sufficient to access the referent object without additional layers of lookup. While this approach aids performance in the common case, it removes the ability to invalidate capabilities in bulk. This implies sweeping memory to revoke capabilities. How to achieve this with small performance impact is investigated in Chapter 7.

Many approaches to temporal safety in systems without capabilities are forced to approximate whether values are references to revoked objects, or incur a large performance penalty to track this information. By making it unambiguous which values are pointers and preventing pointers from being materialised from scratch, capabilities make it possible to sweep precisely.

2.3.1 Related research

Many attempts have been made to mitigate heap temporal safety vulnerabilities in C, including:

Boehm-Demers-Weiser Garbage Collector (Boehm GC) **[20]** A garbage collector for C that marks and sweeps to replace error-prone manual memory management with `malloc` and `free`. However, without capability metadata, it must be both conservative and unsafe. The conservativeness is required since pointers cannot be reliably distinguished from integers due to static typing, so all integers must be assumed to be pointers to avoid freeing live data. The unsafety is because it frees memory that cannot be reached by accessible pointers, but it is possible for pointers to be hidden in C, for example through `XOR` linked lists.

Capability Boehm **[60]** Investigates initial adaptation of *Boehm GC* to CHERI, using capability metadata to allow the conservativeness and unsafety to be overcome. The work is ongoing: the paper mostly highlights challenges adapting the existing code to support capabilities and difficulty coalescing allocated blocks due to bounds monotonicity. The potential benefits of capability tags available at run-time are noted but not yet exploited.

MarkUs **[2]** Reduces the unsafety of *Boehm GC* by freeing only the intersection of manually freed memory (which is kept in quarantine) and memory detected as unreachable by the mark and sweep. The run-time overheads are reported as 10% on average with a worst case of 100%.

MineSweeper **[44]** Uses a linear sweep, very similar to *CHERIVoke* [138], to ensure no dangling pointers remain before reusing memory. Not built atop CHERI, it can run on unmodified hardware but pointers can be hidden from it, so it does not guarantee safety. Sees a 5.4% geometric mean run-time overhead across SPEC.

Magpie **[104]** A transformation of C programs that allows them to be used for precise garbage collection. It automatically generates functions to feed into a garbage collector to perform traversal to find live variables and rewrites `malloc` calls as garbage collector calls. It requires pointers to use the correct type and has undefined behaviour in the face of pointer arithmetic. They experience up to 400% run-time overheads.

Fail-Safe C **[98]** Already discussed in Section 2.2.2. Since the compiler generates typed fat pointers with full information about bounds, this approach is able to use garbage collection to ensure memory is not reused until all dangling pointers are invalidated.

Arm Memory Tagging Extension **[9]** Already discussed in Section 2.2.2. As well as varying the colour across different allocations spatially, the allocator can change the colour when `malloc` reuses memory, meaning all dangling pointers to any old allocations will raise an exception on dereference. Once again, this protection is only probabilistic, as colours must eventually be reused.

AddressSanitizer **[115]** Already discussed in Section 2.2.2. As well as poisoning bytes surrounding structures for spatial safety, this scheme poisons an allocation when it is freed, catching use-after-frees unless the memory has been reused.

DangNull **[76]** Adds a pass to the compiler, registering all stores of pointer types to their corresponding allocation. `free` calls then invalidate all pointers registered to the freed allocation by replacing them with null pointers. It only tracks pointers on the heap, leaving all other locations, most notably the stack, open to contain dangling pointers. It incurs an average SPEC run-time overhead of 80%, with a worst case of over 400%, although they exclude the `omnetpp` benchmark, which is usually the worst case.

DangSan **[73]** Operates as *DangNull*, but also tracks pointers stored to the stack, preserves most of the address bits of invalidated pointers (improving compatibility), and makes efficiency improvements. It is still unable to track all pointers, for example those copied with `memcpy`, pointers in registers, and pointers spilled to the stack by

the compiler. They observe a geometric average of 44% run-time overhead in SPEC, with a worst case of 672%.

FreeSentry **[144]** This again operates similarly to *DangNull* (developed independently and published concurrently). It achieves a better run-time overhead (SPEC geometric mean 25%), at the expense of not supporting multi-threaded programs.

pSweeper **[81]** Similar to *DangNull*, but avoids maintaining points-to relationships with code instrumentation, instead just maintaining a list of live pointers with simpler instrumentation of pointer-typed stores. The approach then exploits additional cores to run concurrent sweeps through memory, checking if any pointers refer to freed memory. Freed memory is quarantined until a sweep is complete. Run-time overheads can be traded off against memory overheads, but average run-time overheads between 12.5% and 17.2% are observed on SPEC, with worst-case overheads on the order of 100%. Note that this uses an additional CPU core for the sweeps.

BOGO **[146]** Runs atop *Intel MPX* to provide temporal safety by scanning through bounds tables on every free, invalidating dangling pointers by restricting their bounds. To optimise, clears only the most accessed bounds tables, relying on page faults to invalidate others lazily. Reports an average SPEC performance overhead of 60%, with a worst case of 16× for `omnetpp`.

CETS **[87]** A compiler transformation that augments pointers with a key value and a pointer to a lock location, checking that the key value matches the value stored at the lock location on every dereference. Frees change the lock value, implicitly invalidating all dangling pointers. Incurs an average SPEC run-time overhead of 48%, with a worst case of 175% (`omnetpp` excluded).

Watchdog **[86]** Associates every pointer with a side-car register containing bounds and an identifier that indexes into a table of all allocations. All memory accesses check this table to ensure the pointer's identifier is still valid and matches the current allocation identifier, and that the access is in-bounds. The approach reports an average 24% run-time overhead on a simulated processor.

Oscar **[34]** Places objects on separate virtual pages, revoking them by removing the mapping for the corresponding page. The scheme never reuses virtual addresses for freed pages, so programs will eventually run out of virtual address space. Reports a 40% geometric mean overhead over the SPEC benchmarks, with a worst case of 350%.

Cling **[3]** Like *Oscar*, avoids virtual memory reuse in general. However, allows type-safe reuse by inspecting the call stack of the `malloc` call. This protects against common vulnerabilities, but does not guarantee safety. Due to the fast reuse path, has low overheads, negative in many cases.

***DieHard* [16]** Provides probabilistic protection against use-after-free errors by randomising heap allocation decisions. Combined with over-provisioning memory for the heap, this makes it unlikely for a dangling pointer to point to a useful new allocation. Reports a 12% geometric mean run-time overhead over SPEC 2000, with a worst case of 109%. Also offers a mode to maintain replica heaps with different seeds to detect non-determinism of results.

***DieHarder* [96]** Adapts *DieHard* to a new allocator, with added security features such as randomising small allocations and using a sparse page layout. Sees similar run-time overheads: 20% average and a worst case of 100%.

***FreeGuard* [118]** Combines various probabilistic protection techniques, based on *DieHarder*. However, makes tradeoffs in favour of performance over security, such as making large `mmap` system calls and avoiding over-provisioning the heap size. Average run-time overheads of 1.8% are reported, with a worst case of just over 20%. They report multi-threaded benchmarks rather than SPEC.

***CRCount* [117]** Instruments code to maintain a bitmap of the locations of pointers on the heap. This is used to detect when a store overwrites a pointer, at which point a reference count to the pointee allocation is decremented. Allocations are freed once their reference count reaches zero. Incurs a geometric mean run-time overhead of 22% over SPEC, with a worst case of approximately 130%.

***Retrofitting Temporal Memory Safety on C++* [18]** Bikineev, Lippautz, and Payer modified Google Chrome to improve temporal safety by quarantining freed memory and scanning for dangling pointers before reuse. The authors use *Arm MTE* to allow multiple reuses of the same memory—one per MTE colour—before a sweep is required. This reduces the run-time overhead from 8% to 2%. Once again, due to pointer hiding, it is possible for this approach to miss dangling pointers. In addition, the sweeps do not detect dangling pointers moved during a sweep.

2.4 CHERI

This section introduces CHERI, giving the design principles and architecture-neutral model, as well as common microarchitectural implications. I joined the CHERI project with these aspects initially developed via the CHERI MIPS processor, and a preliminary proposal for adaptation to RISC-V. Throughout the course of my project, architectural changes were investigated, as discussed throughout the thesis. More detailed descriptions of instructions, as required to understand the content of the thesis, are given in Appendix A. The CHERI ISA document provides a more complete reference [128].

2.4.1 Model

CHERI is an instruction set extension, changing an existing ISA by requiring memory accesses to be performed through tagged capabilities, and adding instructions for inspecting and manipulating capabilities [136]. Starting as an extension to the MIPS ISA [136, 137], CHERI has seen increasing interest throughout this project, for example in Arm's Morello experimental prototype, adding CHERI to a commercial Arm processor [54]. There is also a draft CHERI x86 proposal in progress [128].

Capabilities provide security by enforcing the principles of least privilege [112] and intentional use [93]. By minimising the memory accessible to code at run-time, accidental or maliciously influenced accesses have limited scope to corrupt program state. In addition, requiring every access to quote the intended capability prevents code from performing an operation using authority intended for a different operation, mitigating confused deputy attacks [56]. By making capability use mandatory, enforced by hardware, rather than optional checking instructions, CHERI also allows compartmentalisation and sandboxing. For example, with careful page-table management, even the operating system cannot access a process' memory without access to an appropriate capability [45]. This thesis primarily only considers spatial and temporal safety benefits of CHERI, but note that its benefits extend beyond this.

In practice, CHERI protects the integrity of pointers with a one-bit out-of-band tag. The hardware guarantees that every tagged capability must have been derived from an initial root capability by a sequence of permissible operations, each of which can only monotonically decrease the rights granted by that capability. This ensures all tagged capabilities have strict provenance. The tags prevent corruption of pointers, immediately mitigating a large class of attacks that require manipulating pointers by overwriting them with non-capability data, for example Return-Oriented Programming [108]. In turn, by requiring every memory access to quote a tagged capability, this protects data from accidental or malicious corruption, since the capability contains the intended bounds and permissions. This too immediately mitigates some of the most common attacks, most notably buffer overflows [77].

A study by Microsoft revealed that roughly 75% of their security bugs are bounds overflows or temporal safety issues, indicating they would be mitigated by CHERI with temporal safety support [65]. This provides assurance that CHERI can have a real impact on industry and users. In addition, monotonicity of the ISA has been formally proven, showing that the intended invariants are provided by the architecture, if implemented correctly [15].

2.4.2 Microarchitecture

CHERI is able to perform dynamic bound-checks while performing better than the software approaches mentioned in Section 2.2.2 by keeping the bounds information in the pointer and performing the bounds-check in hardware. The metadata is in-band, treated throughout the processor as data. The validity tag is out-of-band and accompanies each capability-aligned granule of data everywhere capabilities can be stored, including registers, caches, and memory.

The bit pattern of a capability consists of:

A capability tag A one-bit out-of-band field that records whether the capability is valid, i.e. whether it has been derived only via legal monotonic operations.

A cursor A field giving the current address referred to by the capability. This is all that is present in a traditional C pointer and the natural integer interpretation of the capability.

Bounds information The capability only grants access to a single contiguous region of memory, specified by the bounds field. Changes to the capability can only ever grant access to a subset of the bounds without triggering an exception or clearing the tag.

Permissions Permissions are required to access memory in different ways: most obviously read, write, and execute, but also the permissions to read and write capabilities, and more experimental uses. These can only be modified by a bitwise `AND` operation, inherently guaranteeing monotonicity.

Object type (`otype`) CHERI provides a compartmentalisation mechanism for granting protected access to objects, represented as a pair of code and data. Code and data can be sealed with an `otype`, and primarily only unsealed by the `CInvoke` instruction, which atomically jumps to the code pointer and installs the unsealed data capability. Sealed capabilities cannot be mutated other than via a legal `CInvoke` or `CUnseal` instruction without raising an exception or clearing the tag.

Two major complexities when implementing CHERI are now discussed: compression and changes to the memory subsystem to support tagged memory.

2.4.2.1 Compression

One of the major performance overheads of CHERI comes from the increased widths of capabilities compared to traditional pointers. This uses hardware resources, as the register file is bigger, but also adds cache pressure, as pointers in memory must be accompanied by metadata. These effects can be somewhat mitigated by capability compression [135], which reduces the bounds metadata to as little as 39 bits for a 64-bit pointer at the expense

of alignment constraints on representable positions for large objects. The compression and decompression is performed directly by the hardware, allowing software to be largely unaffected.

Capability compression, described in detail in our *CHERI-Concentrate* paper [135], stores the bounds as a base (B) of b bits, a top (T) of $(b-2)$ bits, and an exponent (E) of e bits. The described bounds are relative to the cursor of the capability (C). The actual base is determined by shifting B left by E, filling any remaining upper bits with the corresponding most significant bits of C. The top is similar, except the top two bits of T are determined in the common case by adding 1 to those of B. As a result, bounds are zero-filled in the least significant E bits, imposing the requirements on bound-alignment mentioned above. To save some space, E is only stored when non-zero, in which case (called the *internal exponent* case) it takes the place of the least significant bits of B and T (which are then implicitly zero). This means smaller objects can be represented more precisely than larger ones. The scheme is currently implemented with $b = 14, e = 6$ for a 64-bit address space and $b = 8, e = 6$ for a 32-bit address space.

Bounds with insufficient alignment to be represented as above are said to be *unrepresentable*. CHERI offers two variants of the `CSetBounds` instruction: one triggers an exception when unrepresentable bounds are requested, the other just returns the closest larger representable bounds. In order to guarantee spatial safety, software must add padding (at most 0.2% or 12.5% of the size of the object for 64-bit and 32-bit address spaces respectively) at the beginning or end of large objects to ensure no other objects use the memory that is within the bounds due to the rounding. Instructions that modify a capability cursor may also change the interpretation of the bounds if they modify bits of the cursor that are used to determine the base and top. Care is taken in the ISA and implementations that such cases clear the tag of the resulting capability or raise an exception.

2.4.2.2 Memory subsystem changes

Since capabilities can be transferred to and from memory, the capability tags must be preserved in memory. These tags must be out-of-band, as code should not be allowed to circumvent the capability system by directly writing the capability tag. In addition, the tag must be voided whenever the address is written in a way that might corrupt the bits of an existing capability.

Storage is added for capability tags alongside the data in registers and in caches. CHERI adds a shadow-space for the tags in memory, only accessible by a new tag controller component [63]. The tag controller is inserted into the cache hierarchy after the highest level of cache. It extracts the addresses of requests passing through, uses them to lookup the relevant tags in the shadow-space, and merges these into responses as they go back into the cache. This component also maintains a hierarchical cache of tags, reducing the pressure on memory that would result from two memory accesses per cache miss. An

alternative implementation is to extend DRAM by one bit per capability-sized region of memory. Morello has this option, using Error Correction Code (ECC) bits of DRAM, but we avoid it to allow us to use consumer DRAM. The tag controller also helps to accelerate temporal safety by making tags accessible independently of the data, as discussed in Section 7.3.

2.4.3 Software

C code can simply be recompiled to gain many of the memory safety benefits of CHERI. This is because C contains information on the bounds of the objects within which pointers are intended to be dereferenced. This information is traditionally discarded during code generation. For example, the type of a stack allocation indicates its size: the compiler already needs to know this to generate code and lay out the stack. Other structures, such as globals and thread-local storage, can be accommodated similarly. The LLVM compiler has been augmented by the CHERI group [36] to preserve this information and generate the relevant bounds restrictions for the MIPS, Arm, and RISC-V backends via the CHERI MIPS, Morello, and CHERI RISC-V extensions. The last of these was developed alongside the CHERI RISC-V ISA, OS, and hardware: a co-design process of which this PhD is a part. Heap allocations—via `malloc`—also contain the description of the intended size in the allocation call. By linking against a CHERI-aware library and using its `malloc` implementation, the application can no longer get confused between allocated objects. Ideally, a CHERI `malloc` implementation would also provide heap-based temporal safety: this is discussed in Chapter 7.

Some code does need to be changed for compatibility and for protection [129]. Some C code assumes sizes of pointers and freely casts pointers to integer types and back to perform arbitrary arithmetic on them (such as `XOR` linked lists), despite the C standard deeming this undefined behaviour [111]. With CHERI, this clears the tag and corrupts the metadata: fixing the code often involves using the correct type throughout [129]. Any code that takes pointers significantly out-of-bounds also causes compatibility issues, as capabilities may be taken outside of their representable region. This is, however, not compliant with the C standard [111], and CHERI does support pointers being taken somewhat beyond their bounds for common cases. Enabling sub-object bounds—automatically restricting pointers to `struct`s when a reference to a member variable is taken—causes more compatibility issues. For instance, imprecision caused by capability compression may force sub-object capabilities to overlap, and use of various `container_of` macros would violate monotonicity [107]. As a result, sub-object bounds are optional in the compiler. Additionally, some code may wish to add additional capability protections, for instance if implementing a custom allocator. Capability compiler intrinsics are available for this purpose. Finally, supporting compartmentalisation requires more software work: various models are being actively researched by the CHERI team.

The OS can be modified to varying degrees. Initially, the operating system can just change its interaction with user-level applications, for instance preserving capabilities passed through the kernel and saving and restoring capability state on context switch. This is referred to as a hybrid kernel and is fully supported in CheriBSD [23]—an adaptation of FreeBSD—and Cheri-FreeRTOS [24]. The most complete approach is a purecap kernel, where the OS itself uses capabilities for all its pointers: a prototype version of CheriBSD with a purecap kernel has been developed.

Chapter 3

CHERI for microcontrollers

This chapter discusses the development of CHERI for RISC-V microcontrollers: in particular the Piccolo and Flute cores. As the first CHERI RISC-V hardware implementations, this required significant refinement of the draft CHERI RISC-V ISA. Some of the key architectural decisions are discussed in the context of their impact on microarchitecture. An overview is given of the microarchitectural work required to implement the extensions. The TestRIG and RISC-V Formal Interface with Direct Instruction Injection (RVFI-DII) frameworks that were developed alongside the processors to accelerate debugging and verification are also introduced.

The implementation discussed in this chapter directly addresses the question of amenability of CHERI for RISC-V microcontrollers (Hypothesis H.1). The success of the implementations is evaluated using FPGA synthesis tools and embedded benchmarks in Chapter 4. The microcontroller implementations also act as a baseline to compare against the application-class CHERI implementation discussed in Chapter 5. It therefore enables an investigation of CHERI scaling across core sizes to address Hypothesis H.3.

3.1 Characteristics of microcontrollers

Many applications call for a small, low-power core, where performance is not a primary goal, for instance in small Internet of Things (IoT) devices or for managing state-machines. In these cases, a microcontroller is used, typically running an embedded Real-Time Operating System (RTOS). Such cores have different characteristics and optimisation goals to application-class cores. For example, it is often the case that microcontrollers lack the traditional memory subsystem consisting of DRAM and caches. Some instead have a small scratchpad of on-chip memory [12]. Microcontrollers also often lack MMUs.

The microarchitecture of microcontrollers also tends to be simple, prioritising low area, for example using a scalar in-order design with a small number of pipeline stages.

Microcontrollers also present a different security landscape to application-class cores. Typically the application is known at design-time, so formal verification of the entire codebase is more feasible. For example, the SeL4 RTOS has been formally proven to provide isolation between processes [69]. However, microcontrollers are often used in low-cost environments, meaning the proper security scrutiny might not be affordable. Such low cost devices are increasingly networked for IoT reasons, making bugs exploitable to compromise homes, hospitals and traffic control devices, among others [101]. Traditional MMU-based compartmentalisation is not possible where an MMU is not present, alarmingly making the entire address space visible to every process. The current solution is to employ a MPU, practically offering a subset of MMU protections. Section 2.2.1.2 describes the PMP mechanism, which is RISC-V's specification for an MPU.

3.2 Baseline processors

This section describes the existing baseline processors, going into details relevant to the CHERI modifications.

Rather than implementing the processors from scratch, we augment pre-existing processors with CHERI extensions. As well as saving significant effort, this more accurately represents the path industry would take to adopt CHERI, since many have a large amount of pre-existing processor Intellectual Property (IP). In addition, using pre-existing processors gives more realistic baseline performance, as the processors have been designed and optimised without capability support in mind. Developing a CHERI processor completely from scratch could present different optimisation opportunities, but is left as future work. The particular processors chosen offered an interesting range of configurations and sizes. They were also implemented in Bluespec, making them extensible and allowing reuse of certain existing CHERI MIPS components and logic.

The baseline microcontrollers used—Piccolo and Flute—are developed by Bluespec Inc. Piccolo is a 3-stage in-order pipeline, supporting 32-bit and 64-bit RISC-V, as well as (all sensible combinations of) the RISC-V A, C, D, F, and M extensions. Flute is an extension of Piccolo to add two front-end pipeline stages, increasing clock speed at the expense of reducing Instructions Per Clock (IPC). Flute and Piccolo are separately maintained, although most changes are kept synchronised between the two processors, minimising the difference between them.

Piccolo's pipeline is shown in Figure 3.1. The first stage (`Stage1`) is a combination of fetch, decode and execute. The incoming instruction is decoded, and the Arithmetic Logic Unit (ALU) operations implied by it are carried out, before the calculated next PC is used to start the fetch of the next instruction. This avoids the complexity of predicting branches as the architecturally next PC is known when it is needed for fetch, at the expense of clock frequency as fetch, decode, and execute are all serialised. The second stage (`Stage2`)

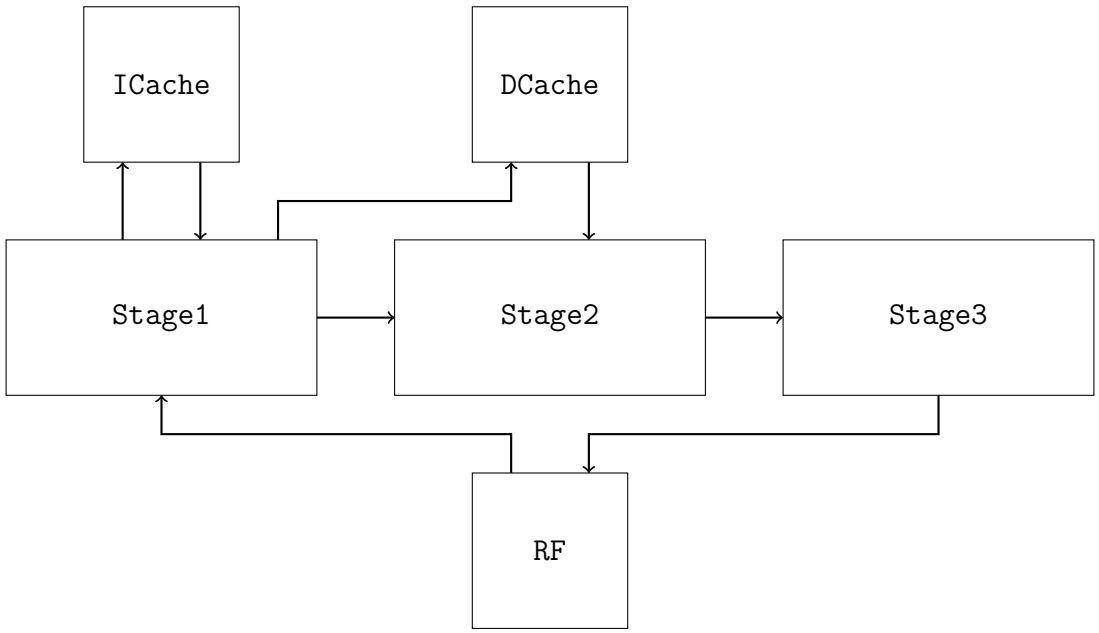

Figure 3.1: The Piccolo pipeline. Forwarding paths from `Stage2` and `Stage3` to `Stage1` are omitted.

is devoted to any operation that might be multi-cycle: multiply, memory access, shifting (if not configured with a barrel shifter), and floating point (if enabled). The third and final stage (`Stage3`) is for register writeback. All reasonable forwarding paths are present, from both `Stage2` and `Stage3` to `Stage1` where the registers are read. The control-path detects any exceptions, interrupts or bubbles, handling each possibility once the following stages are clear.

Despite ostensibly being microcontrollers, the processors were designed around offering an interface to DRAM, with a small write-through L1 cache. Over the course of the project, support was added for additional configurations, including changing to write-back L1 caches and adding an L2.

The default cache is in-order, write-through, and no write-allocate. This simplifies the design as all lines are always clean. The cache accepts requests for reads, writes or atomics, indexing into its BRAMs as it receives the request so it can service the request after a single cycle on hit. The following cycle, the cache issues the response to the pipeline if ready, and starts generating AXI transactions as required. RISC-V's atomic operations (described in Section 2.1.1) are also supported: the caches perform the required operation on either a hit or when the relevant response is received from memory. Atomicity follows from the in-order nature of the caches.

The baseline System on Chip (SoC) was provided by Galois as a Vivado project, consisting of interconnect between the processor, DRAM and various peripherals, including Ethernet, UART, and DMA. However, partly due to the lack of RISC-V cache maintenance operations at the time of implementation, the memory map was split into a cached 3 GiB and uncached 1 GiB region. This requires any DMA to be performed into the uncached region, then

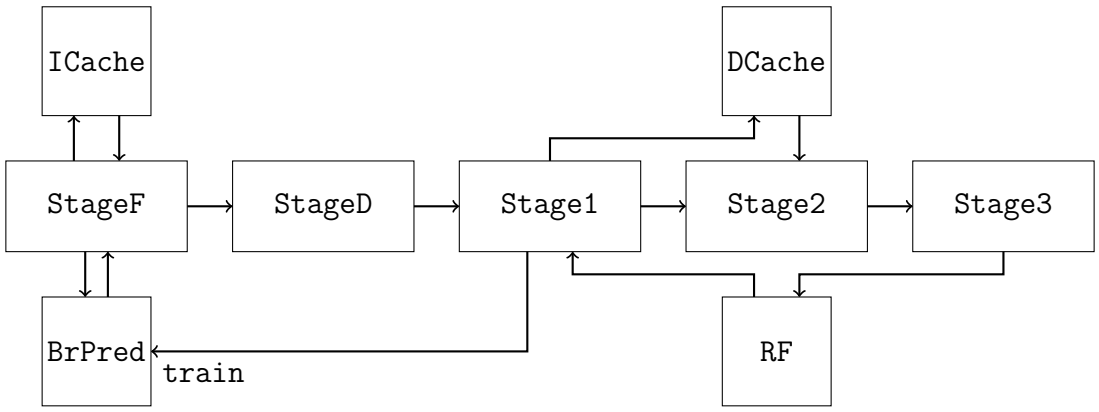

Figure 3.2: The Flute pipeline. Forwarding paths from `Stage2` and `Stage3` to `Stage1` are omitted.

copied by the CPU into the cached region, somewhat defeating the performance goals of DMA.

Flute (Figure 3.2) splits Piccolo's first stage into three: `StageF`, `StageD`, and `Stage1`. `StageF` performs instruction fetch, receiving the fetched instruction, feeds it into the branch predictor, and requests the next instruction. `StageD` only serves to decompress compressed instructions, with all other instruction decoding deferred to switch statements in the ALU, presumably to maximise logic shared with Piccolo. `Stage1` is the remainder of the corresponding stage in Piccolo, completing the decoding of the instructions and performing ALU operations.

Flute also adds MMU support via the additional S-Mode privilege ring, implying support for translation[1]. A TLB is offered to make performance tolerable, but optimisation is lacking. One particular quirk is that the TLB does not lookup in the cache on miss, instead always performing the full access to DRAM. This significantly increases the performance penalty of a TLB miss. As the caches are write-through, this does not cause coherence issues.

Since Flute adds a fetch and decode stage, but branches are still resolved in `Stage1`, there are two cycles during which the fetched instructions are speculative. Flute therefore adds a branch predictor, consisting of a Return Address Stack (RAS), Branch Target Buffer (BTB), and 2-bit saturating taken predictor. The RAS detects calls and returns using RISC-V-defined hints based on the Application Binary Interface (ABI). The BTB is a direct-mapped table containing the last address jumped to from each PC. These structures are trained from `Stage1` based on the architecturally correct behaviour.

[1]While this support is actually included in Piccolo, it is not part of the recommended configuration.

3.3 Architectural changes

Adding CHERI to a processor involves adding extra instructions to manipulate capabilities, adding checks to both new and existing instructions to ensure security guarantees are met, and augmenting registers, datapaths, caches, and interconnect to support wider registers and capability tags. I implement Version 8, as described by the CHERI ISA document [128].

Pending standardisation, CHERI has been added as a largely greenfield RISC-V extension, meaning it uses only encoding space reserved for non-standard extensions. In particular, CHERI uses the `0x5b` major opcode for all added instructions. This space is not completely filled: in total, fewer than 10 million new encodings are added, less than one percent of the total 32-bit encoding space [128]. The majority of these are taken by the two added I-type instructions: `CIncOffsetImm` and `CSetBoundsImm`. In addition, CHERI re-enables the encoding for the load/store access width that is double the current `xlen` (including for atomics) to enable capability-width accesses: 64 bits for RV32 and 128 bits for RV64[2].

Furthermore, CHERI adds `xcause` codes for capability-related exceptions and `xccsr` registers for controlling and inspecting capability-related state. Page-table bits are added for page-granularity capability management: these are most relevant to revocation so are discussed in detail in Section 7.2.1.

Although the current architecture has been validated in hardware and software, architectural optimisation—deciding which CHERI instructions are required, which need immediate forms, and optimising for efficient decoding—is work yet to be carried out. It is hoped that the CHERI-augmented processors can be used to enable this work.

As the first CHERI RISC-V implementations, Piccolo and Flute offered the first opportunity to investigate the microarchitectural implications of several pre-existing CHERI architectural questions, described in the rest of this section.

3.3.1 Merged register file

CHERI introduces registers to hold capabilities. These are twice the bit-width of the addresses in the system, to allow room for the capability metadata [128]. This raises a question of how these capabilities should be arranged and addressed. One option (the split register file preferred by CHERI MIPS) is to have the capability registers completely separate from the integer register file. Instructions are then explicit about whether their register arguments index into the capability or integer register file. An alternative is to have some overlap between the two (the merged register file) and define semantics for reading capabilities as integers if the instruction expects an integer, and writing back capability registers with integers if an instruction does not produce a capability. Note that

[2] A specification for supporting 128-bit registers is being developed for RISC-V.

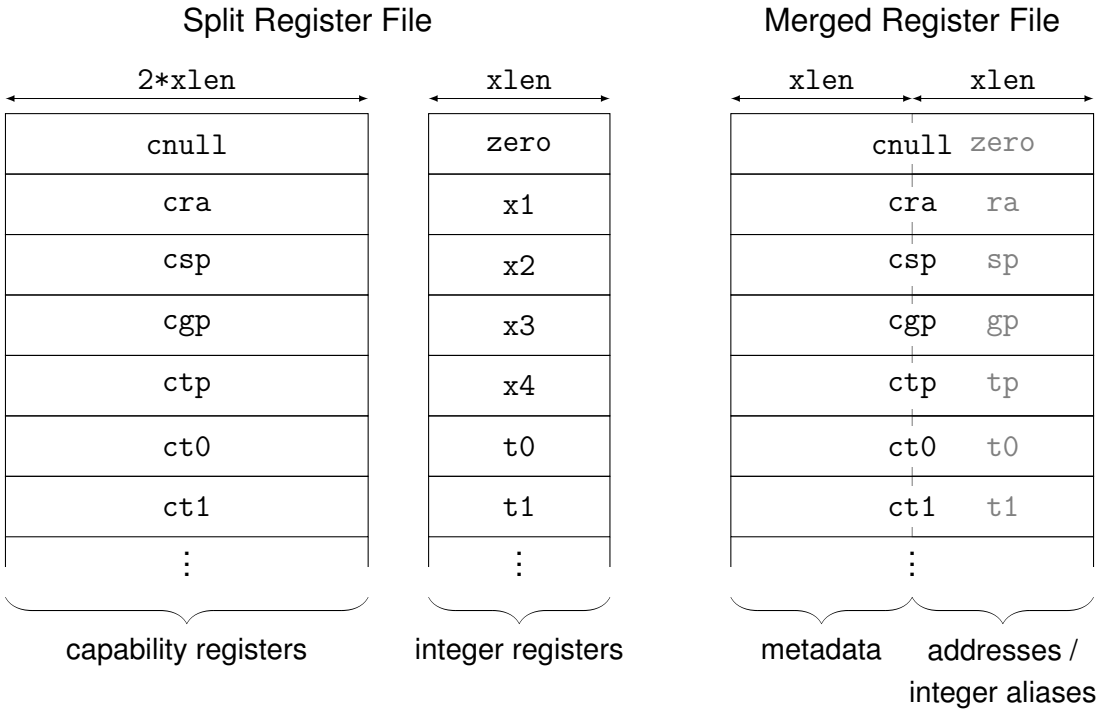

Figure 3.3: Architectural view of integer and capability RISC-V registers with split and merged register files as seen by pure capability software. ABI names are given: hardwired zero (`zero`), return address (`ra`), stack pointer (`sp`), globals pointer (`gp`), thread pointer (`tp`), temporaries (`t0`,`t1`). Capability versions are prefixed with `c`. With the split register file, integer address ABI registers are redundant, so could be reused as other integers, such as to provide more temporaries.

not all integer registers need to be extended with this approach, although this is an idea that remains largely unexplored. Figure 3.3 illustrates the difference in the architectural view of registers between the two approaches.

The decision has far-reaching consequences for microarchitecture, compilers, and software.

A merged register file uses fewer state bits, for any given number of capability and integer registers, as bits are shared between capabilities and integers. This saves area in the processor, especially when synthesising for silicon, where register overhead is not concealed by BRAMs. Microcontrollers are particularly sensitive to area increase due to additional registers, since registers correspond to a large fraction of their area.

In addition, the merged register file should reduce area and complexity in the processor, as it allows the datapath, forwarding paths and control logic to be reused for capabilities. For some processors, the datapath will need to be widened to accommodate capability metadata. However, for processors that already need wide data paths, such as 32-bit processors supporting double-precision floating point (i.e. 64 bits), or processors supporting vector instructions, the existing wide data paths may be reusable, possibly reducing the CHERI area overhead. More complex microarchitectures rename registers to avoid false dependencies. The merged register file allows this mechanism to be reused for capability

registers without modification, as discussed in Chapter 5.

Furthermore, the RISC-V architecture encodes its registers into constant bit-positions in the instructions, encouraging simple in-order microarchitectures to index the register file very early in the pipeline. A split register file would therefore likely require an additional parallel register-read, as the instruction has not been decoded before register indexing must begin. This would impact power, area, and potentially timing due to the additional multiplexing required. A merged register file avoids this issue.

For CHERI RISC-V, we choose to implement the merged register file for these reasons. To preserve the principle of intentional use, the interaction between capability accesses and integer accesses to the registers must be carefully specified. We choose to have integer instructions read from the address of the capability register, preserving their interpretation as integer addresses, with writes producing a `null` capability with the address set to the intended integer. This avoids data corruption manipulating pointer dereference, the very vulnerability capabilities protect against.

3.3.2 Encoding mode

The CHERI instructions to manipulate and inspect capabilities represent a small portion of the opcode space, since most need only encode up to two registers and no immediates. Only the `CSetBounds` and `CIncOffset` instructions have immediate forms, as they are commonly used in code to manipulate and access the stack. However, in order to allow incremental adoption, CHERI provides two modes for accessing memory: legacy and capability. Legacy instructions allow dereference of integers relative to a Default Data Capability (DDC), primarily allowing legacy code to run unmodified in a sandbox. Capability mode requires pointers to be provided as capabilities, allowing finer grained protection and ensuring intentional use of the relevant bounds and permissions. CHERI MIPS supported both sets of instructions. However, memory instructions offer large immediates to allow accessing multiple values at static offsets from a register—most commonly stack-allocated data—without dynamic overhead. These large intermediates mean providing both legacy and capability memory instructions wastes vast instruction encoding space to avoid the performance impact of depriving either mode of immediate instructions. Since it is anticipated that code will generally run using one set of instructions or the other, switching infrequently, this use of opcode space seems wasteful. Although RISC-V supports arbitrary-length instructions in principle, encoding space is still at a premium: instruction cache pressure is reduced by keeping instructions short and it would be undesirable to require capability-supporting processors to incur the complexity of fetching and decoding 48-bit instructions. Instead, the CHERI ISA proposes that the same instructions could be used for both applications, interpreting its arguments based on a capability encoding mode [128]. As well as memory instructions, this also applies to the `auipc` instruction, which requires a capability version (`auipcc`) and has a 20-bit immediate.

As part of designing the ISA alongside the Piccolo microarchitecture, we decided to implement a mode-switching mechanism for CHERI RISC-V. There are many options for changing encoding mode, each having varying tradeoffs relating to ease of pipelining, interaction with the branch predictor, and resistance to confused-deputy attacks. In particular, a mode switch mechanism based on predicted state could require pipeline flushes, as instructions have been decoded based on the wrong mode. For security, if code is tricked into running in a different mode than expected, it may use its privilege in an unintended way.

A mode-switch instruction is the most explicit, keeping the encoding mode state in a CSR. This avoids any confusion about when a mode-switch is intended unless control-flow has already been compromised. However, this does not allow a jump-target to ensure what mode it is run in without adding a mode-switch at every entry-point. One option is to have separate capability execute permissions: one for run-as-legacy, and one for run-as-capability. This allows a function to guarantee it can only be run in the intended mode by clearing the appropriate permission bit in capabilities used to call it. As an alternative, the Program Counter Capability (PCC) could have a bit added that is not monotonic, but requires an explicit instruction to switch. This does not provide the same guarantee, but at least requires explicit manipulation of the PCC to change the target encoding mode. Both of these approaches use an additional bit in the capability encoding, which is expensive as these bits exist for every capability in the system. Another option is the least-significant bit of the target address of a jump[3]. Unfortunately, this bit is often used to stash other information in pointers, and also may be corruptible from software, so does not ensure the same level of intentionality.

We decided to use a mode bit in the PCC to determine the encoding mode, implementing this as a new `flags` field that can be freely changed without violating monotonicity. `flags` can be freely read and written in a capability using dedicated instructions, provided the capability is not sealed. Since return capabilities are sealed as Sentry capabilities (see Section 3.3.3) when linked by jump instructions, this prevents the interpretation of the returned-to code being altered by the callee of a jump. The encoding mode bit of `flags` is interpreted by hardware when the capability is installed as PCC. This means that the encoding mode potentially has to be predicted, as the architectural PCC may not be known in time for decode.

[3]This is the approach taken by Morello.

3.3.3 Secure Entry capabilities

Over the course of the project, a new capability type was added to the CHERI ISA: the Secure Entry (Sentry) capability. These are capabilities that are sealed, and cannot be unsealed except when used as a jump target.

This allows more scalable compartmentalisation than the existing object type mechanism, as compartments do not require their own value in the type space. Compartment-specific data can still be recovered, albeit with a higher performance overhead than the existing `CInvoke` mechanism, such as by storing a data capability alongside the called code.

Another key benefit to this mechanism is the observation that linked addresses, such as from the RISC-V `jalr` instruction, are only ever expected to be jumped to unmodified. As such, linked addresses can be sealed as Sentry capabilities automatically by the processor, without causing exceptions in the expected case. This has the significant security benefit of completely mitigating attacks that rely on corrupting return addresses, such as return-oriented programming.

To validate the approach and enable performance and compatibility measurement, I implemented Sentry capabilities across the cores. The implementation did not imply any particular microarchitectural challenges in terms of area or frequency. However, the additional "kind" of sealed capability required care to handle correctly. This motivated a type-safe treatment of sealing types in the Bluespec source that forces all possibilities to be considered. In addition, a few aspects of the implementation were error-prone checks, for example the need to check that the immediate offset is zero when jumping to a Sentry capability, causing significant testing overhead.

3.3.4 CHERI-optimised compressed instructions

As discussed in Section 2.1.1, RISC-V improves code density by specifying a set of compressed 16-bit instructions, each of which expands into a single 32-bit instruction. To maximise impact, the compressed instructions are intended to be the most commonly used, meaning a large overlap with compiler-generated blocks such as function prologues and epilogues. For example, RISC-V specifies the `c.addi4spn` compressed instruction, which increments the stack pointer by an immediate number of words. This allows the stack to be grown and shrunk with minimal instruction bytes for small stack frame sizes (up to 128 words), reducing the code size for this common part of prologues and epilogues.

The CHERI RISC-V extension specifies compressed versions of commonly-used CHERI instructions, as capability manipulation features heavily in common compiler-generated code. To allow the code-size reduction for both integer and capability code, the decoding of the compressed instructions is mode-bit dependent. Thus, while `c.addi4spn imm` expands to `addi sp, sp, 4*imm` in integer mode, it instead expands to `CIncOffsetImm`

`csp, csp, 4*imm` in capability mode, where the stack pointer is by default a capability. This mitigates the code density reduction from operating on capabilities instead of integer pointers, reducing pressure on the instruction cache, thus improving performance.

While the specification of which instructions to compress was carried out by others on the CHERI team, I implemented the change for the Piccolo, Flute, and Toooba cores. Mostly, these posed little microarchitectural difficulty, simply hooking into the existing mechanism in the decode stage to expand compressed instructions into their decompressed aliases. However, this does imply that the decode stage now requires the capability encoding mode (as discussed in Section 3.3.2) within the `flags` field of PCC in order to perform the decompression correctly. For the Piccolo pipeline, the architectural PCC is fully known at decode, but in Flute, this is not known until execute, the cycle after. This implies adding the capability encoding mode to the predicted state, flushing due to a misprediction on this bit as well as the predicted PCC address. Initially, the encoding mode is predicted as not changing between instructions, pending investigation of the performance improvement that would be gained from fully predicting it using the same infrastructure as for the rest of the PC. This relies on further investigation into software compartmentalisation models to indicate likely mode-switching frequency.

3.4 Microarchitectural implementation

I have carried out all of the work of adding CHERI (with merged register file) to both Piccolo and Flute.

3.4.1 Capability decoding

CHERI compresses capabilities to mitigate large memory footprint due to increased pointer size [135]. An introduction to the compression format is given in Section 2.4.2.1. Capability decompression introduces some pipeline complexity, as capabilities must be decompressed before they can be used in the ALU or bounds checked for memory access. Our *CHERI-Concentrate* paper proposed a method of decompressing these in multiple stages [135], which was implemented by CHERI MIPS. While moving to RISC-V, similar decoding functions were required, but also needed to be shared across multiple processors. As such, we implemented a typeclass describing operations that can be performed on a capability to wrap the existing logic. Thus, different stages of decompressed capability can be described. Operations supported at each level of decompression can be implemented, while the more advanced operations either have inefficient implementations or give errors, as required. Transformations between the different forms of the capability can be described and explicitly called as a type cast at the desired location in the pipeline. This proved very useful for experimenting with decompressing in different stages in the pipeline to

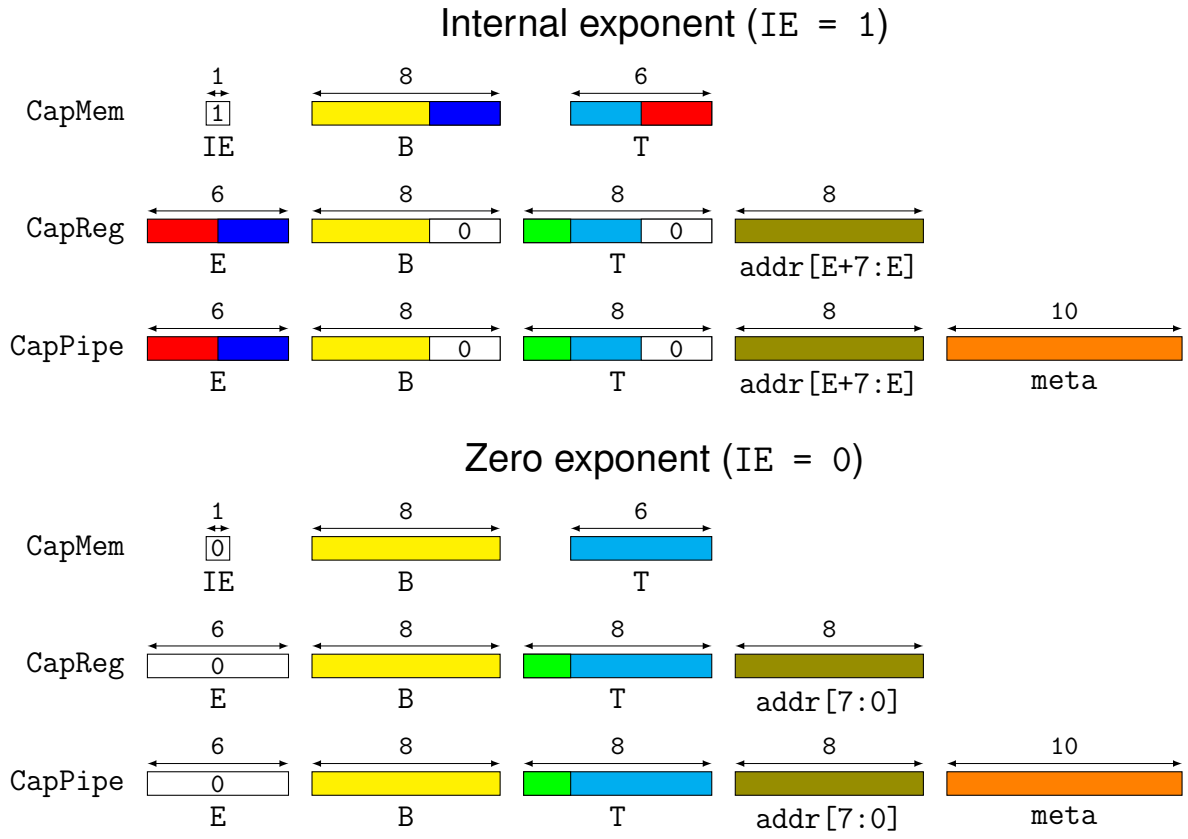

Figure 3.4: Three stages of decompression of capability bounds as they enter the pipeline (RV32). Both internal exponent and zero exponent cases are shown. As per the *CHERI-Concentrate* algorithms [135], the top two bits of `T` are restored based on the top two bits of `B` and some comparisons, `addr` bits are selected out of the capability address (omitted from diagram) based on `E`, and the `meta` bits are computed based on comparisons to determine the representable region.

optimise timing and area. In addition, it avoided common work having to be performed for the Toooba core: the typeclass and its instances were able to be reused without major modification.

In particular, we define three instances of the typeclass: `CapMem`, `CapReg` and `CapPipe`. Figure 3.4 shows the details of the stages of decompression. `CapMem` is a capability exactly as encoded in memory, thus consisting of (`2*xlen`) bits (plus a validity bit). `CapReg` is a capability as stored in registers, containing the extracted exponent (based on the `IE` bit, either zero, or stashed in the least significant bits of `B` and `T`), and using this exponent to extract the bits of the `address` field to which the bounds are relative. The width of a `CapReg`, excluding tag, is 80 bits in RV32 and 150 bits in RV64. The unpacking from `CapMem` is performed on load. Unpacking is relatively expensive, requiring a variable shift of the address based on the exponent field. `CapPipe` is a fully decoded capability that can then be used to extract the fields to implement the required architectural instructions directly. This is 90 bits wide for RV32 and 160 bits for RV64. This contains the additional `meta` field, which allows cases relating to wrapping the representable region

to be identified. The details of this field are not relevant to the discussion of Piccolo and Flute's implementation, but further information can be found in our *CHERI-Concentrate* paper [135]. The unpacking from `CapReg` is performed as the registers are loaded in the decode stage. This is cheaper than the `CapMem` to `CapReg` decompression as it only requires arithmetic on the highest three bits of the fields.

Care has to be taken that the redundant decompressed fields are correct after each capability manipulation operation, as otherwise they could cause incorrect results when forwarded between instructions. This requires recomputing some of the changed temporary fields in the ALU.

In addition to reusing the compression operations used by CHERI MIPS, the CHERI RISC-V implementations required additional efficient functionality that meant directly interacting with the (partially) compressed capability. For example, RISC-V specifies that the least significant bits of a jump target PC must be cleared on a jump, since all instructions are 2-byte aligned with the C extension enabled, and 4-byte aligned otherwise. Naïvely implementing this with the existing capability compression functions would require a full representability check on the new address, which would be expensive. However, masking the least significant bits of an address can never make the capability unrepresentable provided sufficiently few bits are masked (three fewer than the mantissa width). This allows the address, and slice of the address extracted when decompressing to `CapReg`, to be directly masked without additional checks, which is much cheaper. This functionality was added to the compression library used by the RISC-V cores.

Further development was required for the Toooba core, and is discussed in Section 5.4.1. These changes were the applied to the Piccolo and Flute cores also.

3.4.2 Bounds check

The key to security with capabilities is that all memory accesses must be within bounds of a quoted capability. This includes both legacy RISC-V instructions (which implicitly dereference DDC) and any newly added instructions. To avoid a performance penalty, the bounds check should be performed in parallel with existing operations. However, the logic required is somewhat expensive even on the most decompressed capability type, requiring a variable shift to extract the base and top then subtractions to compare against the attempted access.

The required inputs to the bounds check are not ready until the end of the execute stage, so the check must be carried out in parallel with the first cycle of the memory access stage. Piccolo and Flute's interface with the cache enqueues any memory accesses at the end of execute. However, it is not acceptable for writes to go ahead should the bounds check fail: this would allow a process to corrupt data it should not have access to, even if it would go on to raise an exception. I therefore add an additional interface to the cache to

commit the access. Should the check succeed, this interface is triggered, allowing the cache to proceed with the operation as normal. However, on a failed bounds check, the cache cancels the operation without making any observable changes. This benefits performance as it allows the cache to begin lookups in parallel with the bounds check. The commit is expressed as a condition that must fire the cycle after the access is first enqueued to the cache, ensuring memory access can only ever occur following a successful bounds check.

Some other instructions also require a bounds check to detect monotonicity violations, for example `CSetBounds`, `CUnseal`, and `CBuildCap`. Jump instructions must also check bounds to ensure the target code capability is in-bounds. In general, instructions reuse the memory bounds check wherever possible: by design, no instruction in the CHERI ISA requires more than one full bounds check.

In addition, some instructions must compare bounds to determine their result, rather than just to trigger an exception, for example `CTestSubset`. Such instructions must either delay their result a cycle to allow the bounds check to be carried out the stage after execute before their output can be forwarded, or carry out a simplified check in execute. The check can be simplified in many cases by exploiting invariants guaranteed by the instruction itself. When changing to the semantics to tag clearing on error, more instructions move into this category, as discussed in Section 5.4.

Instruction fetch also requires some bounds checking, as although jump targets are checked at the caller, code could execute past the end of its last permitted instruction. This is made additionally complicated as RISC-V instructions with the C extension enabled are variable-length, meaning the width of an instruction access is not known when it is fetched. To address this, instruction fetch performs a check that the PCC has access to at least two bytes—the minimum instruction length—at its current address. Execute then performs a complete check to ensure the full instruction was in-bounds before it has any effect. This has the unfortunate consequence of allowing instruction reads to escape to the fabric despite not being covered by PCC bounds. To some extent this was already a problem since caches fetch on cacheline granularity. Aside from the possibility of side channels, this may allow the first few bytes of a side-effecting memory region, for example a FIFO, to be erroneously accessed if it neighbours instruction memory. The instruction fetch checks are cheaper than the full bounds check required for memory access, as they are with respect to a capability's own bounds and address, meaning the pre-shifted address and temporary fields can be used.

3.4.3 Additional instructions

CHERI requires additional instructions to be added to the processor. These include instructions to query fields from capability values, such as the base, length, and permissions. In addition, there are instructions to manipulate capabilities, changing their fields. These must ensure capability monotonicity and manipulate the *CHERI-Concentrate* encoding [135], making their implementation tricky and error-prone. The TestRIG testing framework (Section 3.6) helped to address errors as they arose. Additional load, store, and jump instructions must also be added to allow operands to be capabilities, rather than integers. This section assumes some detail of the added instructions: the purpose of each instruction is described in Appendix A, and detailed semantics can be found in the CHERI ISA document [128].

The CHERI MIPS ISA presented these additional instructions as a "capability co-processor", which operated in parallel to the ALU, as this was the conventional way of extending the ISA. For RISC-V, capability operations are instead considered to be part of the ALU itself, as this is how extensions are framed in the RISC-V ISA. The distinction is largely academic, as the synthesis tools will likely optimise the two together, although it is possible that the RISC-V approach will allow more logic to be shared as high-level invariants are exploited by the designer.

3.4.3.1 Capability inspection

The capability inspection instructions all query fields of capabilities, possibly after some manipulation. `CGetAddr`, `CGetFlags`, and `CGetTag` are the only instructions that can directly extract values from the encoded capability: the other instructions all have various additional complexities. `CGetBase`, `CGetLen`, and `CGetOffset` must partially decompress the capability to compute their result. Fortunately, prior work had allowed this to be done at high frequency for CHERI MIPS, but some changes were required for the new architecture and microarchitecture. Particular attention is also required for these instructions to handle queries at the top of the address space, especially when the capabilities are not tagged, so do not have legal encodings. `CGetPerm`, `CGetType`, and `CGetSealed` simply require some rearranging of the data into the required format. Treatment of sealed capabilities required some thought to avoid simple bugs potentially producing significant hardware vulnerabilities, to some extent adding intentionality to the hardware design itself. This is made particularly complicated by the different kinds of sealed capabilities, including sealed with an `otype`, sealed as a Sentry capability, unsealed, or a different reserved kind. As such, I implemented the capability type field as a `Kind` enumerated type that requires every use to consider all of the possibilities.

3.4.3.2 Capability modification

Several capability modification instructions require a bounds check to ensure they do not extend a capability or use it out of bounds. As described in Section 3.4.2, the bounds checks required for capability arithmetic are carried out the cycle after execute, sharing the hardware for the bounds check on memory accesses. Execute then simply marshals the control signals and relevant bounds to feed into the check. The processor already had the ability to raise an exception from this stage, since it also carries out memory accesses. Since the processor is in-order, the single cycle delay in observing the exception does not cause problems with side-effects in the pipeline. However, this was found to be a problem in application-class processors, described in Section 5.3.6.

`CAndPerm`, `CSetFlags`, and `CClearTag` have very direct implementations, simply manipulating the required fields after some checks.

`CSeal`, `CUnseal`, and `CCSeal` manipulate the type field of the result capability, after performing the checks required to determine that the operation is permissible. This requires a bounds check to ensure the capability authorising the type-space is in-bounds, combined with various more sealing-specific checks.

`CSetOffset`, `CIncOffset(Imm)`, `CSetBounds(Imm)`, `CSetBoundsExact`, and `CSetAddr` all perform arithmetic on the compressed encoding format. These use the common capability library discussed in Section 3.4.1, marshalling their arguments into a few common compression functional units that perform the required representability checks. These are a `setAddr` function, a combined `set/incOffset` function, and a `setBounds` function. `CSetBoundsExact` requires an additional check in series with the main operation to ensure that no rounding was required. While this was acceptable for Piccolo, it prevented Flute from achieving its target frequency, requiring optimisations discussed in Section 3.5.2.

`CBuildCap` requires rederiving a target capability from an authorising capability, for use in paging and dynamic loading. This is the instruction whose implementation differs most from its specification. The instruction is designed to allow the hardware to perform the required checks before setting the tag on the target capability. These checks include a bounds check, permissions checks, and a derivability check. The derivability check is of particular interest: it must ensure that the capability requested is one that could be derived from the almighty capability, forbidding any non-canonical bounds encodings. This can be done efficiently using a concise check that has been formally proven to be equivalent to derivability.

`CTestSubset` is the only instruction requiring a bounds check to determine its result rather than just whether to raise an exception. As such, it is implemented as a two-cycle instruction in Piccolo and Flute. The ALU sets signals such that the output of the bounds check will be used to determine the written back result, and the result is not forwarded until the end of `Stage2`. This does not incur a performance penalty unless the following instruction reads the result.

3.4.4 Cache modifications

Beyond the pipeline itself, the data caches also need to be modified. In particular, space is required for the capability validity tags alongside the corresponding data. Furthermore, the granularity of the caches must be increased to cover at least one capability, since loads and stores are added for capability data.

The data cache in the Flute design is responsible for implementing atomic operations. This implies some additional complexity as capability atomic operations must be implemented. The added operations include atomic swaps of entire capabilities. Atomic operations down to byte-granularity are also required, since capability bounds may restrict code from using a wider access.

The caches also require modification to abort accesses on a failed bounds check, as discussed in Section 3.4.2.

3.4.5 Memory subsystem changes

The caches need to communicate their capability validity tags to main memory. This is facilitated by the tag controller component developed for CHERI MIPS, which splits accesses into requests for tags and requests for data, caching tags to minimise additional accesses [63, 64]. The integrity tags need to be communicated through the AXI fabric between the components. AXI offers `ruser` and `wuser` fields to allow user-specified out-of-band metadata to be communicated between components. We use these fields to convey capability integrity tags everywhere beyond the caches. This minimises the chance of confusion between data and tags. However, AXI does not define what the behaviour of these fields should be, including at interconnects. This means care must be taken when using stock interconnect components: although forwarding the fields is typically the default, some components expect the fields to be zero-width. We imagine that AXI would add dedicated bits for capability tags in CHERI systems. Work was required to augment the tag controller to support an AXI interface on the core side, including support for burst transactions.

Until the DMA devices are made CHERI-aware, care must be taken that DMA devices cannot corrupt capabilities, including writing to the tag table, or rewriting capabilities without going through the tag controller, so without clearing the tag. This is achieved by only mapping uncached memory for DMA AXI initiators. All uncached transactions are then assumed not to contain tags (or rather, that all the tags are not set). Future work is required to determine how CHERI can safely interact with DMA in a performant way.

3.4.6 Other changes

Additional CSR state is required, as some CSRs must be extended with capability metadata to form SCRs. For example, RISC-V's mechanism for specifying the exception vector—`mtvec`—must be extended to be a capability so that the processor can run the exception handler without risking overrunning its bounds. Once again, capability compression complicates this process, as an offset is added to the capability pointer, which may be sealed, or operations may take it out of bounds or representable bounds. While these cases should never occur for correct code, their behaviour must be architecturally specified and enforced to ensure capability monotonicity guarantees cannot be violated when they are intentionally provoked, undermining compartmentalisation. Extra exception conditions and exception metadata also add design complexity.

Each processor's memory management unit (where present) needs alterations due to the additional CHERI page-table permissions. This includes bits to prevent loading or storing tagged capabilities from the page, as well as a bit indicating the page is capability-dirty to accelerate revocation (this is discussed more in Section 7.2.1).

3.5 Flute

Following the initial implementation in Piccolo, Flute was an extension to a deeper pipeline. The chief new challenges were branch prediction and timing. While Piccolo avoids needing prediction on branches by having fetch, decode, and execute in a single stage, Flute introduces branch prediction. This poses some interesting questions for CHERI, as predicted instruction addresses could now be extended with bounds information. In addition, compared to Piccolo's modest 50 MHz target, Flute's additional pipeline stages allowed it to be clocked at 100 MHz. Therefore, CHERI extensions were more likely to cause the design to violate timing. However, investigation showed these delays were not inherent to the capability model, but could be mitigated with optimisation work.

3.5.1 Branch prediction

Branch prediction poses challenges in capability processors, as the architectural bounds against which instruction accesses should be checked are not known. This leaves open many possibilities, including adding all the PCC-metadata to speculated state or bypassing bounds checks in speculation entirely, cleaning up in the event a misprediction is detected. In practice, the latter approach is not viable, both because of side-effecting reads (such as dequeuing a FIFO), and because of speculative execution vulnerabilities [71]. The former approach mitigates this to some degree, but still allows the capability model to be bypassed in the face of malicious branch predictor training. In addition, it could incur

a significant performance penalty as mispredictions will be detected (in PCC bounds or other metadata) where execution would have proceeded the same regardless.

I therefore choose a compromise solution, where the front of the pipeline (`StageF` and `StageD`) works with addresses, assuming the current PCC will be used to fetch them. In `Stage1`, the correct PCC of the next instruction is computed, and compared with the next instruction address that was predicted. This is possible because the PCC metadata is not required until `Stage1`, so it never needs to be speculated. The exception to this is the `flags` field, which is required to perform decompression in `StageD` (see Section 3.3.4). This additional bit therefore does need to be added to the predicted state.

3.5.2 Timing

The baseline processor was synthesised for a VCU-118 FPGA at 100 MHz. Initial CHERI modifications focused on correctness, with little emphasis on performance. This led to an initial $F_{\max}$ of 66 MHz, which was lower than expected from the CHERI MIPS work. I therefore performed some optimisation to improve timing. To help me do this, I synthesised the design for a DE4 Stratix IV Intel FPGA board, as I found the path length information provided by Quartus much more useful for optimising the critical path than the Xilinx tools. Figure 3.5 shows anecdotally how the timing improved with each of the changes discussed in this section.

The improvements were mostly either CHERI extension improvements or baseline core improvements. In some cases, the way I had chosen to add the CHERI extensions caused needlessly long paths. Some aspects of the baseline Flute processor were not optimised for timing, since they were not on the critical path for the unmodified processor. In some cases, adding the CHERI extensions put these aspects onto the critical path, meaning improvements to them were necessary for a performant design.

Forwarding fixes Initially, `Stage2` would perform its bounds check and only proceed with normal operation if the bounds check succeeded. However, this meant the value forwarded back to `Stage1` depended on the slow bounds check before it could be fed into the ALU. Removing this dependency took this off the critical path, and was safe to do as the forwarded value would not be used if the bounds check failed, as this is an exception case.

Cache-bus latching As discussed in Section 3.4.5, Piccolo and Flute were changed to use a different AXI library, containing different interconnect implementations to convey tags. This initially caused timing problems, as paths within the memory bus were limiting the maximum frequency. Adding additional registers broke these paths, taking this off the critical path, at the expense of additional cycles to service cache misses.

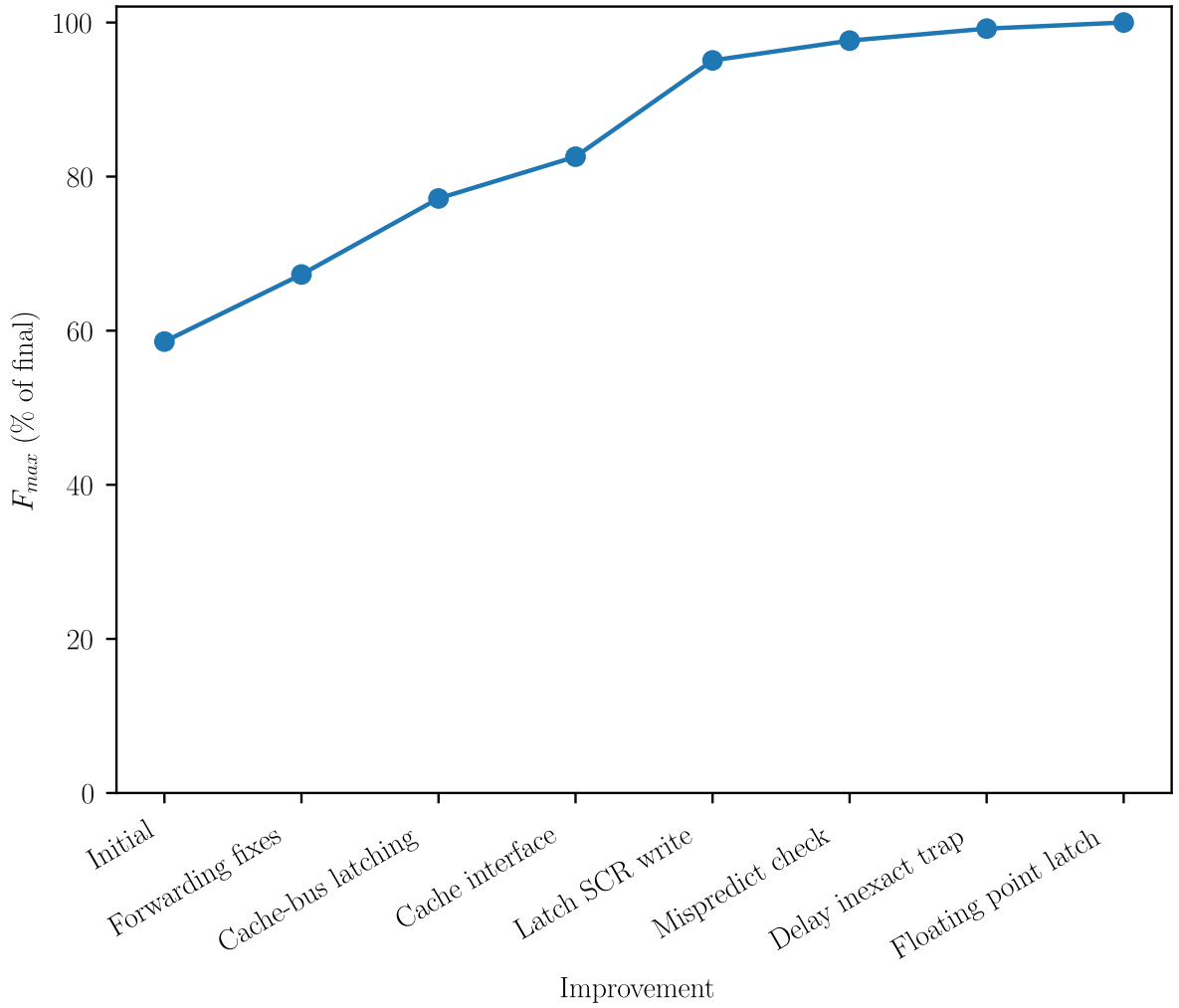

Figure 3.5: Changes in timing of the Flute core as improvements were made. Frequencies are those reported by Quartus synthesising for a DE4 FPGA normalised against the final achieved frequency, which exceeded the 100 MHz target for VCU-118.

Cache interface The cache interface caused timing issues that were unnecessary given higher level invariants. The interface was specified as an action guarded by whether the cache was ready. However, by construction, the pipeline would never issue a request to the cache unless the cache was ready: in any other case, the pipeline would be waiting for the cache to resolve itself into the ready state by servicing the pending request. This is a consequence of Piccolo and Flute being in-order: the pipelines halt under a miss. Exposing this invariant to the Bluespec compiler required some refactoring of the cache interface. The baseline processors wrote to the cache registers in the `req` method, causing logic to be generated to push back on the pipeline should the caches be busy. I changed the interface to instead write to Bluespec's `Wires`. This frees the compiler to discard the request if not ready, which is known not to happen. Additional optimisation of the commit interface discussed in Section 3.4.2 also improved timing. Rather than guarding certain cache behaviours on the result of the bounds check, the result was propagated all the way

to the valid bits of the AXI write transactions to main memory and the write enable of the BRAMs containing the cachelines themselves. This allows the evaluation of the bounds check to be carried out completely in parallel with the processing in the caches, up until the very end of the cycle when the final values are required.

Latch SCR write CHERI RISC-V defines SCRs: effectively RISC-V CSRs that hold full capability-width values. Latching the computed values before updating them in exception cases allowed timing to improve further. Since writes to SCRs are relatively uncommon, the state machine that handles flushing the pipeline when they are modified can be extended to shorten the critical path without noticeably impacting performance.

Mispredict check The next critical path was the comparison between the predicted PC and the architecturally correct PC, which now depends on an add with the base of PCC. Since detecting a mispredict is required at the end of execute in the baseline, the comparison is serialised at the end of the next PC computation in the ALU, explaining why this path is problematic. Delaying the mispredict check would incur additional complexity, as the pipeline would need to tolerate one more level of misspeculated instruction. This would also worsen performance, as the mispredict penalty would be increased by a cycle. To experiment with precomputing different functions of the PCC, I replaced all references to the PC with references to a custom typeclass. An approach that passes timing is to compute the full PCC base eagerly whenever PCC is updated. This allows the predicted next PCC offset to be computed in parallel with the architectural next PCC offset in the ALU. Since the offset is the value observed by the legacy RISC-V instructions, this is the value ready earliest out of the ALU, so this approach allowed the most direct comparison to detect a mispredict.

Delay inexact exception The `CSetBoundsExact` instruction was then on the critical path, as it required the output of complex rounding logic to determine whether an exception should be triggered. This could safely be avoided by delaying the exception by one cycle: the exactness failure was registered and then fed into the bounds checking logic to trigger an exception from `Stage2`. This improves frequency without affecting IPC unless an exception is raised.

Floating point latch The final fixed critical path lay within the floating point execution unit. Rebalancing the input and output latches allowed this critical path to be broken without impacting performance.

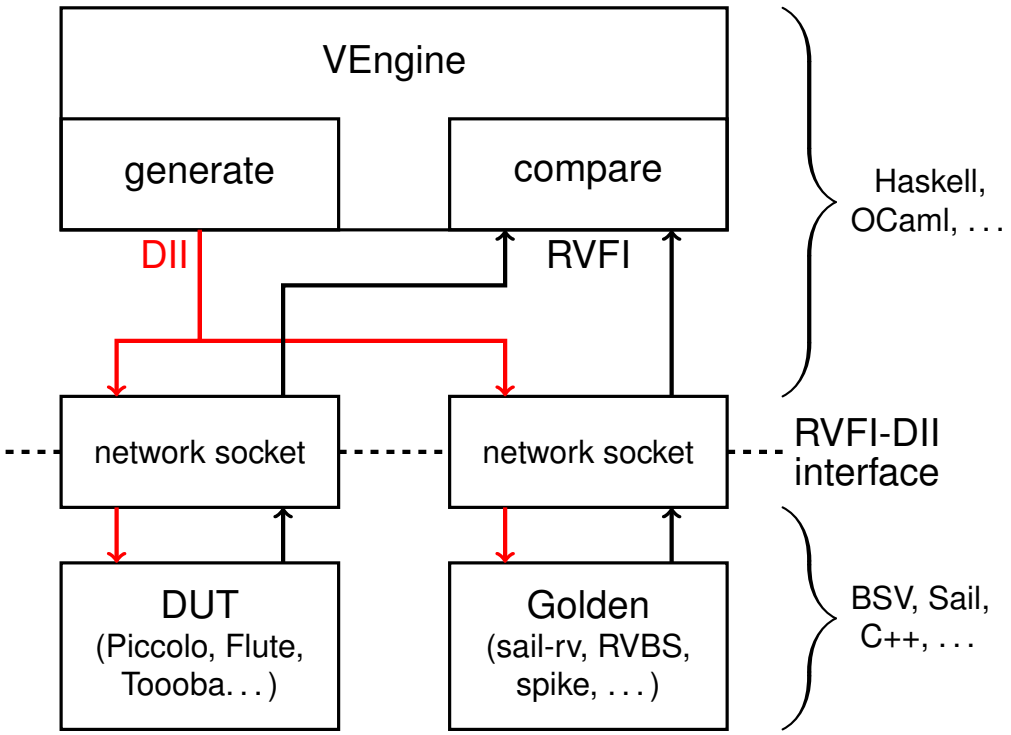

Figure 3.6: The RVFI-DII interface as it connects a Vengine to implementations.

3.6 TestRIG

This section discusses TestRIG (Random Instruction Generation): an automated testing framework for RISC-V processors. We developed the TestRIG framework to aid with rapidly bringing up CHERI implementations incrementally with some assurance the implementation of each instruction was correct. The framework describes the general design of test-case generators (Vengines) and interfaces to implementations (RVFI-DII). To enable comparison of CHERI RISC-V processors, we developed QuickCheck Vengine—a Vengine written in Haskell using the QuickCheck library [28]—to perform CHERI-aware verification. Testing can then be carried out in a model-based way, with an implementation being compared side-by-side against an executable model: in our case the Sail CHERI-RISC-V model.

Due to the inflexibility of hardware systems once manufactured, the economic consequences of hardware bugs can be very high [102]. This has lead to hardware design companies devoting a large proportion of their resources to verification. In the case of CHERI, and other capability systems, the design is even more sensitive than usual to security bugs. While a common fix for a hardware vulnerability is to offer a means of working around it in software (possibly with a performance penalty), this would not be an option for many types of CHERI bug, due to the security model it offers. For example, if operating with mutually distrusting processes in the same system, the processes are entirely reliant on the hardware to enforce isolation. Violations in capability monotonicity properties therefore cannot be mitigated in software, as malicious code could choose not to use the updated

software. This motivates increased effort for verification: in particular directed random testing that can find edge-cases that would be difficult to trigger with manual testing.

Traditional testing consists of running a test suite, which runs prescribed code and checks at predetermined points whether certain execution results match hard-coded values, for example the RISC-V test suite [21]. While this sort of testing is essential to ensure processor behaviour matches expectations, it cannot be expected to cover all potential errors in the processor thoroughly. Fuzzing has been shown to be effective in finding errors in other applications [123].

TestRIG uses the RVFI-DII protocol to both insert instructions into the CPU, bypassing instruction fetch (DII, developed by the CHERI group) and report the results of executing the instructions using the existing RISC-V Formal Interface (RVFI) interface [134]. Reporting per-instruction traces with RVFI allows fine-grained comparison of behaviour between the Design Under Test (DUT) and model. RVFI has already seen use for verification [134]. We use DII to inject instructions to allow for easier random instruction generation, and counterexample shrinking.

One of the key usability features of QuickCheck Vengine is counterexample shrinking. Since tests are randomly generated, counterexample sequences can be very long, containing many instructions that are irrelevant to the errant behaviour. By augmenting QuickCheck's shrinking functionality, we are able to remove irrelevant instructions automatically, saving effort to diagnose failures. An example of this process is given in Appendix C.

Automatically generating tests without directly injecting instructions, especially to test behaviour of jumps, is made difficult since the input must be generated to ensure the processor stays within the intended code. This requires highly constrained generation of jumps. It also restricts counterexample shrinking, which could alter control flow. The DII approach bypasses this problem by allowing the processor to ignore the PC, and just execute a predefined sequence of instructions sequentially. The PC the processor believes it is executing is still reported, so the control flow is still tested. Only the front-end of the processor, most notably the instruction cache, goes untested. This approach allows tests to be generated much more freely, as modelling the PC is not required to generate effective tests.

3.6.1 QuickCheck Vengine

We have implemented a Vengine based on Haskell's QuickCheck [28] library. The Vengine includes descriptions of the encodings of RISC-V and CHERI RISC-V instructions. It also includes a definition of equality between RVFI reports, allowing traces to be compared in an intelligent way. Finally, it includes libraries of useful templates, including useful mixes of instructions and templates for getting the processor into particular states.

I added various functionality to QuickCheck Vengine, including smarter instruction shrinking and recursive templates.

3.6.1.1 Smarter instruction shrinking

The existing instruction shrinking just removed instructions from a failing stream and checked if the stream still gave a counterexample. This could lead to shrunk traces still being complicated, as they may contain instructions that do not contribute to the bug, but just serve to pass their operands (slightly modified) into their destination register. I added functionality to specify simpler versions of each instruction. I then also developed a library of possible shrinks, enumerating cases where likely bugs in instructions can be observed in a simpler way. For example, a `CAndPerm` instruction copies its source capability to the destination register, `AND`ing the permissions field with a given bit vector. In cases where the bug does not relate to permissions, this instruction can be replaced with a simple `CMove` giving the same behaviour. I then also added the ability to remove these simple "bypass" instructions, and replace the subsequent instances of their destination registers with their source registers until updated later.

3.6.1.2 Recursive templates

The existing generator only allowed a single distribution over instructions to be specified. This made it difficult for the Vengine to find counterexamples requiring a complex sequence of instructions to reveal the errant behaviour. For example, detecting an endianness issue with memory accesses requires loading some immediate to a register, writing that register to a memory address, writing the same memory address with a different width, then reading back the value. Recursive templates allow arbitrary combinations of instructions to be specified to test the processor more deeply. For example, I added a template that surrounds an existing template with a store to a memory location (including the arithmetic to generate the address) and a load from that same location. I then produced a set of these templates that probe behaviour of the CHERI instructions, guided by areas of the Sail specification lacking coverage and complicated cases. For example, a template to produce a short capability that is then consumed by other templates identifies common off-by-one mistakes in interpreting and manipulating the upper bound of a capability. More information is included in Appendix 3.6.

3.6.2 Implementing RVFI-DII

Augmenting a processor with RVFI-DII requires that it offer RVFI (prescribed tracing format) and DII (direct instruction injection).

RVFI is a format designed by Wolf to facilitate formal equivalence proofs between processors [134]. The format describes the state changes undertaken by a processor when an instruction is retired, including: the PC of the next instruction to be run; the index of the written back register and the written data; the memory address written if the instruction is a store, as well as the data and strobe; and whether the instruction triggered an exception. Augmenting Piccolo with RVFI reporting had already been done when I joined the project.

The implementation of DII, which I carried out, causes more significant difficulty, since it involves interposing between the usual instruction fetch mechanism and the pipeline. This particularly causes problems when speculative execution of instructions would cause the instruction traces between two processors to go out of synchronisation as one processor cancels an instruction in the trace while the other does not.

Even though Piccolo is a scalar core with no branch prediction, it still carries out some speculative execution. In particular, since it may not be known until the end of the second stage whether an instruction has triggered an exception, all instructions are technically executing speculatively until their predecessor exits the second stage.

I solved this problem by introducing a *replay buffer*: a FIFO of recent instructions injected into the processor. The replay buffer models what instructions are present in the pipeline: it is enqueued with the instruction injected into decode, and dequeued as the instruction commits from writeback. Each instruction is tagged with a sequence number, indicating the instruction's position in the DII stream. When the processor requests an instruction, it must provide the expected sequence number as well as the PC, allowing the correct instruction from the replay buffer to be returned. The PC cannot be used for this purpose as the processor could be executing a loop or repeatedly triggering an exception, but DII allows the same instruction address to contain different instructions on different cycles. The sequence number bit-width and replay buffer depth are parametrisable, allowing this approach to generalise to any size of pipeline. This approach generalises to solve the problem for Flute and Toooba, which have much more speculative execution. The only change required in the pipeline is that it must keep track of the sequence numbers alongside their instruction metadata. This is fairly straightforward for Flute and Toooba, as all such requests are carried out in dedicated rules depending on where the mispredict happened, so each call-site can be annotated with the expected sequence ID.

3.6.3 Testing with TestRIG

While bringing up the CHERI processors, I used TestRIG with QuickCheck Vengine substantially to perform debugging of features as they were added. The effectiveness of a testing framework can broadly be split into two categories: productivity impact of running the tests and thoroughness in finding (significant) bugs.

In terms of productivity, the existing implementation of CHERI RISC-V in Sail provided a model to test against. The tests could be run as instructions were added, increasing the test frequency of the most recent instruction to catch simple errors. The Sail model had yet to be thoroughly tested or used to run significant software itself. Therefore, this process also allowed refinement of the Sail model as some divergences with the hardware implementation were revealed to be specification bugs. There are no false-positive failures, as any divergence is either a bug in the model or the implementation. However, model-based testing is very pedantic in requiring no unspecified behaviour in the specification, posing problems for fast representability checking[4] and time-sensitive events (such as interrupt handling).

In terms of discovered errors, as well as many trivial bugs in the bringing up of Piccolo, TestRIG found several deep bugs in relatively mature projects:

- A critical security flaw in the CHERI MIPS compression implementation, whereby a three-instruction sequence could turn a zero-length capability into a capability covering the entire address space. This was caused by incorrect treatment of the top bits of the bounds overflowing. The bug also revealed an error in the proof script for checking the correctness of the CHERI MIPS compression code.
- Several compression bugs resulting in overly conservative representability-check failures.
- A bug in Sail RISC-V's determination of which memory region contained a given address.

Appendix C includes more evaluation of TestRIG and QuickCheck Vengine, including more detail on some discovered bugs.

Furthermore, shrunk counterexamples provide a good candidate to save and quickly run on the processors for regression testing. We have gathered such counterexamples into a library. Traces captured from booting operating systems were also used to accelerate the early stages of operating system bring-up, with counterexample shrinking again rapidly highlighting areas of divergence.

[4]To simplify hardware implementations, the CHERI architecture allows some operations to give an error (by clearing the tag of the result) when they come close to producing an unrepresentable result (see Section 2.4.2.1), even if the result would actually be representable.

3.6.4 Other verification

We continuously tested both processors against the RISC-V test suite [21]. The high parametrisability of the cores to support different architectural extensions, and different microarchitectural decisions (such as whether to have a serial or barrel shift unit) caused significant additional design and verification effort. To avoid the full combinatorial explosion of combinations, in continuous integration we simply run the base architecture, the architecture with each feature enabled individually, then the architecture with all features enabled.

Both processors have additionally been validated by running respective appropriate CHERI-extended OSs: in Piccolo's case Cheri-FreeRTOS [24] and CheriRTOS [139], and in Flute's case, additionally, CheriBSD [23]. CheriBSD also includes the `cheribsdtest` test suite, which runs successfully.

As part of the ECATS project, we tested the processors further using the Galois-provided BESSPIN test-suite.

Furthermore, an external project by Gao and Melham attempted to formally verify the CHERI Flute processor against the Sail specification [51]. I gave significant support to diagnose and fix any differences discovered between the implementations. In particular, this required Flute and Sail to agree on the treatment of "underivable capabilities": capabilities that cannot be produced by legal capability manipulations so can never exist with a tag, but can nonetheless exist in registers as untagged values, for example when loaded in from memory. Since capability inspection instructions can be executed on such capabilities, the length and base field decoding of such capabilities must be specified. I decided to do this by clearing the "internal exponent" bit whenever the internal exponent would otherwise imply a capability granting permission outside the address space (it turns out this is the only underivable case). This verification work provided further reassurance that the processor correctly implements the specification, at least on an architectural level.

3.7 Future work

This work has focused on producing correct CHERI implementations, validated by TestRIG against the Sail specification and running of CHERI software including operating systems and benchmarking suites. This has enabled co-design of software, hardware and the architecture, helping to settle some of the architectural decisions discussed in this chapter. In addition, microarchitectural amelioration has been carried out to bring the performance and area to an initially acceptable level. However, it is likely that more detailed analysis and optimisation of the microcontroller cores would allow the CHERI overheads to be further reduced. This remains as future work.

Security improvements offered by CHERI include spatial safety, temporal safety and compartmentalisation [128]. This chapter has mostly focused on the spatial safety aspects of security improvement. Further investigation is required, particularly around temporal safety and revocation, for instance implementing the `CLoadTags` instruction: Chapter 7 discusses this in the context of an application-class core. In addition, fast compartmentalisation support is yet to be implemented, such as the implementation of the `CClearRegs` instruction to enable fast domain transition.

3.8 Summary

This chapter gives an initial qualitative answer to Hypothesis H.1: CHERI is an option for increasing microcontroller security. We have seen that RISC-V microcontrollers can be augmented with CHERI to run pure capability code, including RTOS and (in Flute's case) application-class capability-enhanced OS support. In particular, the added CHERI RISC-V ISA features are validated for microarchitectural feasibility, including the merged register file, capability encoding mode, Sentry capabilities, and CHERI-optimised compressed instructions. *CHERI-Concentrate* compression incurs challenges in the pipeline that require careful consideration to resolve. The critical path was not lengthened in the case of the processors investigated, albeit with some effort to restore clock frequency in Flute's case. The quantitative aspects of Hypothesis H.1 are investigated in Chapter 4.

While the Piccolo and Flute cores provide assurance that CHERI can be applied to microcontrollers, a gap remains in implementing the extensions for more optimised and commercially-used cores. Work has been carried out within the CHERI team to add the CHERI features to the more tightly area-optimised lowRISC Ibex core [22]. In addition, Microsoft have announced their successful implementation of the CHERI extensions, also to Ibex [106]. Arm have also announced their investigation of CHERI for their M-class microcontrollers [105]. However, little has been announced regarding expected area and performance overheads for these designs.

Chapter 4

CHERI microcontroller evaluation

This chapter details the evaluation performed to investigate the power, performance, area and security of the CHERI-augmented microcontrollers discussed in Chapter 3. The evaluation is carried out on a VCU-118 FPGA board, using the BESSPIN SoC design. Figure 4.1 gives the configurations of the processors used for benchmarking. The caveats associated with FPGA evaluation as discussed in Section 2.1 apply. The baseline processor used is the version based on which the CHERI changes were added, with some non-CHERI-specific changes backported.

The aim of the evaluation is to investigate the effectiveness of the microarchitectures developed. This depends on CHERI software development—most notably the toolchain and OS—to compile and run CHERI code (see Section 2.4.3). This software is not a contribution to this thesis, and is the result of development by others in the CHERI team.

The evaluation of CHERI overheads addresses the quantitative component of Hypothesis H.1, giving an initial measure of the costs that might be expected when applying CHERI to microcontrollers. Since the Piccolo and Flute microcontrollers have both been evaluated, the differences between the overheads also serves as an initial investigation of how the overheads scale with the size of baseline core (Hypothesis H.3).

4.1 Baseline core information

To provide context for the overheads found in the chapter, I first summarise key metrics for the baseline processors in Figure 4.2 for the SoC used for evaluation. While the load-to-use delay is realistic for such simple pipelines, the access latency for DRAM is significantly lower than would be expected for DRAM on an ASIC, as discussed in Section 2.1. However, the latency is also much higher than might be expected for a microcontroller connected to a Tightly-Coupled Memory (TCM).

Possibilities for comparison with other cores are limited, as they will depend on the SoC and evaluation context. However, the CoreMarks/MHz scores for these cores are at the

	Piccolo	Flute
Frequency	50 MHz	100 MHz
`xlen`	32	64
Supported extensions	C, M	A, C, D, F, M
Supported privilege modes	M, U	M, S, U
TLB	—	12 entries direct-mapped
Pipeline stages	3	5
RAS	—	16 entries
BTB	—	512 entries direct-mapped
Data cache	4 KiB write-through L1	8 KiB write-through L1
Instruction cache	4 KiB L1	8 KiB L1
Tag cache (CHERI only)	4 KiB	4 KiB

Figure 4.1: Benchmarking configuration for the Piccolo and Flute processors. All caches are two-way set associative, except the tag-cache, which is four-way set associative.

	Piccolo	Flute
CoreMarks/MHz	4.6	3.8
L1 hit load-to-use cycles	1	1
DRAM latency cycles	56	66

Figure 4.2: Metrics for the baseline Piccolo and Flute cores in the evaluation SoC. The cache latency was determined in simulation. DRAM latency was determined based on performance counters in the `adpcm_decode` MiBench benchmark. TLB accesses are performed in parallel on hit, causing no additional delay.

higher end of those reported by Elsadek and Tawfik [43], including lowRISC's Ibex (2.44) and Andes' N22 (3.97). The high performance likely comes at the expense of significantly higher power and area, and possibly slower clock frequencies. For example, Ibex uses 2,500 6-input LUTs on a Xilinx 7-series FPGA at 50 MHz [125]. Piccolo as used in this chapter is significantly larger at 20 kLUTs. As can be seen in Figure 4.3, this is primarily due to the inclusion of caches and CSRs capable of supporting an OS and performance counting. Full comparisons would require setting up and evaluating the cores in comparable contexts.

4.2 Area

This section investigates the area of Piccolo and Flute before and after adding CHERI. My particular implementation of CHERI leans towards better performance at the expense of area: structures are fully extended to allow the common case to be carried out without additional cycles of delay. As such, most of the overhead of the extensions is expected to be seen as area overhead.

Syntheses of the FPGA designs are performed using Vivado 2019.1 for the VCU-118. Vivado is configured to use the default optimisation strategy for Piccolo, but the "`Performance_ExplorePostRoutePhysOpt`" strategy for Flute, which makes more attempts to meet Flute's more demanding clock frequency. The strategies used are the same between the baseline and CHERI versions. Determining the exact source of overhead is made difficult by the various layers of optimisation performed between specification of the hardware in Bluespec, and the final synthesised design on the FPGA. Even compiling the design from Bluespec to Verilog loses some of the structure. The Vivado tool also significantly rearranges the boundaries between modules during its optimisation, so the attribution of area to modules is only approximate. To minimise this effect, Vivado "`keep_hierarchy`" directives are used on all module boundaries to restrict optimisations between modules, showing the overheads for individual components more accurately. This does change the overall area numbers, so I also report the fully-optimised area numbers and overhead separately. While the synthesis results depend on non-deterministic placement and routing algorithms, Vivado does not provide an easy way to vary the seed used. This prevents uncertainties from being indicated in the area, power, and timing metrics produced by the tools.

The VCU-118 board used contains a XCVU9P-L2FLGA2104E FPGA [140]. The two primary metrics used to measure area overhead are LUTs and Flip-flops (FFs): the VCU-118 offers 1,182 kLUTs and 2,364 kFFs. LUTs provide combinatorial logic operations: each can take up to six inputs. Bitstreams program each LUT with a bit for each of the 64 possible input combinations, indicating the corresponding output. This means that a LUT can potentially act as up to six traditional logic gates chained together. LUTs therefore give an indication of the area of the processor used for combinational logic. FFs provide storage of individual bits, giving an indication of the area of the processor used for preserving state between cycles. BRAMs allow much more area-efficient storage on FPGAs at the expense of a cycle of delay on access: the VCU-118 offers up to 345.9 Mb of such storage. In the key cases where these are present, I manually calculate the number of bits stored to indicate the overheads. LUT overheads for CHERI are generally higher than FF overheads, so I mostly focus on LUT overheads in analysis.

The LUT and FF overheads for various components of Piccolo and Flute are given in Figure 4.3 and Figure 4.4 respectively.

The overheads for each component are broken down as follows:

Processor On the processor level, Piccolo has a 62% and Flute a 49% LUT overhead, with smaller FF overheads of 15% and 24% respectively. This includes everything written in Bluespec: the pipelines themselves, caches, and components like the debug module and Platform-Level Interrupt Controller (PLIC), but not peripherals, DMA, and the corresponding interconnect of the BESSPIN framework. Note that while caches are included, their main area contribution is hidden by BRAMs on FPGA.

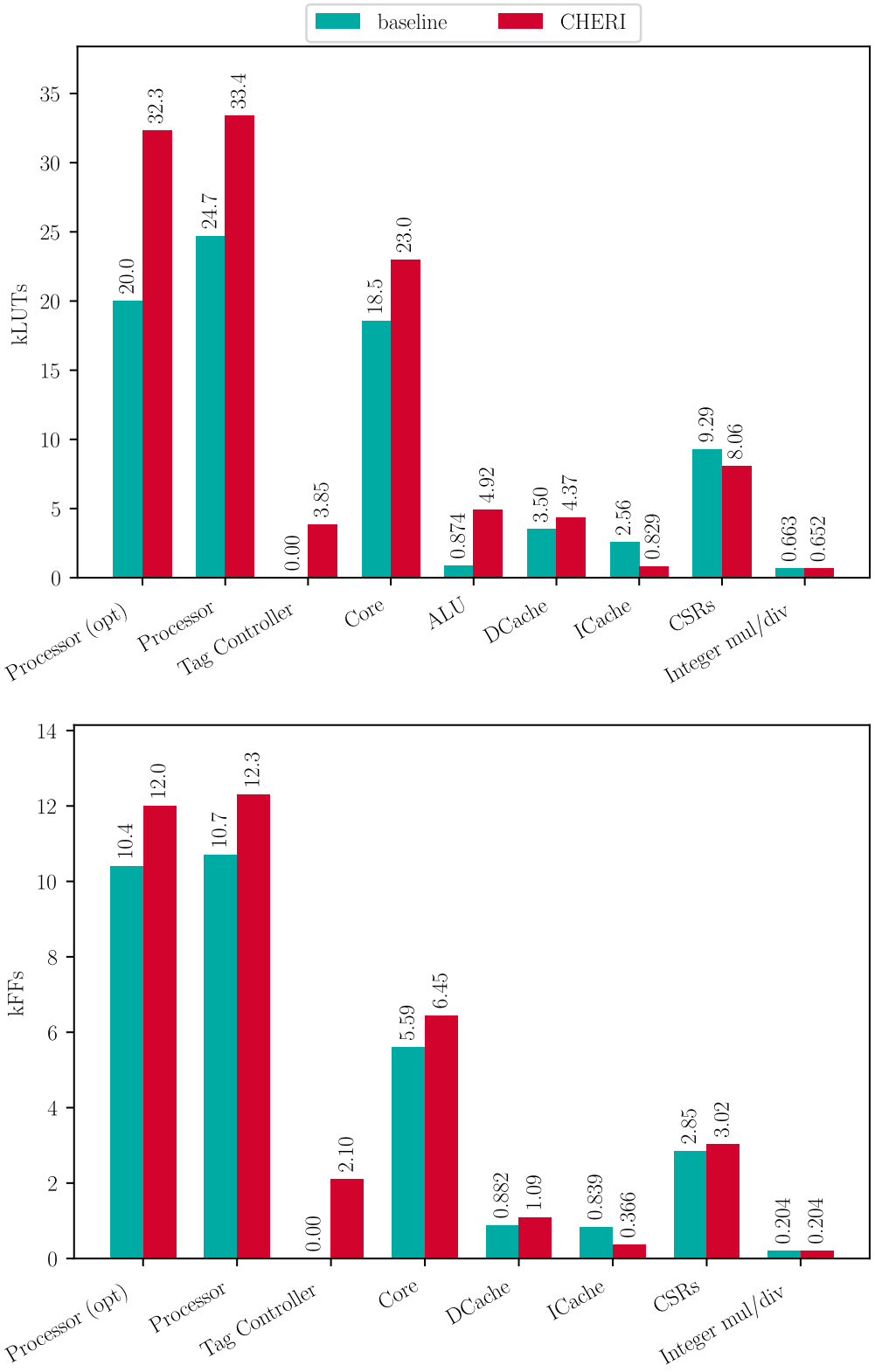

Figure 4.3: Area overhead of CHERI for the Piccolo processor. The tag controller and core are included within the processor, and all other components are in turn contained within the core. "Processor (opt)" shows the fully optimised area, i.e. without boundaries between modules.

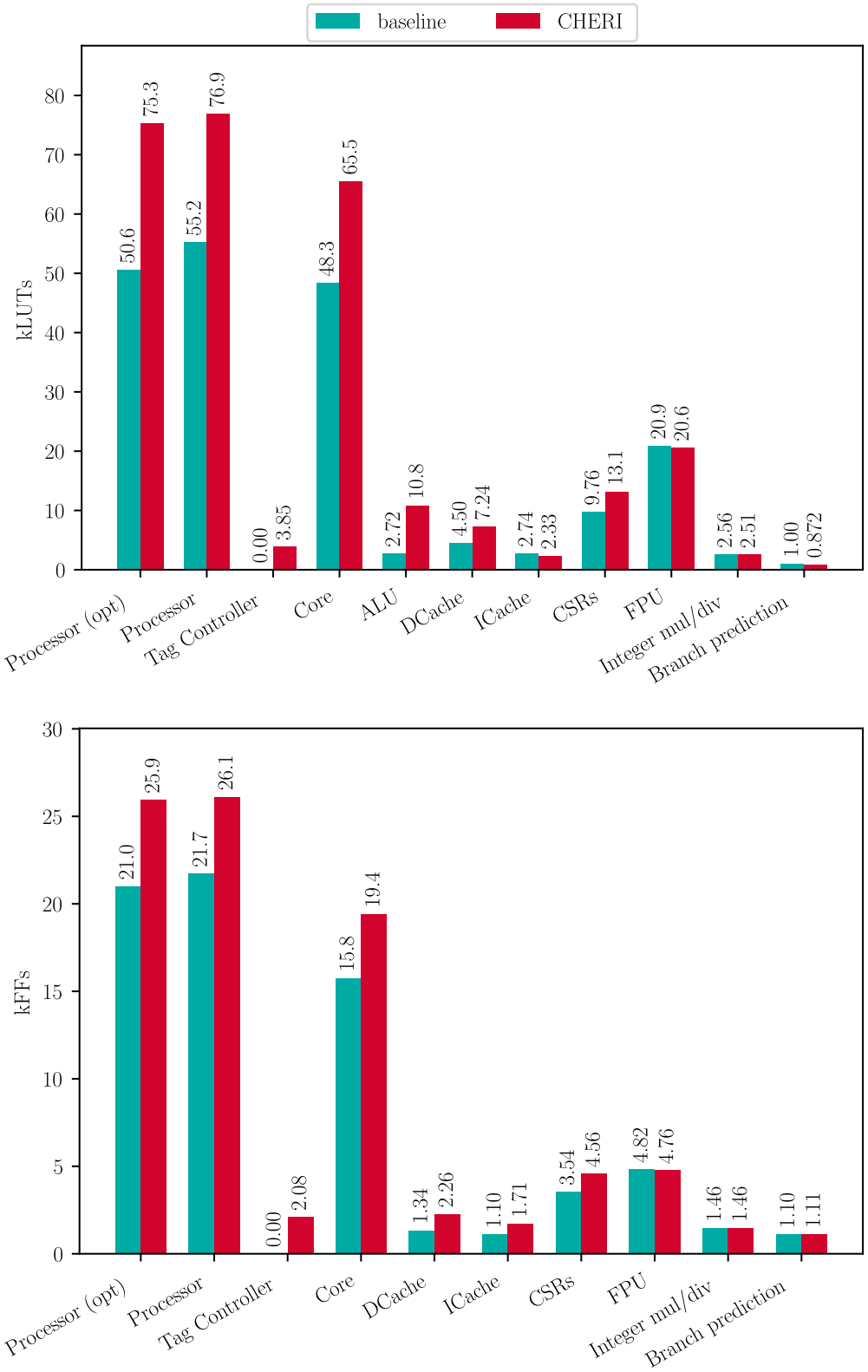

Figure 4.4: Area overhead of CHERI for the Flute processor. The tag controller and core are included within the processor, and all other components are in turn contained within the core. "Processor (opt)" shows the fully optimised area, i.e. without boundaries between modules.

Structure	Piccolo		Flute	
	Baseline	CHERI	Baseline	CHERI
Integer register file	1,024	2,592	2,048	4,832
FP register file	—	—	2,048	2,048
Data cache	35,840	36,352	72,320	72,832
Instruction cache	35,840	35,840	72,320	72,320
Tag cache	—	38,400	—	38,400

Figure 4.5: Number of stored bits in Piccolo and Flute structures that are hidden by BRAMs on FPGA. Numbers for caches include data, capability metadata and tags, and cache lookup tags. CHERI adds bits for capability metadata and tags to the general purpose register file, capability tags to the data caches, and an entirely new tag cache. Note also that caches and registers are synthesised very differently on ASIC due to different access characteristics, so these numbers are not directly comparable.

This significantly inflates the area overhead: an ASIC design would be expected to show much smaller fractional overhead as the caches (where capabilities have a small impact) increase the size of both the baseline and CHERI designs. For context, CHERI MIPS reported a 32% FPGA logic overhead to a larger (while still in-order) baseline processor [136]. Almatary reported a 7,588 LUT overhead when instantiating a 16-entry PMP in Flute (configured to be 32-bit) in a very similar context [5], corresponding to a 38% relative overhead for Piccolo at this level.

Tag controller The tag controller contributes 3,849 LUTs of design area for Piccolo and similarly 3,850 LUTs for Flute. This is significant, especially for Piccolo where it alone contributes 44% of the CHERI processor overhead (18% for Flute). It is not obvious how to account for the tag controller fairly. On one hand, the tag controller would be instantiated once per SoC rather than once per core. This is both to reduce area and to ensure coherence of tags, which is essential to prevent capabilities being forged. This would imply that the tag controller area should not be included in the processor area overhead in order to give a more meaningful estimate of the impact of adding CHERI. However, this is not relevant for SoCs that contain only a single core. As discussed in Section 3.1, microcontrollers may also be connected to a TCM rather than DRAM [12]. In this case, where commodity DRAM does not need to be targeted, the tag controller can be omitted from the design in favour of additional tag bits inline with the data, at a much smaller logic cost. However, this does have implications for efficient temporal safety, as the tags are not available independently from the data (see Section 7.3.3). The tag controller could also see further area optimisation: for instance, its cache is currently four-way set associative, which may be more than required in microcontroller contexts.

Core The core area numbers exclude the tag controller, and other components that would

only occur once per SoC: the debug module and PLIC. The Core-Local Interrupt Controller (CLINT) is also excluded: this would be reproduced per core, but its area contribution is relatively small. These other components have minimal changes for CHERI. On this granularity, Piccolo sees a 24% and Flute a 36% LUT overhead.

ALU Within the core, the ALU LUT overheads are the most striking: 463% for Piccolo and 298% for Flute. The absolute overhead is greater for Flute than for Piccolo due to the wider capability-width: 128 bits rather than 64. This alone accounts for 47% and 37% of the total CHERI area overheads for Piccolo and Flute respectively. A high overhead here is to be expected due to the complex operations, including shifts, required to manipulate the *CHERI-Concentrate* encoding [135].

Register file Since Piccolo and Flute register files are synthesised as BRAMs on FPGA, they comprise negligible LUTs and FFs, so are not shown on the graph. The CHERI impact on the size of the register files can be seen in Figure 4.5. By assuming each bit implies one FF[1], the CHERI FF overhead that might be expected on ASIC can be estimated. For Piccolo, this comes out as an overall FF overhead of 28%, and for Flute 31%. As discussed in Section 3.3.1, this could be mitigated by reducing the number of registers extended with capability metadata. This is an idea that is explored more in the context of physical registers in Chapter 5, where it can be implemented without affecting the architectural view of the register file.

DCache The data cache LUT overhead differs significantly between the two processors: 25% for Piccolo and 61% for Flute. The reason for this divergence is that the granularity of Piccolo's caches, and thus the datapath width between the pipeline and the caches, was already 64 bits wide on the 32-bit processor. This was presumably to allow support for double-precision floating point, even though it is not enabled in our configuration. This allowed capability loads and stores to be supported without increasing the cache interface width. Thus, the only additional logic required for the Piccolo cache is to manage tags and support the new commit interface explained in Section 3.4.2. The Flute caches also had 64-bit granularity, but this needed to be increased to 128 bits for CHERI due to the wider capability format. This is compounded by the fact that Flute also supports atomics, giving the wider capability width a further logic overhead (see Section 3.4.4).

ICache The instruction caches should see no changes due to CHERI. However, both processors see a LUT *decrease* in the CHERI version: -68% for Piccolo and -15% for Flute. This is an artefact of the baseline instruction caches being under-optimised, perhaps including support for logic only required for the data cache. CHERI further differentiates the data and instruction caches, meaning their implementations are

[1]This assumption is validated by the Toooba numbers in Figure 6.3, where the register file is synthesised as FFs and there is an exact correspondence between the expected number of register bits and the number of FFs.

more strictly separated. For example, instruction caches do not need tag support and the Flute instruction cache does not need its granularity widened. This seems to allow Vivado to optimise away more logic from the instruction cache that is only required for the data cache.

CSRs The CSR register file occupies a significant fraction of Piccolo and Flute. This is partially due to the processors' approach to legalisation: handling the complex rules for each CSR separately. In addition, both processors support performance counters to allow in-depth investigation of performance effects. CHERI's impact on the CSR register file is relatively small: only `xtvec` and `xepc` need extending with capability metadata and additional legalisation, and a few additional exception codes are added to report capability exceptions. The DDC and PCC are handled outside of the CSR register file due to the frequency of their access. Piccolo sees a modest *decrease* in CSR area with CHERI: -13%. This appears to be an artefact, with no clear explanation. Flute sees a positive overhead here, as expected: 35%. Flute additionally extends the supervisor mode CSRs, and each extended register adds twice as much metadata as in Piccolo's case.

FPU, integer mul/div, and branch prediction As expected, components largely untouched by CHERI—the integer multiply/divide module, and in Flute the Floating-Point Unit (FPU) and branch prediction—show no area overhead[2]. This helps to reduce the overall fractional area overhead of CHERI.

These numbers provide additional evidence for Hypothesis H.3: the area overheads are less significant for the larger baseline Flute processor than for Piccolo. This is due to a smaller fractional scaling of the key extended components, most notably the ALU, as well as additional unchanged area in components such as the FPU to dilute the CHERI overhead. In fact, various artefacts bring the Piccolo and Flute area overheads closer together than they otherwise would be. This includes Piccolo's baseline cache granularity already being wider than required and Piccolo's anomalous negative CSR area overhead.

4.3 Frequency

CHERI has largely been designed to allow the additional operations to be carried out in parallel with existing logic in typical pipelines. For example, the capability manipulation operations are completed in parallel with the ALU's arithmetic operations: in CHERI MIPS, this was even presented as a completely parallel capability coprocessor. In addition, the bounds check is deliberately placed in parallel with the initial cache operations associated with processing the corresponding request. Ideally, this would mean that there

[2]This is where the Vivado `keep_hierarchy` annotations are useful, as otherwise some CHERI logic is factored into these components.

$F_{\max}$ (MHz)	Design target	Baseline	CHERI
Piccolo	50.0	82.1	76.4
Flute	100.0	102.3	110.7

Figure 4.6: Maximum frequency of the Piccolo and Flute processors synthesised for the VCU-118. The target frequency is that provided by Bluespec Inc. for the baseline processor: no optimisation was attempted beyond this.

is no impact on the critical path, and thus the maximum frequency would be unaffected. However, in practice other effects do perturb the critical path. Some operations, for example atomic operations in the caches, are extended because they must now operate on the capability metadata in addition to the addresses, doubling their bit length. Capability compression operations also have long paths, so must be carefully managed to ensure they are not serialised with other logic. In addition, area overhead from capability operations can cause congestion, increasing the distance between communicating structures.

The FPGA tools cease to optimise once a design meets the specified timing target. By targeting an unobtainable frequency, we can obtain an estimate for the maximum frequency at which the processor would pass timing. Figure 4.6 gives the maximum frequencies after synthesising for the VCU-118, obtained by attempting timing closure at 150 MHz. These show that the CHERI designs clearly achieve the target frequencies. The initial Piccolo design passed its relatively conservative timing target. However, as discussed in Section 3.5.2, some optimisation was required to meet the target for Flute. In the case of Flute, the CHERI $F_{\max}$ *exceeds* that of the baseline: this is an artefact of optimisations performed to meet timing, as of course there is no reason for CHERI to improve critical paths.

For Piccolo, the critical path for both the baseline and CHERI designs is `Stage1`, as might be expected due to its correspondence with three Flute pipeline stages. The baseline critical path starts with the instruction read from the cache, passes through the decoding logic, through the ALU, interacts with control logic, then ends by updating a performance counter. The CHERI path is similar, starting with the fetched instruction from the cache, decoding it, then passing through the ALU and in this case feeding back into the instruction cache BRAM to start fetching the next instruction. This path involves some compression logic, explaining why it is longer than the baseline.

For Flute, in both cases the critical path takes the read instruction from the cache, partially decodes it to determine if it is a return, uses this information to decide whether to use the RAS, possibly performs an add to advance the instruction, then uses the new PC to start lookup of the next instruction in the cache's BRAM. The CHERI path appears to be shorter only due to perturbations of the cache logic, possibly due to the changes to the request interface discussed in Section 3.5.2, or due to changes with the alternative AXI library discussed in Section 3.4.5. The path is unaffected by compression logic, since

prediction only deals with addresses and does not interact with capability metadata, as discussed in Section 3.5.1. In any case, this shows that CHERI can be made to not limit frequency—in this case leaving the critical path largely unaltered—albeit after some optimisation as discussed in Section 3.5.2.

No effort has been made to optimise either the baseline or CHERI designs targeting frequencies beyond the initial target frequencies. Analysis into the impact of CHERI for higher frequency designs would require optimising the baseline processors to target more ambitious frequencies; observing whether the CHERI modifications lengthen the critical path when applied to the optimised baselines; and, if so, seeing if this can be mitigated.

4.4 Performance

The performance of the cores is measured using the CoreMark and MiBench embedded benchmarks. While we have run the SPEC benchmarks on the Flute core, I focus on embedded benchmarks here, leaving SPEC analysis for Toooba in Chapter 6. These benchmarks are introduced in Appendix B, which describes their operation sufficiently to explain performance observations. The benchmarks are used to measure the performance impact of moving from RISC-V to CHERI hardware without enabling the additional protections in software. Finally, the efficiency of the capability operations is investigated by observing the overhead from enabling the protections.

All benchmarks are compiled with `-O3` with the same version of the LLVM compiler modified to support capabilities, with capability code enabled by an option for pure capability code. The benchmarks run atop Cheri-FreeRTOS for CHERI benchmarks and the corresponding version of FreeRTOS for non-CHERI benchmarks. Each is run ten times, and the values shown are for the sum of these ten runs. Runs are largely deterministic, with only DRAM latencies introducing variability (typically less than 0.1%) between runs, so error bars would not be meaningful on the graphs.

4.4.1 Legacy performance

I first show the performance impact of legacy RISC-V operations, relative to the baseline cores. The CHERI ISA offers complete binary compatibility: the processor boots into a state where non-capability RISC-V code will run as if on an unmodified processor (with no protection benefit). This is principally achieved by setting PCC and DDC to be fully permissive capabilities, implicitly authorising all instruction and memory accesses. This allows measurement of the impact of the modifications on the core when the protections are not used. Within the pipeline, these should be cycle-for-cycle identical to the baseline processor. However, there are some differences within the memory subsystem. The tag controller must lookup tags for memory as it is accessed, even though these will all be

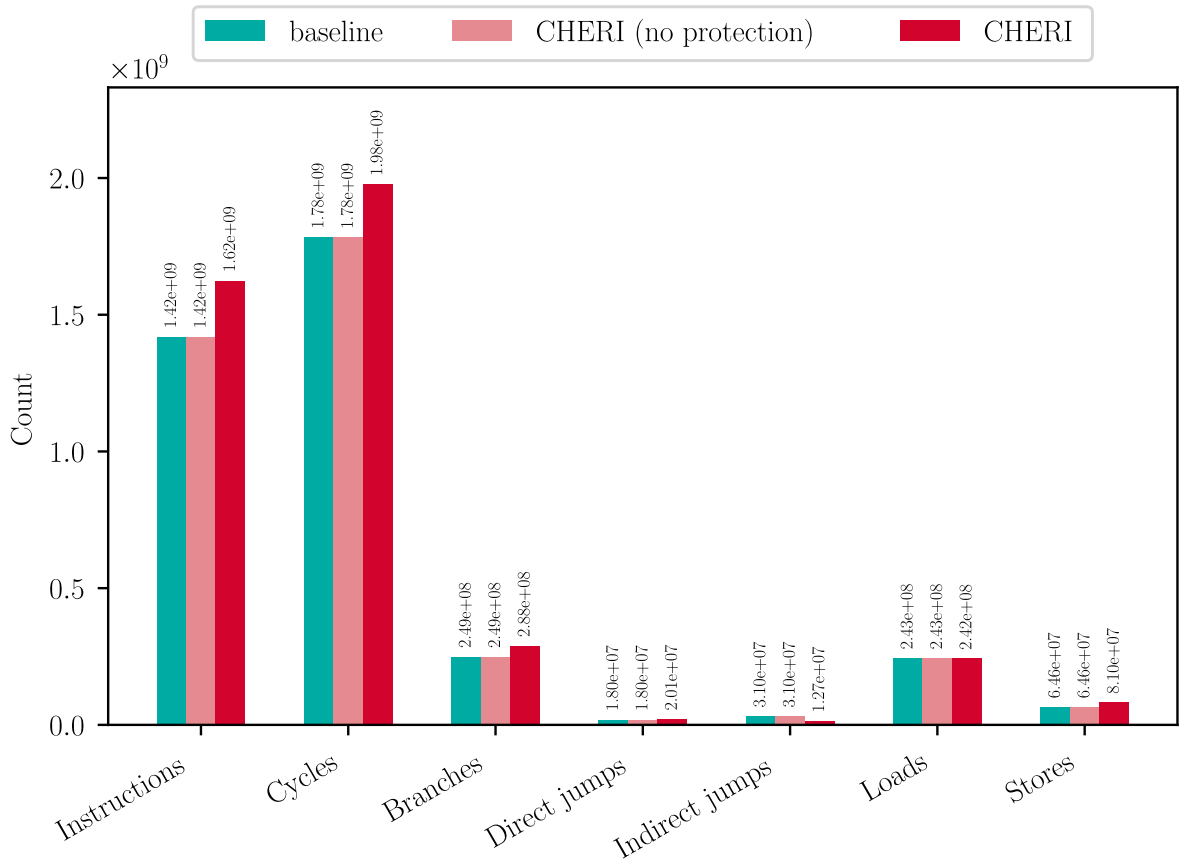

Figure 4.7: CoreMark run statistics for the Piccolo baseline core, the CHERI core running the baseline software (no protection), and the CHERI core running pure capability code, including relevant performance counters. The working set fits in cache, so negligible cache misses are observed.

unset, since no capabilities will reside in memory. The tag controller is somewhat optimised for this case due to its hierarchical table [63]. In addition, the modified AXI interconnect used to allow tag support may have different latency and bandwidth properties.

The CoreMark results shown in Figure 4.7 and Figure 4.8 include values for the baseline and CHERI cores running without protections, enabling this comparison. As can be seen, in both the cases of Piccolo and Flute, the overheads are negligible (within 1%). In fact, the Flute core is slightly *faster* with CHERI, perhaps due to favourable perturbation of latencies in the AXI interconnect. This is not a significant or fundamental effect. The reported branch, jump, and redirect counters also differ very slightly between the baseline and CHERI implementations of Flute: this is due to minor perturbations of the BTB.

The MiBench results reinforce the low performance overhead without using capability features in software. All instruction counts are identical, since identical binaries are run on both cores. No graph is shown, because cycle overheads are negligible. In fact, once again, the Flute overheads are almost all slightly negative. The performance counters confirm that the overall cycle differences correspond to time spent waiting for caches, again suggesting AXI latency changes as the cause of the discrepancy. The average cycle

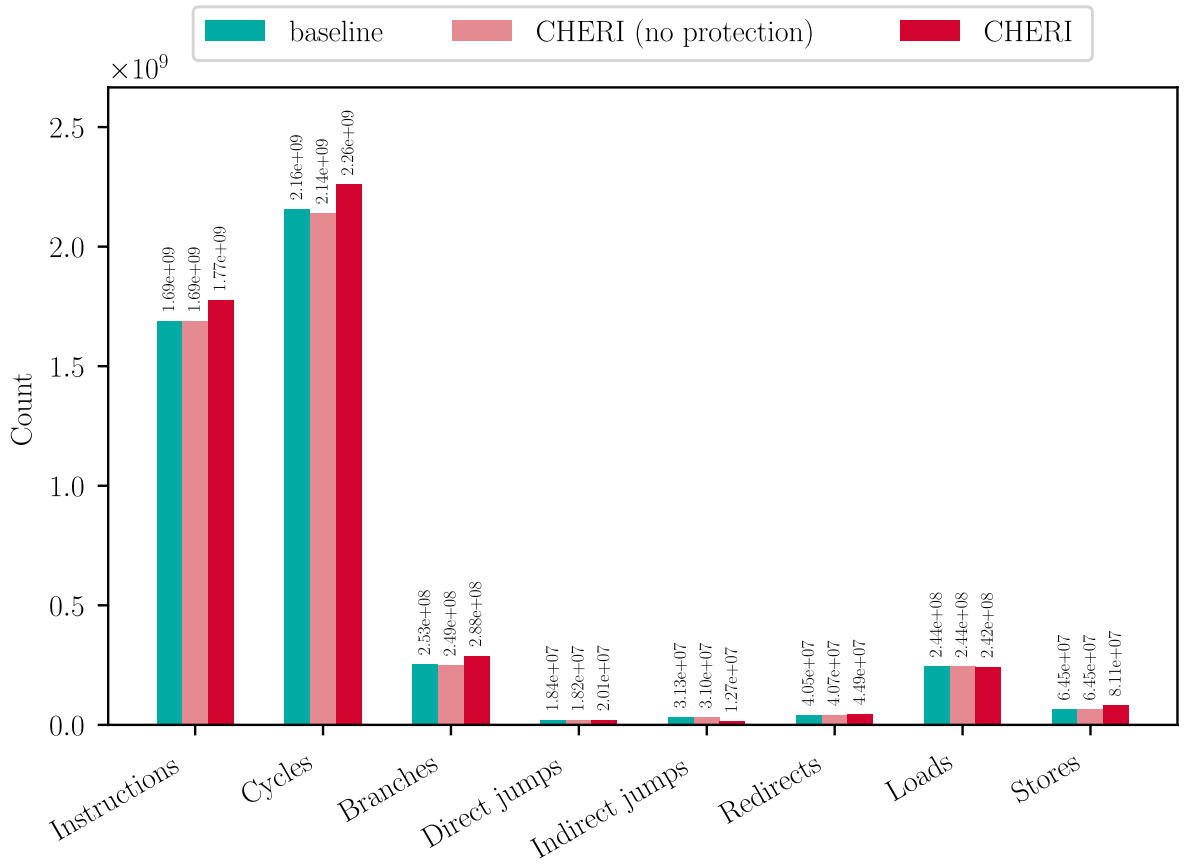

Figure 4.8: CoreMark run statistics for the Flute baseline core, the CHERI core running the baseline software (no protection), and the CHERI core running pure capability code, including relevant performance counters. The working set fits in cache, so negligible cache misses are observed.

difference for Piccolo and Flute is less than 1% either way.

4.4.2 Capability performance

The performance of CHERI code is now shown, relative to the baseline code on the same CHERI processor. This allows some investigation of the impact on IPC and thus the performance of the added operations, as well as effects due to increased cache pressure and tag lookups. Note that the performance is heavily sensitive to dynamic instruction overhead, which depends on the effectiveness of LLVM CHERI code generation and optimisation. While the compiler is not a contribution of this thesis, some investigation of the source of instruction overheads provide context for the performance results.

CoreMark results are shown for Piccolo in Figure 4.7, and Flute in Figure 4.8. As mentioned in Appendix B, CoreMark is designed to only test the core, so the performance does not depend heavily on the memory subsystem. For Piccolo, the cycle overhead is 11% for an instruction overhead of 15%. This instruction overhead indicates potential for code generation optimisation, as few capability-manipulating instructions should be required given the CoreMark operations are not pointer-heavy. However, instruction overheads will

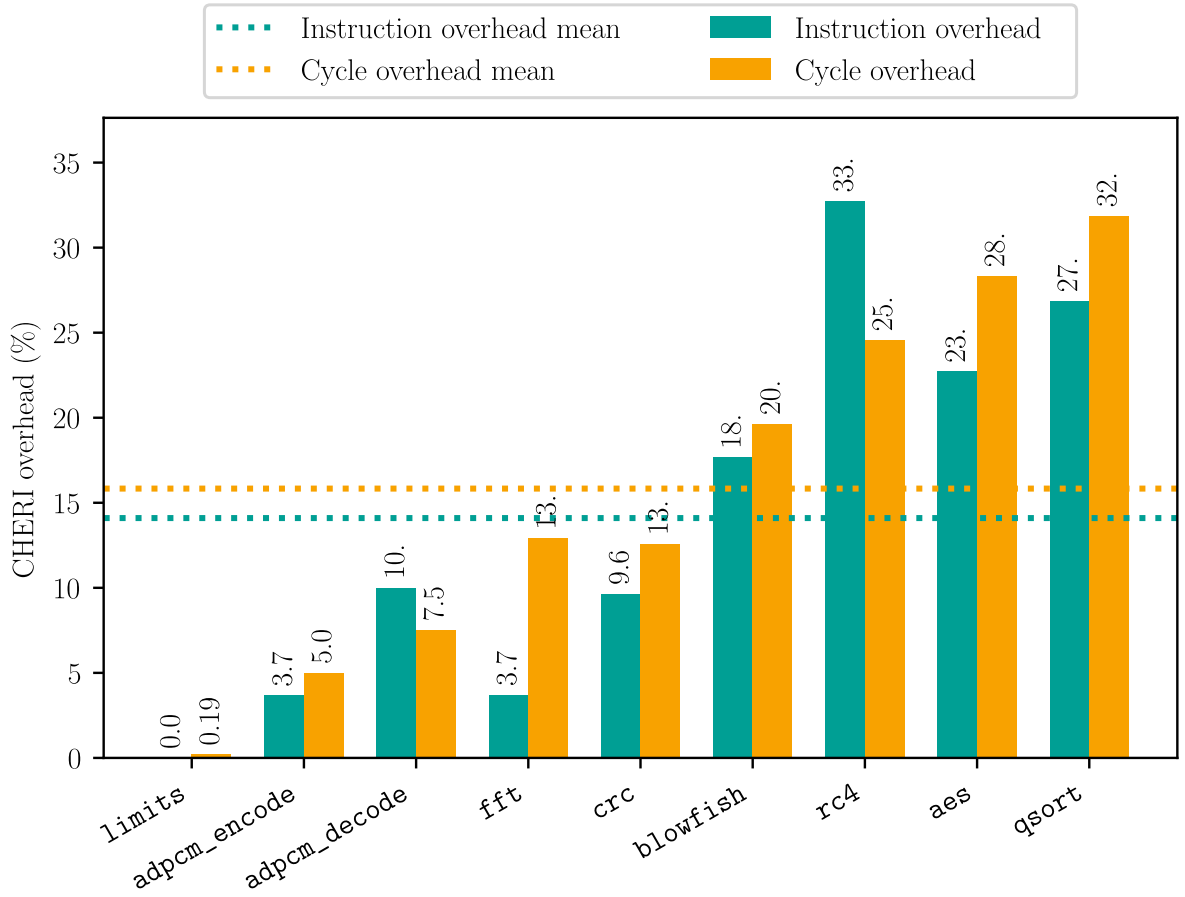

Figure 4.9: MiBench overhead of CHERI for the Piccolo core. Arithmetic mean instruction overhead is 14% and cycle overhead is 16%.

be investigated more thoroughly for MiBench, where even higher overheads are seen. Flute sees smaller overheads: 6% cycles and 5% instructions. The discrepancy here may be a consequence of the toolchain being more mature for 64-bit targets. In any case, the cycle overhead scales comparably with the instruction overhead for both processors, indicating that microarchitecture is not adversely affecting performance. This might be expected, since the capability operations added to the processor all execute in a single cycle[3], leaving only the memory subsystem to cause performance anomalies. The merged register file also ensures that register pressure is similar for capability and baseline code. Note that the CHERI code generation favours branches over indirect jumps: this is not significant for Flute, which does not have very sophisticated branch prediction and only has a small mispredict penalty.

Figure 4.9 and Figure 4.10 show the dynamic instruction and cycle overheads when running MiBench for the Piccolo and Flute cores respectively. Piccolo sees a 16% cycle overhead for a 14% instruction overhead. Flute sees a similar 16% cycle overhead for a 16% instruction overhead. I investigate a few of the benchmarks showing noteworthy overheads to determine the causes: `limits`, `fft`, `rc4`, and `qsort`.

[3]With the exception of `CTestSubset`, which is not generated by the compiler.

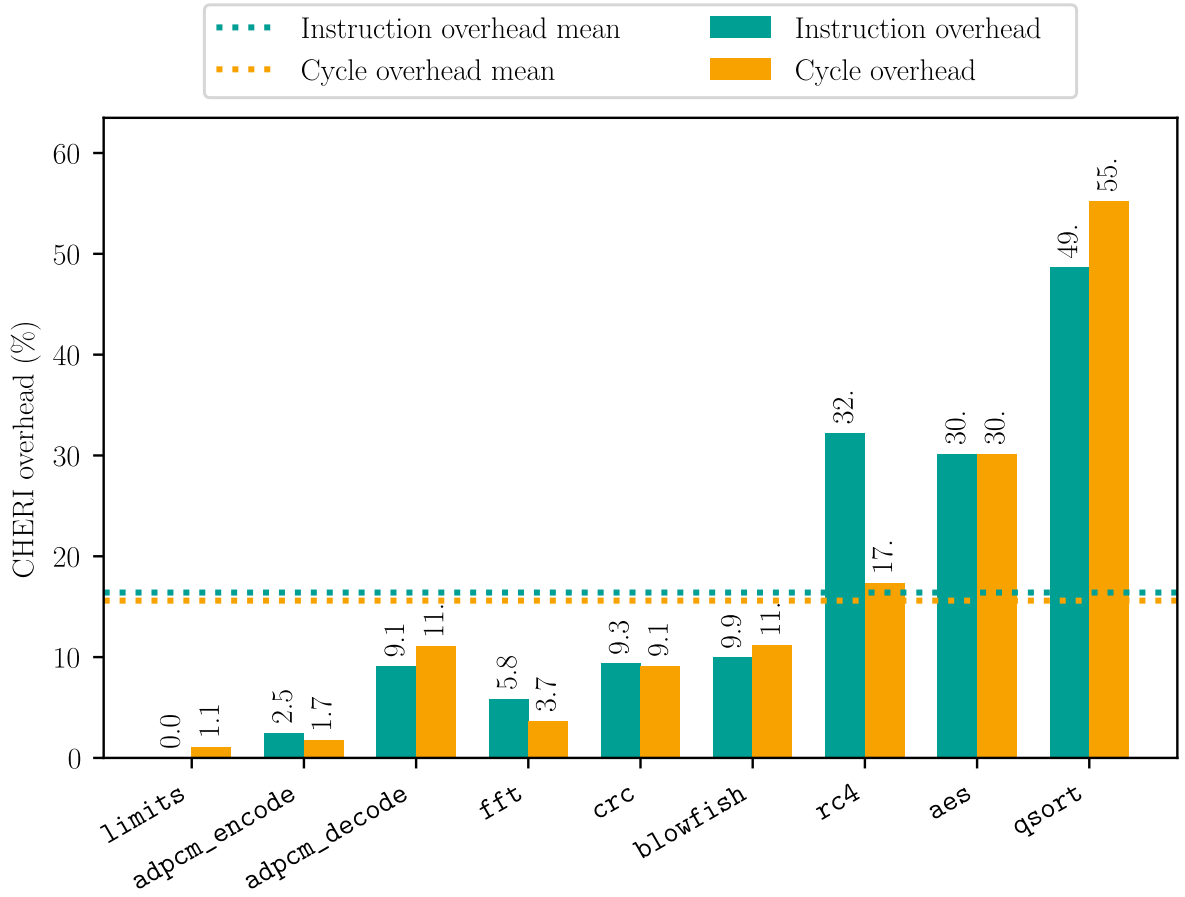

Figure 4.10: MiBench overhead of CHERI for the Flute core. Arithmetic mean instruction overhead is 16% and cycle overhead is 16%.

`limits` This benchmark shows zero instruction overhead: the capability code generated is identical to the baseline, except with capability versions of some instructions. This is possible because all variables fit in registers, so no stack-allocated variables are required. There is a small cycle overhead, seemingly because of one additional cache miss as the memory layout is perturbed to make room for capability metadata.

`fft` This benchmark has one of the lowest instruction overheads on Piccolo, but a comparatively much larger cycle overhead, indicating the microarchitecture might be introducing wasted cycles. From the performance counters, the main contributor to cycle overhead is instruction cache misses. The baseline code spends 30% of its cycles waiting for instruction cache misses to be resolved. This grows to 35% for the pure capability code. This is the result of a 31% miss increase despite total instruction cache loads only increasing by 3.5%: the miss rate increases from 1.4% to 1.7%. Each instruction miss takes the same time for both the baseline and CHERI code: 34 cycles. This seems to be a case of either the additional CHERI instructions pushing the working set past the 4 KiB of the instruction cache, or more likely unfortunate aliasing in one of the cache sets causing thrashing of instructions in a key loop. This is the benchmark that benefits the most from the capability-aware

compressed instructions described in Section 3.3.4: without them, Piccolo sees a 26% cycle overhead. Flute does not see the thrashing effect: the increased cache size means that both the baseline and the CHERI implementation spend a much smaller fraction of their cycles waiting for the instruction cache.

`rc4` This benchmark sees one of the highest instruction overheads, but the cycle overhead is low in comparison. The core encryption loop grows from 21 to 28 instructions when compiling for pure capability: the added instructions arise from different loop factoring decisions and do not seem inherent to capability code. Figure 4.11 shows the encryption and decryption inner loop, showing that the overhead is caused by unnecessary additional instructions. Four instructions are added for rederiving globals: since there is no additional register pressure in the capability case, the cause of this is unclear. Three instructions are added to rederive the `state` variable 8 bytes into the `ctx` structure, while the baseline factors this out of the loop. This is likely because the `addi` instructions can not be reordered with the `CIncOffset` instructions, pending further code generation optimisation. Since the added instructions are cheap arithmetic rather than expensive memory accesses, the cycle overhead is smaller than the instruction overhead.

`qsort` This has significant instruction overhead, pending pure capability code generation improvements. The low-level `swap` implementation within `qsort`, for instance, grows by 76% in static instructions when compiled for pure capability. The `qsort` implementation is used from the `Newlib` library, which has been augmented with CHERI support, rather than being contained in the benchmark itself. We can also note the significant difference between the instruction overhead in Piccolo and Flute. This appears to be due to a granularity issue. The elements being swapped consist of three integers, padded to four for alignment, and a double precision floating-point number. This makes the structure 24 bytes wide as compiled by LLVM. However, the `swap` function has a fast case copying `long`s at a time, and a slower fall-through case that copies `char`s at a time if alignment is not sufficient. In supporting CHERI, the fast case was changed to copying a capability at a time, both exploiting the wider architectural access widths and preserving capabilities in the swapped data. However, this means the structure width for this benchmark is no longer a multiple of the fast copy width in the 64-bit Flute processor where capabilities are 16 bytes, so falls through to the byte-by-byte case. This makes each copy 24 iterations as opposed to the baseline's three. Piccolo does not have this issue as 24 is a multiple of its capability width of eight bytes. The cycle overhead scales quickly with the instruction overhead as the additional instructions tend to be expensive loads and stores: stack spills and instructions for swapping.

The MiBench benchmarks were also run both with and without the capability-aware compressed instructions enabled. The numbers reported throughout this section show

performance with them enabled. Without them, Piccolo incurs a mean cycle overhead of 19% (3% worse) and Flute 16% (no change). While not a huge effect overall, benchmarks where instruction cache misses are an issue see large benefit from the optimisation: without it, `fft`'s cycle overhead grows to 26% and `qsort`'s to 41% on Piccolo. Flute is less sensitive to instruction cache pressure due to its increased cache sizes, especially for these embedded benchmarks.

In summary, while the instruction and cycle overheads are high, at least some of the overhead appears to be a result of artefacts and code generation anomalies. This is made worse since many embedded benchmarks consist of a tight core loop that is very sensitive to added instructions. As future work, compiler improvements could ensure that the CHERI compiler matches the baseline except where capability operations require otherwise. As well as bringing down time spent executing unnecessary instructions, this would also avoid issues due to instruction cache pressure, which is especially relevant for these microcontrollers with such small caches.

4.5 Power

Power is an important metric for some microcontrollers, especially in battery-powered devices. Unfortunately, while area itself can give some estimate of how power will scale, it is difficult to determine ASIC power consumption based on an FPGA design. FPGA tools provide an estimate of power usage of the synthesised design. However, this is not fully representative of ASIC power, particularly as the use patterns of the synthesised components are not known by the synthesis tool, and power usage scales significantly with how often transistors are switched. The results in this chapter are gathered using Vivado's "vectorless power analysis" [141]. This assumes a certain switching distribution for input pins (switching one in eight cycles), and propagates this through the design. Future work could improve the accuracy of these measurements by capturing benchmark runs in a way that could provide Vivado with more dynamic information. Even further, on-board power measurement tools could be used to capture actual power used during a benchmark run. However, note that these techniques would increase accuracy of measuring the power used by the FPGA, rather than more accurately approximating power in an ASIC context.

Where microcontrollers are connected to DRAM, a large fraction of the power drawn is that required to drive the relatively long wires to DRAM. This means that DRAM traffic overhead can also increase power consumption.

Figure 4.12 shows the power of the designs reported by Vivado. It gives both the logic-only power and the overall power, which includes power for clocks, signal transmission, and BRAMs. The overhead is approximately the same in either case, and scales roughly with the LUT area of the designs.

The DRAM overhead can also be used as an estimate for power overhead. CHERI causes

```
struct {int x; int y; char state[256]} *ctx;
char *src, *dest; int i; char x, y, sx, sy, b, d;
ctx->x = (ctx->x + 1) & 255;
                                  auipcc      ct0, 35
                                  clc         src, 1246(ct0)
  addi t1, ctx->x, 1              addi        t1, ctx->x, 1
  andi ctx->x, t1, 255            andi        ctx->x, t1, 255
sx = ctx->state[ctx->x];
                                  addi        t2, ctx->x, 8
  add  t3, ctx->state, ctx->x     cincoffset  ct3, ctx, t2
  lb   sx, 0(t3)                  clb         sx, 0(ct3)
y = (sx + ctx->y) & 255;
  add  t4, ctx->y, sx             add         t4, ctx->y, sx
  andi ctx->y, t4, 255            andi        ctx->y, t4, 255
sy = ctx->state[ctx->y];
                                  addi        t5, ctx->y, 8
  add  t6, ctx->state, ctx->y     cincoffset  ct6, ctx, t5
  lb   sy, 0(t6)                  clb         sy, 0(ct6)
  add  &src[i], i, src            cincoffset  &src[i], src, i
  lb   src[i], 0(&src[i])         clb         src[i], 0(&src[i])
ctx->state[ctx->y] = sx;
  sb   sx, 0(t6)                  csb         sx, 0(ct6)
ctx->state[ctx->x] = sy;
  sb   sy, 0(t3)                  csb         sy, 0(ct3)
b = ctx->state[(sx + sy) & 255];
  add  t7, sy, sx                 add         t7, sy, sx
  andi t8, t7, 255                andi        t8, t7, 255
                                  addi        t9, t8, 8
  add  t10, t8, ctx->state        cincoffset  ct10, ctx, t9
  lb   b, 0(t10)                  clb         b, 0(ct10)
d = src[i] ^ b;
                                  auipcc      ct11, 35
                                  clc         dest, 1180(ct11)
  xor  d, b, src[i]               xor         d, b, src[i]
dest[i++] = d;
  add  &dest[i], i, dest          cincoffset  &dest[i], dest, i
  addi i, i, 1                    addi        i, i, 1
  sb   d, 0(&dest[i])             csb         d, 0(&dest[i])
```

Figure 4.11: Compiled inner loop body for the `rc4` benchmark for baseline (left) and pure capability code (right). Registers are renamed for clarity. Extra CHERI instructions are highlighted: orange shows rederived globals and blue shows instructions arising from not factoring out the indexing into `*ctx` to find `state`.

Power usage	Baseline (mW)	CHERI (mW)	Overhead
Piccolo (logic only)	20	32	60%
Piccolo (overall)	62	98	58%
Flute (logic only)	84	116	38%
Flute (overall)	274	372	36%

Figure 4.12: Power usage of the Piccolo and Flute processors synthesised for the VCU-118, as reported by Vivado.

additional DRAM overhead, both as a result of the tag controller's tag lookups, and metadata that must be stored alongside any pointers. This is again an area where microcontrollers expect increased overhead compared to larger processors. This is because smaller caches hide less of the pointer metadata overhead, especially since the caches are write-through. Figure 4.13 shows the DRAM traffic overheads across the MiBench benchmarks, with Piccolo reporting a 61% overhead and Flute a 42% overhead. Coincidentally, these match surprisingly well with the power overhead reported by Vivado. As with performance overheads, it is possible these will improve with code generation improvements, as fewer additional instruction cache misses cause fewer DRAM lookups.

4.6 Security

Security evaluation of the CHERI model itself has been carried out in various other work (see Section 2.4.1), for example by Joly, ElSherei, and Amar [65]. This is largely outside the scope of the thesis.

The implementations were evaluated for security using a Common Weakness Enumeration (CWE) test suite developed by Galois, and the Fett bug-bounty red-teaming exercise [35]. The security evaluation was carried out by other members of the CHERI team, so is only summarised here.

The results of CWE testing were largely as expected, in particular highlighting the protection provided by the core against buffer overruns and exploitable numeric errors affecting pointer arithmetic. Some buffer overruns were not caught on Piccolo due to its extreme capability bounds compression. However, the software has been extended to make these unexploitable by providing the necessary padding.

The Fett bug-bounty program required some work to debug the processor and ensure it supports all of the required features. The exercise evaluated the effectiveness of the CHERI extensions at their purpose: to protect against attack by a motivated adversary. Three exploitable bugs were found: an incorrect capability of `realloc` in the allocator, a problem with bounds checking on variadic argument arrays, and an issue with the kernel not checking the bounds of its capabilities in software before performing optimisations.

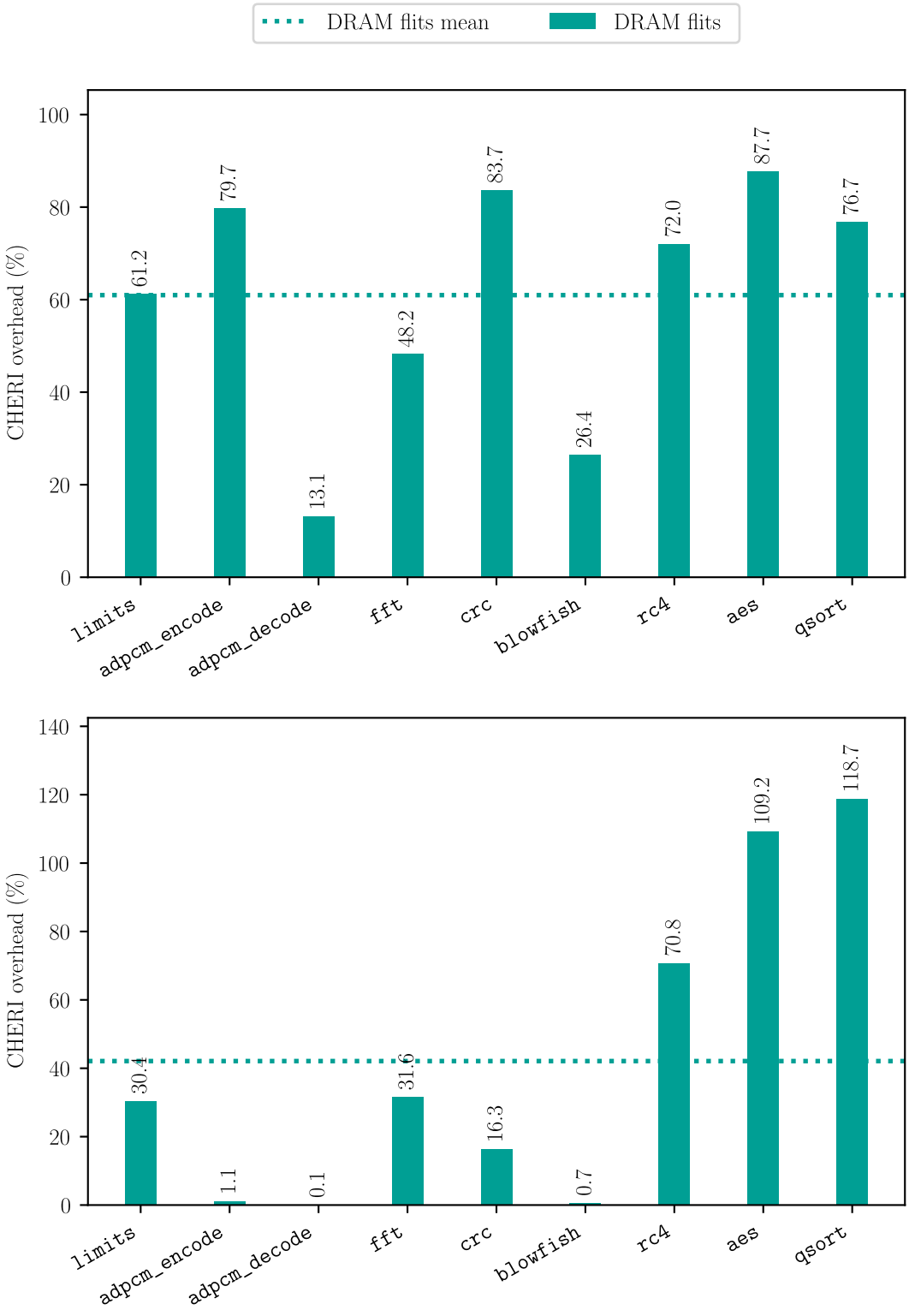

Figure 4.13: MiBench DRAM traffic overhead of CHERI for the Piccolo (top) and Flute (bottom) cores. Arithmetic mean Piccolo overhead is 61% and Flute overhead is 42%. The count is of AXI flits: single transfers within a burst.

Although hardware bugs were in-scope for the exercise, most attackers seemed to focus on more familiar software bugs, so no hardware bugs were reported.

4.7 Future work

For the microcontrollers and benchmarks examined, additional instructions generated by the compiler that are not inherent to the capability model are perhaps the main cause of performance overhead. This is exemplified in Figure 4.11. Additional work is planned on compiler optimisation, identifying the source of added instructions and enabling optimisations to mitigate them. This is likely to accompany additional work on CHERI RISC-V architectural optimisation: examining whether additional instructions, or immediate forms of existing instructions, could improve code density.

The comparisons shown here have kept all parameters the same between the baseline and CHERI cores to measure the overheads objectively. However, this may not be the realistic approach taken to augmenting microcontrollers with CHERI: capabilities may change the tradeoffs that need to be considered in parametrisation. For example, given the area added to support CHERI in the ALU, it may be worthwhile to increase cache sizes or associativity to keep performance and DRAM traffic overheads low. Future work could examine some of these parameters to produce a recommendation of how to augment microcontrollers with CHERI in practice.

All evaluation has been performed on FPGA platforms, due to the costs associated with silicon synthesis. It is likely that the different synthesis target will produce different evaluation results, as discussed in Section 2.1. Further evaluation of the cores using silicon synthesis tools therefore also remains as future work.

4.8 Summary

This chapter gives an initial quantitative answer to Hypothesis H.1: depending on what overheads are acceptable for a given context, CHERI is an option for increasing microcontroller security. It has been shown that CHERI can provide spatial safety for RISC-V microcontrollers to protect at least against common, existing attacks, as well as initial evidence that it protects against motivated adversaries via a bug-bounty program. The (LUT) area overheads of Piccolo and Flute were 62% and 49% respectively: the tag controller and capability manipulation logic form the majority of the overhead. After optimisation of Flute, timing is not affected, and it seems that CHERI does not fundamentally change critical paths for these types of microcontrollers. The performance overheads are negligible for unmodified code (no security benefit), and approximately 16% (average MiBench cycles) for CHERI-augmented code on both Piccolo and Flute. Code generation issues

were identified that are not fundamental to capabilities, and significantly increase the run-time overheads. Power overheads are measured as approximately 60% for Piccolo and 40% for Flute across both logic overhead and DRAM traffic overhead.

These numbers also give an initial answer to Hypothesis H.3: the area and power overheads are smaller for Flute than for Piccolo. This is primarily due to additional unmodified baseline logic, such as the FPU, reducing the fractional overhead. For dynamic measurements of power, this scaling is primarily caused by larger caches being able to absorb more of the memory overheads arising from capability metadata and increased instruction cache pressure. However, the performance overhead appears unchanged between the two processors. Flute is able to absorb the instruction overhead better than Piccolo due to its larger caches, but the `swap` anomaly discussed for the `qsort` benchmark raises Flute's instruction overhead, negating this effect.

Chapter 5

CHERI for application-class processors

This chapter describes the first open implementation of CHERI RISC-V for an out-of-order core: an extension to MIT's RiscyOO processor [145]. The architectural changes from Chapter 3 all apply, as well as much of the microarchitectural implementation. This chapter explores the new challenges and opportunities presented for implementing CHERI for a large, superscalar, out-of-order design. Particular attention is paid to ensuring the capability model cannot be violated in speculation, motivating further refinement of the CHERI RISC-V ISA.

This chapter answers general questions about implementing CHERI for application class cores, addressing Hypothesis H.2. Finally, the implementation enables exploration of revocation in a new context, targeting Hypothesis H.4 in Chapter 7.

5.1 Characteristics of application-class processors

As discussed in Section 2.1, application-class processors have been developing to ever higher clock frequencies, instructions-per-clock, and concurrency. Beyond technology node improvements, this has been achieved via advancements in microarchitecture, for example increasing superscalarity, extracting instruction-level parallelism, and out-of-order execution, allowing memory latency to be hidden by speculation and executing independent instructions.

Unlike for microcontrollers, application-class cores generally require support for memory virtualisation, so are armed with MMUs for efficient memory translation. This doubles as a security mechanism, as memory can be mapped per-process, preventing corruption among processes, or between processes and the kernel, unless memory is explicitly shared. However, as discussed in Section 2.2, this is far from a complete security solution, as spatial safety can still be violated within a process. Other partial solutions are already widely

adopted, and their performance penalties accepted, to address this issue, for example ASLR [17].

The attack-surface is very different from the microcontroller world, as code is often acquired dynamically and from untrusted sources, such as downloaded from the internet. Servers accessible via the internet are the highest payload targets for attack, as a single attack can compromise millions of users' data, sometimes via a mistake in a single line of code, for example Heartbleed [41]. This makes the core running the application code the most obvious choice to add architectural protection.

5.2 Baseline processor

RiscyOO is an open-source RISC-V superscalar, out-of-order processor developed by MIT. The processor was designed in Bluespec, with the particular aim of being modular and extensible. A full description of its design is given by Zhang et al. [145], with a general description focusing on the details relevant to the CHERI modifications reproduced here. It supports parametrisable superscalar execution, and has been synthesised for ASIC in a 32 nm process at 1 GHz [145]. The processor has been augmented by Bluespec Inc. with support for compressed instructions, and to fit the same SoC environment as Piccolo and Flute. They have dubbed this modified processor Toooba, which is the name used throughout the thesis [59].

Toooba supports the single combination of 64-bit RISC-V with the A, C, D, F and M extensions, meaning it has support for atomics, compressed instructions, multiply, and double-precision floating-point. As shown in Figure 5.1, the processor is split into: a front-end (instruction fetch and decode), register rename, the reorder buffer, instruction-specific pipelines, and finally a commit stage. There are three pipeline types: memory, arithmetic (including jumps and branches), and one for floating-point, integer multiply and integer divide. While the number of arithmetic and floating-point pipelines is parametrisable, there is always exactly one memory pipeline. The front-end and commit are in-order, but pipelines are able to process the instructions within the reorder buffer out-of-order. The processor has support for machine, supervisor, and user modes, meaning full support for address translation. This allows it to boot application-class OSs including Linux and FreeBSD.

Toooba is parametrised in its superscalarity by `SupWidth`: the number of instructions that can be executed per cycle when operating at full bandwidth. Throughout the chapter, `SupWidth` is fixed at two. Various other processor attributes are parametrised, including cache sizes, maximum number of in-flight instructions, and sizes of reservation stations. While the CHERI modifications support varying these parameters, the "small" configuration is consistently used for evaluation: this allows investigation of all the key Toooba features without causing complications due to over-congestion on FPGA. The

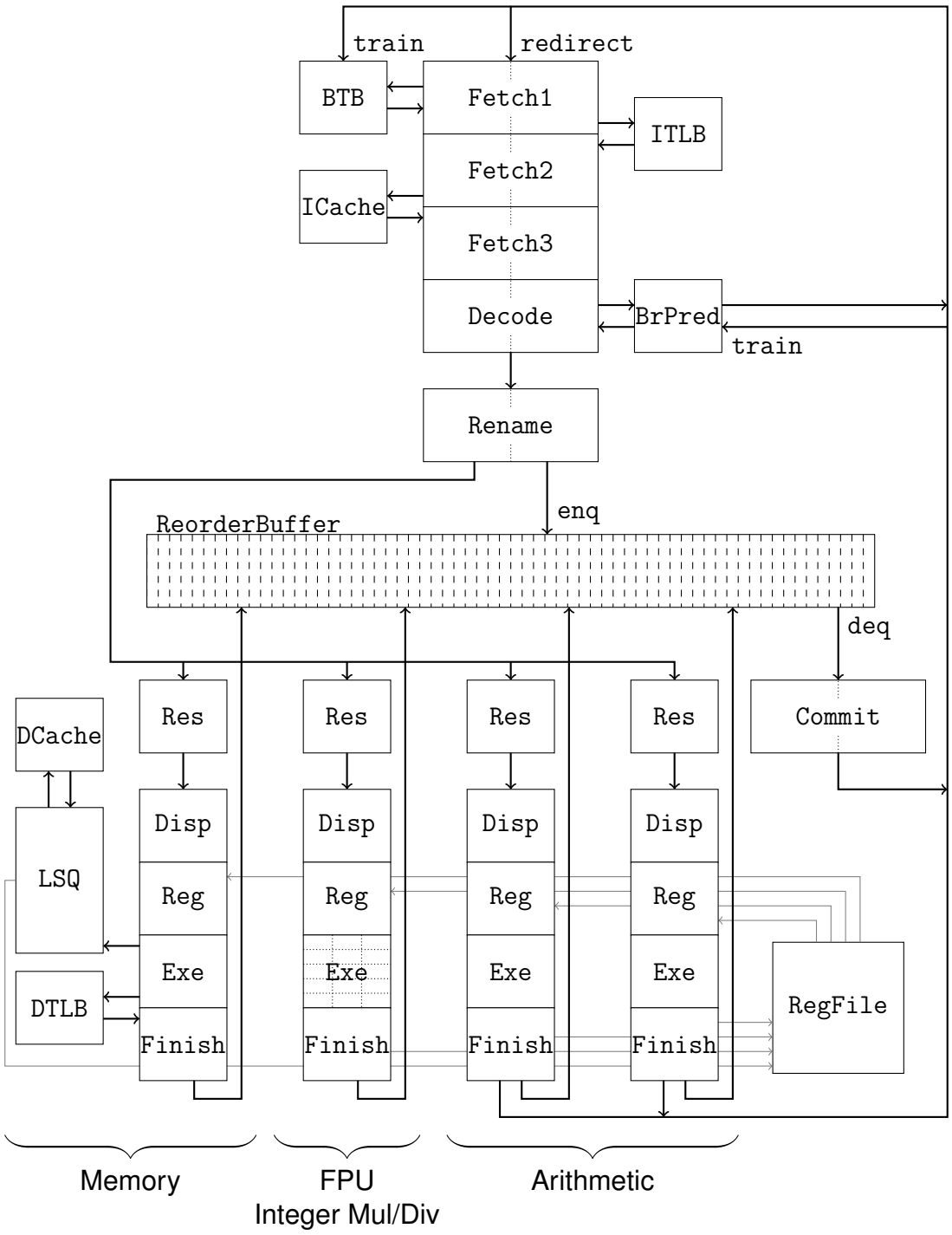

Figure 5.1: The Toooba processor. Forwarding paths from the arithmetic pipelines' `Exe` and `Finish` stages to all pipelines' `Reg` stages are omitted.

exact parameters of this configuration are listed in Figure 6.1.

Toooba's front-end fetches up to `SupWidth` 32-bit instructions at a time (which may be more instructions due to compression). The front-end pipeline is split into four stages: three for fetch and one for decode. The first fetch stage takes the predicted next PCs from the BTB and begins looking them up in the TLB, while also using them to predict the PCs for the next cycle. The second stage completes the TLB lookup, detecting memory management exceptions, then issues the request to the instruction cache or Memory-Mapped Input/Output (MMIO) controller. The final fetch stage completes the instruction lookup, passing its 16-bit instruction parcels to the decode stage. Decode combines parcels corresponding to halves of 32-bit instructions, decodes them, and produces a more accurate PC prediction using decoded fields from the instruction, the direction predictor, and the RAS.

Once decoded, instructions are passed into the rename stage where their architectural registers are mapped onto physical registers. The number of physical registers in the processor depends on the reorder buffer size, allowing every reorder buffer entry to have its own physical register, in addition to one per architectural register (including floating-point). This prevents physical register availability ever causing a structural hazard: rename simply allocates a destination physical register for every instruction that comes through, even if the instruction does not need it. A table is maintained of the current mapping from architectural to physical registers. The rename stage enqueues each instruction into the reorder buffer and into the reservation station of a relevant pipeline. For system instructions and any other instruction that must clear the pipeline, this is detected in rename and no more instructions are issued until all prior instructions have been committed.

The reorder buffer contains a record for each in-flight instruction, containing all information required for when the instruction eventually commits. This includes information to determine when the instruction is no longer speculative based on when preceding predicted branches are architecturally resolved. Importantly for the CHERI modifications, each row contains the full PC of the corresponding instruction, and PC of the next predicted instruction. For a sense of scale, each reorder buffer entry is approximately 200 bits. The number of rows in the reorder buffer, i.e. the maximum number of in-flight instructions (except a few in the fetch, rename and commit stages), is also configurable, with the "small" configuration containing 64 entries.

All instructions are serviced by a particular type of pipeline, each of which has a similar structure. Each pipeline has a reservation station that accepts relevant instructions from the rename stage and buffers them until their register dependencies have been resolved. The pipelines then have a dispatch stage that updates the required state in the reservation station and passes an instruction to the rest of the pipeline. Register read latches the relevant values from the physical register file or forwarding paths if available. Execute then performs the required operation. The finish stage then updates the reorder buffer entry and typically writes the result to the physical register file.

The arithmetic pipeline is special in that all its instructions have a fixed latency. This allows its reservation station to mark its instructions' destination registers ready as soon as the instruction is dispatched. This is possible because any pipeline can safely schedule an instruction that depends on this value knowing that it will be available through forwarding by the time the dependent instruction reaches register read. The arithmetic pipeline is therefore the only pipeline that forwards its outputs from the execute and finish stages to all pipelines' register read stages. The arithmetic pipeline's register read stage also fetches the instruction's PC and predicted next PC from the reorder buffer and has the possibility of reading a CSR. Since control-flow instructions are handled by the arithmetic pipelines, their finish stages also resolve speculation. This involves redirecting the fetch stage if required, updating instructions that are speculative based on the current instruction, and providing information to train future branch predictions.

The pipeline supporting floating-point and integer multiply and divide is a standard non-forwarding pipeline, but the execute stage is split into several variable latency stages depending on the operation.

The design of the memory pipeline mirrors that of the other pipelines, with dispatch and register read performing the same operations. The execute stage performs address calculation by adding the immediate to the read register value if required, and performs the required interactions with the Load/Store Queue and data TLB. The Load/Store Queue is also signalled from other parts of the processor: a slot is initially requested in the rename stage, and the memory pipeline's finish stage has to commit the access once all exceptions are resolved. The finish stage is triggered when the TLB issues a response, and any exceptions are handled. Crucially, the finish stage does not require the access to have occurred for a load, giving the core its ability to process out-of-order. Memory responses are handled asynchronously, possibly after commit, writing their data to the relevant physical register.

The Load/Store Queue and cache are most relevant to the revocation work, so are described in more detail in Section 7.3.1.

The commit stage is in-order, so acts as a gathering point for instructions that interfere with others' execution, including exceptions and system instructions. The system and exception cases perform the required flushing of all of the in-flight instructions and redirect the fetch stage. The common case just dequeues the reorder buffer and commits the rename of the destination register.

5.3 CHERI implementation

This section discusses our CHERI augmentation of the Toooba core. Note that the implementation was completed in collaboration with Jonathan Woodruff and Alexandre Joannou: see Section 1.3. While effort was made to make the changes efficiently, little optimisation of area has been performed. The implementation has an interesting tradeoff space: many structures can have capability metadata included inline by widening the existing hardware, or can sacrifice performance by restricting the amount of metadata stored. I identify these tradeoffs when they arise, but mostly lean towards taking the widening approach initially, partly due to the simpler implementation this implies. Investigating alternative points in the tradeoff space remains as future work, hopefully enabled by this initial implementation.

Much of the implementation work for CHERI for microcontrollers from Chapter 3 also applied to Toooba. As discussed in Section 3.4, the library for capability compression was reused for Toooba, and the CHERI-tag-aware AXI interconnect and tag controller could also be directly reused. Capability instruction and CSR encoding tables could also be shared. TestRIG was used to verify the processor, reusing the effort in creating templates for Piccolo and Flute. The architecture details, including merged register file, capability encoding mode, Sentry mechanism, and capability-aware compressed instructions all apply.

5.3.1 CHERI instruction pipeline

For the original CHERI MIPS implementation, the capability arithmetic was described as a capability co-processor [136]. As discussed in Section 3.4.3, the more natural model in CHERI RISC-V was to think of the capability arithmetic extending the ALU. When extending a superscalar processor, more options present themselves. Since Toooba has multiple different types of pipeline, capability instructions could be implemented alongside arithmetic, memory access, or even have a pipeline to itself.

I decided to augment each arithmetic pipeline with capability arithmetic, allowing pointer arithmetic to be performed largely cycle-for-cycle identical to the baseline, except where additional instructions are required for security. The arithmetic pipelines are the most natural choice to extend with capability arithmetic. These pipelines are designed for single-cycle operations, unlike the floating-point and memory pipelines, which tolerate variable latency. By design, the capability arithmetic can all be carried out in a single cycle. The arithmetic pipeline can therefore forward results of capability manipulations to all pipelines. In addition, the number of arithmetic pipelines scales with the superscalarity of the processor, meaning capability-aware arithmetic pipelines maximise capability operation throughput.

The memory pipeline remains unmodified, except to allow the new access types and perform the required capability checks to ensure accesses are authorised. This approach

requires the capability bounds checks, including the expensive bounds decompression, to be duplicated between the memory pipeline and arithmetic pipelines. However, since Toooba is a much larger processor than those discussed in Chapter 3, this area overhead is expected to be small as a fraction of the core. In addition, since control-flow instructions are handled by the arithmetic pipelines, these would need a bounds check anyway. Implementing the capability arithmetic per arithmetic pipeline does mean paying for its area multiple times: dynamic instruction measurements may indicate that a single capability pipeline could be sufficient depending on common capability operation density in compiled code. To avoid any capability overhead in the unmodified pipelines would likely require that these pipelines not be capable of performing jumps[1]. While investigating alternative approaches to the capability arithmetic pipeline is future work, the design has been carried out in a modular way to enable this research.

Unlike Piccolo and Flute, which decode in a switch statement in the execute stage, Toooba has a distinct decode stage. This allows the capability instructions to be properly decoded into more direct control signals for the capability module within the ALU. Operations are grouped into options for common functional units, with register arguments reordered to maximise sharing without incurring multiplex overhead in the ALU. Capability arithmetic itself is carried out as in Piccolo and Flute. The partial decompression from `CapReg` to `CapPipe` discussed in Section 3.4.1 occurs in the register read stage of the arithmetic pipeline.

Exception conditions, for example checking for modifications to sealed capabilities or violation of capability permissions, are mostly checked in parallel with the ALU, again similar to Piccolo and Flute. As before, the expensive bounds check is handled differently, since it depends on the base and top values that are produced late within the ALU. The execute stage produces signals that are fed into the bounds check in the finish stage of the ALU pipeline. In all cases, the exceptions do not affect the produced value, instead setting an exception field in the reorder buffer entry that is checked in the commit stage. As discussed in Section 5.3.6, this initial approach opened the door to speculative execution attacks, motivating an alternative architecture and design.

5.3.2 Memory pipeline

The memory pipeline must be extended with modest additional instructions, again requiring additional signals emitted from decode. These instructions mostly extend existing functionality, for example adding byte-granular and capability atomic operations, and adding additional exception cases to the data TLB. The width of the interface to the Load/Store Queue must be extended to full capability-width. In addition, the register read stage of the pipeline must read DDC in case it is used to authorise and offset legacy memory operations. Address calculation is extended to work with capabilities: addition of

[1] This could be a worthwhile tradeoff independent of capabilities.

Location	Purpose	Read/Write
Fetch	Fetch the next instruction	Read
Fetch	Update to next speculated instruction	Write
Decode	Determine capability encoding mode	Read
Execute	Jumps and `auipc(c)`	Read
Execute	Fix misprediction	Write
Commit	Update `mepc(c)` on exception	Read
Commit	Install `mtvec`/`mtcc` on exception	Write

Figure 5.2: Usage of the PCC within Toooba.

the immediate to the address register in the baseline becomes an increase of the authorising capability register's offset by the immediate in the CHERI version.

In addition, the memory pipeline requires its own checks to ensure memory accesses only occur when authorised by an in-bounds capability with relevant permission bits set. This requires checks, including the bounds check, to be carried out before accesses can have side-effects on the memory subsystem. As in the arithmetic pipeline, the simple checks are carried out in execute and the more complex bounds check is prepared in execute to be carried out in the finish stage. Fortunately, the pipeline already has a commit call to the Load/Store Queue in the finish stage to support TLB exceptions. This means the additional interface that was required for Piccolo and Flute (described in Section 3.4.2) is not required here.

5.3.3 PCC implementation

CHERI extends the PC with capability metadata, forming the PCC, to protect control-flow and enable code compartmentalisation.

The tracking of this metadata posed some interesting questions even for the scalar processors (see Section 3.5.1), where there are already several notions of the PC: one for each of the instructions in the pipeline, the architectural PC to be taken if the processor is interrupted, and predicted PCs passed into the instruction cache to be fetched. This complexity is compounded in Toooba, presenting many options to store metadata.

Figure 5.2 gives a summary of the various PCC interactions within the processor. The baseline Toooba has several copies of the PC within the front-end, including within prediction structures. The ALU pipelines are the only ones requiring the PC: for the `auipc` instruction and to link and calculate the new PC after a jump or branch. The commit stage also needs the PC, as it must be installed in `mepc` in the event the instruction traps. This implies that the PC must be preserved in the reorder buffer. Each occurrence

of the PC needs to be considered when adding CHERI to reflect the fact that it has architecturally become a capability.

When extending the PC with capability metadata to produce PCC, one extreme is to include the metadata in all cases. This presents the simplest option, as it requires no change of structure from the unmodified processor. However, this may have significant area impact due to scaling up of all of the structures. The other extreme is to have only one set of PCC bounds present in the processor, with a pipeline flush whenever PCC bounds change. This may significantly reduce the area overhead, at the expense of performance in code that regularly changes PCC bounds. Current CHERI code generation does not tightly bound the PCC within functions, since globals are still accessed relative to the current PCC. Therefore, this approach is unlikely to carry a significant performance penalty in the benchmarks shown in this thesis. However, work on compartmentalisation is likely to experiment with tighter bounding of PCC, so we opt to augment all instances of the PC with bounds in the initial implementation to enable this research.

A particular concern for area growth is the reorder buffer: this is a large structure, comparable to the physical register file itself. In the baseline processor, for the configuration shown in Figure 6.1, there are 64 entries, each consisting of 225 bits. Most notably, this includes: 64 bits for the PC, 64 bits for another union field, and 32 bits for the original encoded instruction. The union field can contain the predicted PC, the value to write to a CSR, or the virtual address of a store. The remaining bits are all for various tracking metadata. The current implementation of CHERI adds 138 bits to each entry: 65 bits of capability metadata and 65 bits for the other union field, as well as a few bits to track SCRs and CHERI exception codes.

A possible compromise is to allow a limited number of distinct PCCs to exist in the processor, storing the full metadata in a lookaside table and only storing an index into each table and the PC as an offset in the other structures in the processor. The upper bits of the PC address could also be moved to this structure, making this a potential saving even without CHERI. Once again, performing a study to determine an appropriate tradeoff between area and performance is future work, depending on software elaboration of compartmentalisation models.

5.3.4 Special Capability Register implementation

The baseline Toooba processor takes a very conservative approach to dealing with CSR writes. This is because CSRs can have wide-reaching effects on other instructions: for example a change to `frm` can change the rounding mode of all floating-point instructions, and a change to `mtvec` would change the address to take on any exception. The designers therefore choose to allow implicit CSR reads freely within the pipeline, and stall the pipeline on any architectural access to a CSR as it reaches rename, waiting for all outstanding instructions to complete before issuing it directly to commit.

CHERI adds SCRs that serve a similar purpose to CSRs but hold capabilities. These are initially treated akin to CSRs since the assumption that they are infrequently modified holds for current capability code. This allows them to be freely read within the processor at the expense of a pipeline flush on write. However, one particularly contentious SCR is DDC, which is implicitly used as the capability authorising and offsetting legacy RISC-V memory operations. Therefore, similar to some other CSRs, the effects of writing DDC are far-reaching for other, seemingly independent, instructions. DDC could potentially be used for compartmentalisation to protect non-CHERI-aware libraries from each other, so it is not yet known how often DDC will be written. If the performance cost of flushing the pipeline on every DDC write is too high, an alternative implementation is to add it, and possibly other SCRs, as renamed registers. This would mean extending the physical register file modestly, as its size is determined by the number of architectural registers plus the maximum number of in-flight instructions to eliminate structural hazards due to register contention. The size of register indices throughout the pipeline, for example in forwarding paths, would also need to grow. However, the main cost would be additional physical register file read ports to distribute the value to the memory pipeline. As with other optimisations, this approach could also benefit the baseline core, as commonly modified CSRs could also be forwarded with negligible additional cost.

5.3.5 Extending structures

In addition to the reorder buffer and pipelines already mentioned, other structures within the processor pose a tradeoff when extending them with capability metadata, most crucially the physical register file and Load/Store Queue. Figure 5.3 gives an overall summary of the key extended structures.

5.3.5.1 Physical register file

Superscalar processors have large physical register files, allowing registers to be renamed to avoid false dependencies. We initially extend every register to capability-width. As discussed in Section 3.4.1, this grows registers from 64 bits to 151 bits due to partial decompression. Metadata is decompressed from a `CapMem` to a `CapReg` on the path into the register file from memory on load, and vice versa when storing a capability. Integers are stored in the register file with null capability metadata, presenting significant redundancy depending on the mix between capabilities and integers. On read, the `CapReg` is further decompressed to a `CapPipe` in the register read stage of the relevant pipeline. The merged register file makes implementation particularly convenient: interactions with registers, including forwarding, can simply reuse the existing control logic but with a different datatype.

The area impact from extending every physical register is large: our configuration has 128 physical registers. Limiting the physical register usage by only extending some registers to

Structure	Change	Potential optimisation	Optimisation impact
Pipelines	Add capability manipulation logic	Extend only a single pipeline	Throughput of dense capability operations
Register file	Support capability metadata	Extend only a subset of registers	Throughput depending on live integer/capability mix
Load-store queue	Widen data width	Extend only a subset of entries	Throughput of dense capability stores
Reorder buffer	Add PCC metadata	Restrict live PCCs with different bounds	Performance of compartmentalised code

Figure 5.3: Summary of the key Toooba structures extended to support capabilities, and potential optimisations to reduce the area impact.

full capability width would reduce this overhead at the expense of performance, stalling whenever a capability register is needed but unavailable. Therefore, future work would investigate how many physical registers are required for capabilities for common generated capability code. However, Toooba is designed to avoid any structural hazards due to scarcity of physical registers, so this approach would also require some extra complexity. An alternative to having a limited subset of capability registers is to allocate a sidecar register when non-null metadata is required. This is the approach taken by *Watchdog* [86] (discussed in Section 2.3.1), although the authors used a pipeline simulator, meaning they did not validate the approach in microarchitecture. This would require significant changes to register rename and would once again require a mechanism to stall when insufficient sidecar registers were available. One possible variation of this approach would be to allow registers to hold either integers or metadata. This would make capabilities consume two physical registers, trading further complexity for less wasted area.

5.3.5.2 Load/Store Queue

Application-class processors also tend to feature a Load/Store Queue between the pipeline and caches, allowing bursts in cache activity to be absorbed, and out-of-order cache responses to be supported without violating memory model ordering constraints. Such structures contain records of data transferred to the memory subsystem, often at the maximum granularity of atomic access (the maximum width that can be loaded or stored per instruction). CHERI RISC-V doubles this width by allowing 128-bit capabilities to be loaded and stored in a single instruction. The most natural way to accommodate this is

to widen the entries in the Load/Store Queue, but this may lead to a large area increase. This is the approach we take, keeping the number of Load/Store Queue entries the same and doubling the size of the entries for stores, which must hold the stored data. For our configuration, there are only 14 Load/Store Queue entries for stores, so this overhead is less significant than the other structures discussed.

Alternatively, one could support this in different ways, such as dividing the capability accesses into their address and metadata bits, and being careful to ensure all accesses are atomic between these. This would reduce the area overhead, at the cost of performance if the Load/Store Queue fills up with the additional entries. This is very similar to the tradeoff presented by the physical register file, except in this case the performance penalty scales with the maximum number of outstanding capability stores, rather than the number of capabilities live in the register file.

5.3.6 Safe speculation

The recent Meltdown [80] and Spectre [71] attacks have shown that processors can be attacked using cache side channels triggered by code that has speculatively bypassed security checks. Following these initial attacks, Canella et al. have performed a systematic overview of transient execution attacks and defences [25]. Capabilities can interact with these attacks in two very different ways. Capability hardware may make it easier to mitigate speculative execution attacks efficiently, since bounds checks can be carried out even on speculative paths. Work on this is proceeding within the CHERI group [130]. Conversely, speculative attacks may be used to bypass the capability security model, for example using a capability out of its bounds if the bounds check is delayed. This section discusses the second interaction in the context of the CHERI Toooba implementation.

Work by Fuchs, in collaboration with the rest of the CHERI team, identified a significant security vulnerability in the initial CHERI Toooba implementation [48]. Fuchs discovered Meltdown Capability Forgery: an attack that allows capabilities to be forged in speculation and used to signal cache side channels. As discussed in Section 5.3.1, the initial CHERI adaptation of Toooba determined the exception conditions in parallel with executing the instruction, with the exception flag not being checked until commit. This gives a window of opportunity for the capability to be read from the physical register file or forwarded until the instruction is committed.

More explicitly, the attack proceeds as shown in Figure 5.4. Microarchitecturally, the first load is committed, but the branch cannot be dispatched into the ALU pipeline until the value is read from DRAM. This clogs the in-order commit stage. The `CSetBoundsImm` is dispatched out-of-order to the ALU, where it marks in the reorder buffer that an exception is required on commit, but produces the capability with the requested bounds, sending it on forwarding paths and writing it to the physical register file. The reorder buffer entry stays queued up behind the branch, giving the rest of the code time to run before

```
# Perform a load that will miss the cache
cld             t0, 0(ca1)
 # Branch on loaded value, trained not taken
bnez            t0, exit

# Branch will be architecturally taken
# The rest runs only in speculation:

# Extend the bounds
csetboundsimm ct1, ca0, 24 # Would trap if committed
# Load the secret
cld             t2, 16(ct1)
# Load a cacheline dependent on the secret
cincoffset      ct3, ca2, t2
cld             t4, 0(ct3)
```

Figure 5.4: A CHERI RISC-V program that violates capability guarantees on Toooba using a speculative side channel. Reproduced based on work by Fuchs [48]. `ca0` contains a capability pointing at its base, with bounds of potentially just a single byte. An attacker knows that a secret is stored 16 bytes after the base of the capability, but the secret is beyond its top. While 16 is used for illustration, there is no limit on how far out-of-bounds this attack can reach, so long as the secret is resident in the cache. `ca1` contains a capability whose address is known to be not resident in the cache. `ca2` contains a capability to memory also known not to be resident in the cache: after this attack is run, this memory can be probed to determine which line was fetched, extracting the secret.

the exception is committed. This includes loading the secret: even though the memory pipeline does check the bounds and tag, even in speculation, the capability that has been presented to it is perfectly valid. Finally, the original load returns and the branch resolves, squashing all the speculative instructions, but leaving behind the effect on the caches.

Once the mechanism for Meltdown Capability Forgery was identified, it became possible to audit the Toooba core comprehensively for potential similar vulnerabilities. Since the CHERI error conditions were all specified as raising exceptions, all possibilities for transiently monotonicity-violating operations can be identified from the source code. In particular, any instance of the ALU pipeline setting the exception field of the instruction's reorder buffer entry could potentially imply a vulnerability. This allowed other vulnerabilities related to the initial proof-of-concept test case exploiting the `CSetBounds` instruction to be predicted. This was done in parallel with work by Fuchs et al. to generate attacking code automatically using TestRIG [49], validating each others' findings.

The following vulnerabilities existed in the initial implementation, all exploitable using the same framework as shown in Figure 5.4. More detailed descriptions of the purposes and intended semantics of the instructions are given in Appendix A.

- `CSetBounds` changes the bounds of the input capability, raising an exception if the

new bounds are not a subset of the original. Any length can be given, and will be honoured (and possibly rounded up) in the forwarded value, allowing any larger addresses to be transiently accessed. Due to compression, the capability pointer, i.e. the requested new base, can only stray so far below the original base, limiting access below the capability, although rounding down of the base with large lengths and repeated `CSetBounds` instructions can likely be leveraged to access any address.

- `CBuildCap` performs various checks to ensure a capability is a subset of an authorising capability before setting its tag, and otherwise raising an exception. By exploiting the fact that the forwarded value is tagged, even if the input capability is not a subset of the authorising capability, an attacker can produce any (unsealed) capability based on any bits they choose to craft. Thus, this turns out to be a very convenient vulnerability for an attacker.

- `CUnseal` removes the sealing type of an otherwise immutable, sealed capability provided the operation is authorised by a suitable unsealing capability. Once again, any errors are signalled only using an exception, so capabilities can be unsealed in speculation without a valid authorising capability to allow them to be dereferenced, despite this normally being forbidden.

- Similarly, `CSeal` seals a capability, making it immutable after checking the operation is permitted. Again, an attacker could use Meltdown Capability Forgery to create a sealed capability illegally in speculation, though the utility of this is questionable.

- Any instruction that modifies a capability value raises an exception if the value was sealed. This can be bypassed in speculation, although this also has limited utility to an attacker given we have already seen that they can freely unseal capabilities in speculation.

- Jumps also signal capability-related errors as exceptions, so may be used to violate capability guarantees. Jumps can only change capabilities in limited ways: changing the offset (with atomic tag clear if unrepresentable) and possibly unsealing them (with associated non-monotonic compartment change). This prevents jumps from being used to forge capabilities. Execution outside of the architecturally correct behaviour is fundamental to branch-prediction and speculative execution, so a full fix of control-flow based attacks is future work, possibly orthogonal to capabilities.

Notably, certain instructions, such as `CAndPerm`, were not vulnerable due to not being able to express a non-monotonic operation. `CSetAddr` and other instructions that change the pointer of a capability can change the interpretation of the bounds by taking the pointer outside of the representable region. This cannot lead to an attack as the architecture specifies to clear the tag in such cases rather than raising an exception. This prevents monotonicity being violated by the forwarded values, even in speculation.

In the context of Hypothesis H.2, these attacks violate the capability model, and would potentially limit CHERI's applicability to superscalar cores if not resolvable. Fortunately, the implementation of Toooba can be fixed to be resilient against Meltdown Capability Forgery.

Fixing the vulnerabilities posed varying levels of difficulty. For the checks that are performed in the execute stage, the tag of the forwarded value can be gated based on whether an exception condition is detected. This prevents the resulting capabilities from being used by the memory pipeline. Minimal additional hardware is incurred: simply an additional `AND` gate on the tag of the value produced.

However, this does not fix the most egregious of the vulnerabilities. As discussed in Section 5.3.1, the bounds check used by `CSetBounds` and `CBuildCap`, among others, is performed during the finish stage of the ALU pipeline. This means that the value would be forwarded before it is even known whether the operation should trigger an exception. One option is to delay the forwarding in these cases, essentially making these instructions multi-cycle. Due to the nature of forwarding and register scoreboarding in Toooba, this would mean the destination register could not be marked as available in dispatch as then it may not be ready in time for other pipelines that depend on it. Therefore, the scoreboard would either have to delay marking the register as ready until the end of the pipeline, or add the option to signal it as ready from the next stage (register read). While this may be acceptable for some instructions, `CSetBounds` is relatively common, so delaying its result may incur a significant performance penalty. I instead attempt to break into the compressed capability encoding to accelerate determining whether the results will be in-bounds so that this can be indicated in the forwarded value. Since these changes remove the motivation for an architecture that raises exceptions on monotonicity violations, I implement this as part of switching to tag-clearing error semantics, discussed in Section 5.4.1.

5.4 Avoiding exceptions

For CHERI to provide security, it must signal errors when monotonicity violations are detected, for example when an attempt is made to widen the bounds of a capability. This is discussed in more detail in Section 2.4.1. These can be signalled immediately, via an exception, or postponed by clearing the integrity tag of the returned capability. Raising an exception as soon as possible after the error aids debugging and arguably security as the system spends less time in an unexpected state. Microarchitecturally, it is tempting to consider that an exception-based approach is simpler since the datapath can naïvely perform the intended operation while the error conditions are checked independently and only influence the control path. However, Section 5.3.6 highlights that performing this optimisation safely is not as easy as it might appear. To avoid bypassing capability

security, the processor must avoid forwarding the results of exception-triggering operations to operations that could affect cache state before the exception flushes the pipeline. Tag clearing may in fact simplify microarchitecture by limiting the set of instructions that can raise exceptions, especially if certain pipelines are otherwise unable to do so. In addition, tag-clearing error behaviours provide more opportunity for compiler optimisation, as more operations can be reordered safely if they have no possibility of causing an exception. Furthermore, the possibility of instructions raising exceptions, such as when handling capabilities passed as an argument, can be a large burden for code that cannot tolerate exceptions, for instance in exception handlers themselves.

CHERI MIPS and the initial CHERI RISC-V specification chose to deliver exceptions on monotonicity-violating instructions, primarily due to the importance of debugging in initial CHERI work. However, due to the reasons discussed above, we made the decision to switch CHERI RISC-V to tag-clearing semantics. This also matches Arm's design choice for Morello of avoiding exceptions in favour of tag clearing where possible. Operations that use capability privilege—memory accesses and jumps—still need to trigger an exception. However, register-to-register operations can have their exception conditions replaced with tag clearing. This presents a timing difficulty for hardware: all the checks must be performed before the value can be forwarded. However, it should be noted that this work is required in any case to avoid capability forgery in speculation.

I made the required changes to the architecture to avoid unnecessary exceptions, including the CHERI architecture document and Sail CHERI RISC-V model, as well as performing the microarchitectural changes for Toooba, Piccolo, and Flute.

Most of the removed CHERI exceptions can be split into the following categories:

Tag assertion exceptions Exceptions not required for security, but where the operation being performed only makes sense on a tagged capability. For example, the original CHERI RISC-V specification threw an exception when modifying the permissions of an untagged capability. These exceptions aid debugging, as a cleared tag can be found earlier than waiting for the capability to be dereferenced. However, they can be particularly troublesome for software that needs to avoid exceptions. In this case, the exception condition can safely be completely removed architecturally. If the programmer prefers to have the exception for debugging purposes, the compiler can emit a tag assertion before any instance of the instructions, preserving the old behaviour.

Sealed modification exceptions These are raised when an attempt is made to modify a sealed capability. The sealing mechanism creates immutable capabilities that can be used for limited non-monotonic transitions between security compartments, or Sentry capabilities that can only be jumped to unmodified, impeding control-flow based attacks. Attempting to modify a sealed capability formerly produced an exception, but with tag-clearing semantics the resulting capability is instead untagged. This is

a relatively easy change to make in hardware: the tag bit is gated based on the type of the input capabilities in parallel with the operations.

Bounds modification exceptions Operations that change bounds must be monotonic to prevent privilege escalation. Due to the compressed capability format, these are some of the trickier cases to detect in hardware, lengthening critical paths. As such, I add logic to compute efficiently whether the result of a `CSetBounds` instruction will be in-bounds in parallel with carrying out the operation. Section 5.4.1 goes into more detail on this change.

Bounds checking instructions Other than for restricting bounds, several capability modification instructions must perform bounds checks. Sealing and unsealing operations must ensure the authorising type capability is in-bounds. These checks are relatively cheap as determining whether a capability is within its own bounds can be determined simply (without any expensive shifts) from the mantissa bits of the base and top alongside the corresponding address bits that are already extracted in the `CapReg` format. `CBuildCap` is a trickier case as it must perform a full subset check with minimal constraints on the input. This instruction is designed to be used in a very limited set of circumstances, so it is possible it can tolerate an additional cycle of latency.

5.4.1 Fast bounds check

One of the difficulties with moving from exceptions to tag-clearing is that the legality of `CSetBounds` must be determined before the result is forwarded. This would need to be solved even without any architectural changes to avoid Meltdown Capability Forgery [48]. Achieving high frequencies requires either a stall before the value is forwarded as the new tag is determined, or faster logic to allow the new tag to be calculated in a single cycle.

While the general-purpose bounds check cannot be performed dependent on the result of the ALU, the particular constraints of `CSetBounds` allow specific optimisations. In particular, the new base is determined by the capability cursor of the input capability. The `CapReg` partially-decoded format present in registers already contains a copy of the address bits shifted by the exponent. This means a mantissa-width comparison is sufficient to determine whether the requested base is above the existing base and thus legal. The new top is trickier, as it is specified as the new length of the intended capability. By shifting the requested length down by the capability's initial exponent, any bits above the mantissa width immediately disqualify the length from being legal. The base bits plus this shifted increment can be compared against the top bits to determine whether the requested top is in-bounds, with the carry of the lower bits of the increment plus the address being used in case of equality.

This gives an algorithm to determine the monotonicity of the `CSetBounds` that may allow single-cycle implementation. The naïve approach—computing the full bounds then performing the comparisons—requires a full 64-bit shift followed by a 64-bit subtract. The new approach also requires a shift, but this is followed only with mantissa-width arithmetic (14 bits rather than 64). This should significantly reduce the critical path length. Due to Toooba's modest frequency (see Section 6.3), both approaches pass timing when instantiated in its ALU. Evaluating the effect of this algorithm on the critical path length concretely would therefore require an alternative synthesis platform. Performing this evaluation is future work. However, I have checked the correctness of the algorithm by testing equivalence with the naïve approach. This was done using *BlueCheck* [89], and by exhaustive search on a cut-down 32-bit capability format.

5.5 Software and verification

As with the microcontroller implementations, TestRIG was used to check correctness of instructions as they were added, significantly accelerating implementation and verification. The generality of the TestRIG framework was very useful: by augmenting Toooba with the RVFI-DII interface, all of the templates written to verify the CHERI instructions for Piccolo and Flute could be reused for Toooba. The RISC-V test suite [21] was also run continuously as the processor was developed, although this does not test the new capability operations.

As well as running Cheri-FreeRTOS like the microcontrollers, Toooba runs the application-class CheriBSD OS. This allows the FPGA platforms to be used for OS evaluation, and run as (somewhat slow) interactive machines, including support for SSH and common FreeBSD utilities, as well as allowing `cheribsdtest` to be run. This also allows the SPEC benchmarks to be run, enabling the evaluation in Chapter 6.

As the baseline processor supported instantiation with multiple cores, Toooba presented the first opportunity for multicore CHERI RISC-V: this is fully supported and CHERI Toooba can boot CheriBSD with multiple cores. One difficulty with this was augmenting the Bluespec-designed debug module to support multiple Hardware Threads (Harts).

5.6 Future work

This chapter has highlighted several tradeoffs between area and performance, summarised in Figure 5.3. Future work would evaluate these. This would involve performing the area-saving optimisations on FPGA to determine the microarchitectural complexity and impact on frequency. Separate work could analyse capability software patterns to determine the performance penalties from different points in the space.

The changes to the architecture proposed in Section 5.4 imply microarchitectural changes that may cause timing difficulties. Attempts to mitigate these, for example the fast bounds check for `CSetBounds` in Section 5.4.1, would require a higher-frequency FPGA implementation or ASIC synthesis tools to validate fully.

The protections against speculative execution attacks described in Section 5.3.6 only protect against the Meltdown Capability Forgery attack described by Fuchs [48]. Further work would investigate robustness against speculative attacks more broadly, such as Spectre [71] attacks.

As with the microcontroller implementations, CHERI Toooba currently lacks support for the `CClearRegs` instructions for accelerating compartmentalisation.

5.7 Summary

This chapter has confirmed the qualitative component of Hypothesis H.2 by showing that there are no fundamental obstacles to implementing CHERI for application-class processors. An implementation of CHERI is presented that is sufficient to boot capability operating systems. Tradeoff spaces in implementation of key structures are identified, highlighting the choice between area and performance overheads for the pipelines, physical register file, load-store queue, and reorder buffer. While naïve implementations may be vulnerable to Meltdown-like speculative execution attacks, it is possible to produce an implementation robust against such attacks. The architecture itself can be changed to tag-clearing rather than exception semantics, encouraging such implementations while deriving further compiler and software benefits.

Chapter 6

CHERI application-class processor evaluation

The Toooba CHERI implementation is evaluated in a similar manner to the microcontrollers in Chapter 4. FPGA tools are used to estimate timing and area, and the SPEC benchmarks are used to measure impact on IPC and performance. The microcontroller benchmarks MiBench and CoreMark are also run to enable comparison with Piccolo and Flute.

The evaluation performed is again of FPGA frequency, area, power, performance and security on a VCU-118 FPGA within the BESSPIN SoC. Once again, the software and compiler work relied on is not a contribution of this thesis. The configuration of the processor used throughout is shown in Figure 6.1.

The evaluation enables answers to the quantitative component of Hypothesis H.2. In addition, I investigate whether the trends for how the overheads scale with core size indicated by Chapter 4 continue to the significantly larger microarchitecture. This addresses Hypothesis H.3 more fully.

6.1 Baseline core information

Figure 6.2 shows performance metrics for the Toooba baseline core. Once again, the cache latencies appear realistic, while the DRAM latency is dramatically reduced compared to an ASIC environment.

The CoreMarks/MHz score is competitive, significantly exceeding Rocket (2.94) and CVA6 (2.08) but behind BOOM (6.25) according to Dörflinger et al. [40]. As with microcontrollers, care should be taken drawing direct comparisons as results may depend on evaluation environment and core parameter choices.

	Toooba
Frequency	25 MHz
Core count	Dual-core
`xlen`	64
Supported extensions	A, C, D, F, M
Supported privilege modes	M, S, U
L1 TLB	32 entries fully associative
L2 TLB	1024 entries four-way associative
Pipeline stages	11
Superscalar width	Two-way
Reorder buffer	64 entries
Physical registers	128
Max. outstanding branches	12
Load/Store Queue	24 load entries, 14 store entries
RAS	16 entries
BTB	1024 entries two-way associative
Data L1 cache	32 KiB eight-way associative
Instruction L1 cache	32 KiB eight-way associative
L2 cache	1 MiB 16-way associative
Tag cache (CHERI only)	128 KiB four-way associative

Figure 6.1: Benchmarking configuration for the Toooba processor.

	Toooba
CoreMarks/MHz	4.6
L1 hit load-to-use cycles	5
L1 miss penalty cycles (L2 hit)	22
DRAM latency cycles	16

Figure 6.2: Metrics for the baseline Toooba core in the evaluation SoC. Cache latencies were determined in simulation. DRAM latency was determined based on performance counters in the `adpcm_decode` MiBench benchmark. TLB accesses are performed in parallel on hit, causing no additional delay.

6.2 Area

This section investigates the area impact of the initial augmentation of Toooba with capability support. Little effort has yet been made to optimise the area overhead. In addition, as with the Piccolo and Flute processors, the CHERI changes have been implemented so as to increase the size of structures liberally. In other words, we choose increased area over the possibility of new performance bottlenecks. Further investigation of this tradeoff space is future work enabled by an initial implementation and evaluation.

The LUT and FF usage of various structures within Toooba is shown in Figure 6.3. See Section 4.2 for discussion of these two metrics. For Toooba, we use the "`Congestion_-`

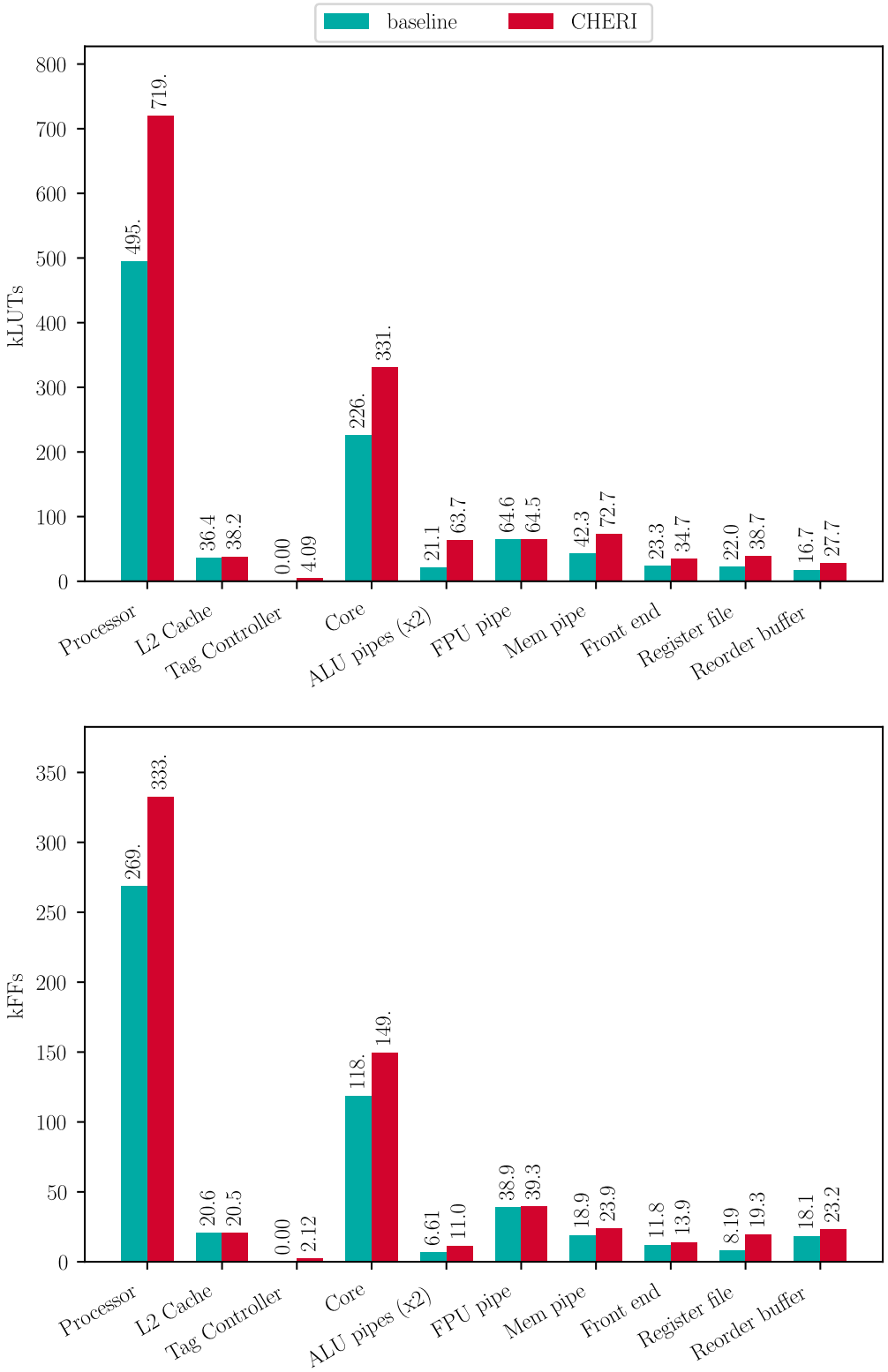

Figure 6.3: Area overhead of CHERI for a dual-core Toooba processor. The core, tag controller, and L2 cache are contained within the processor. All other components are in turn contained within the core. Since the two cores are identical, only one is shown, including for its constituent components.

SpreadLogic_High" optimisation strategy to avoid synthesis failures due to congestion. This also obviates the need for "keep_hierarchy" directives, which otherwise do not suit Toooba due to its different module structure.

Processor Excluding hardware for interconnect with DMA and other peripherals, CHERI has a 45% LUT overhead, and a 24% FF overhead. These are very similar to the overheads found in Flute, although the FF overhead is significantly lower. The LUT overhead significantly overestimates the die-area overhead that would be expected from a silicon synthesis as the caches, which are not significantly enlarged by CHERI, are implemented as BRAMs on FPGA, so do not show in the baseline or CHERI variants.

L2 cache The L2 cache is not significantly altered for CHERI, so does not see much area overhead. Only tag support needs to be added.

Tag controller While the tag controller has a very similar size here as for the microcontrollers—4,091 LUTs—it is dwarfed by the other components of the processor, contributing only 2% of the CHERI LUT overhead. This is in stark contrast to the microcontrollers, where it was a major contributor to the area overhead.

Core Zooming in to a single core, and excluding components shared between the cores—most notably the PLIC and debug module—gives a core area overhead of 47% LUTs and 26% FFs.

ALU pipelines The primary source of LUT overhead is the ALU pipelines. Assuming symmetry between them, each grows from 10,561 to 31,870 LUTs: a 202% increase. This alone constitutes 38% of the overall CHERI LUT overhead. The primary causes of overhead here are the capability manipulations themselves, including the capability decompression, as well as the bounds checking and other checks. The wider path to and from the register file (including forwarding paths) is also a likely contributor. This observation further motivates future work to examine whether a single capability-enabled pipeline is sufficient.

FPU pipeline The FPU pipeline is not extended by CHERI so sees no real LUT or FF growth, helping to reduce overall fractional overhead, especially since this pipeline is a significant fraction of the baseline area.

Memory pipeline The memory pipeline sees a LUT overhead of 72%: smaller than the ALU pipelines, but still significant. This includes the Load/Store Queue and L1 data cache. The area increase is likely attributable to the liberal widening of these structures, as well as the widening of the incoming data from the register file and forwarding paths. The bounds check and additional atomic operations also contribute.

Front end The front end of the processor (which includes the L1 instruction cache and branch prediction) grows by 49%. This is so high because of the multiple copies of PC, all of which are extended to include capability metadata. In addition, the front end also grows checks to ensure that instruction accesses are within bounds.

Register file Unlike the microcontrollers, Toooba's physical register file synthesises to FFs. Note that the number of FFs exactly corresponds to what would be expected based on the baseline and CHERI register types. Each has 128 registers: for the baseline these are 64-bit integers, giving 8,192 bits, and for CHERI these are 150-bit (plus one bit tag) `CapReg` instances, giving 19,328 bits in total. This 136% increase could be mitigated by limiting the number of physical registers that can hold capabilities, as discussed in Chapter 5.

Reorder buffer The reorder buffer grows significantly: 66% LUTs and 28% FFs. This can be explained by the additional fields in each entry to track capability information, most notably the PCC metadata.

This analysis of the area overheads allows insight into the potential savings of the optimisations in Figure 5.3. Only extending a single ALU pipeline with capability logic could be expected to save 21,309 LUTs and 2,217 FFs per core. Limiting the number of capability physical registers could save an upper bound of 14,496 FFs per core. Avoiding widening the Load/Store Queue entries could save an upper bound of 30,400 LUTs and 4,973 FFs per core. This may be a very loose upper bound, as it assumes all of the memory pipeline overhead comes from the Load/Store Queue. Finally, avoiding extending the PCC and other fields in the reorder buffer could save up to 10,990 LUTs and 5,101 FFs per core. This suggests prioritisation for applying the optimisations, depending on the tradeoff between LUT and FF overhead and the possible performance penalties. The lower bound of the overall processor overheads after applying all of these optimisations is approximately a 20% LUT and 4% FF overhead. However, it should be stressed that these are lower bounds, assuming very aggressive savings.

The fractional area overheads are smaller than Piccolo, but similar to Flute, at least in LUTs. This contradicts Hypothesis H.3: the much larger Toooba core sees a similar fractional logic overhead to the smaller Flute. The reason for this is that the structures added to support out-of-order execution and superscalarity all need augmentation with metadata, while little is added that is unchanged by CHERI. However, the area-saving optimisations discussed above could provide quick wins to bring the overhead down for Toooba, without changes to the architecture, while Flute has fewer such opportunities.

F_{max} (MHz)	Design target	Baseline	CHERI
Toooba	25.0	43.8	38.7

Figure 6.4: Maximum frequency of the Toooba processor synthesised for the VCU-118. The target frequency is that provided by Bluespec Inc. for the baseline processor: no optimisation was attempted beyond this. For timing only, Toooba is synthesised single-core to avoid congestion errors from Vivado and keep synthesis times reasonable.

6.3 Frequency

The baseline maximum frequency is much lower than that of the microcontrollers. The design is not optimised for FPGA: for example, the register file synthesises as FFs rather than BRAMs. Zhang et al. report that RiscyOO achieves 40 MHz on an AWS F1 FPGA, but up to 1.1 GHz on a 32 nm ASIC flow [145]. This limits the extent to which the frequency impact of the CHERI changes can be evaluated.

As discussed in Section 4.3, most of the CHERI changes apply in isolation to select parts of the processor, where they can operate in parallel, such as the bounds check. This means they are unlikely to cause timing issues in Toooba, especially as similar operations passed timing within Flute at 100 MHz. However, increases in congestion due to area overheads can increase critical path lengths.

Figure 6.4 shows the F_{max} for Toooba on the VCU-118, again attempting timing closure at the unachievable 150 MHz. The CHERI processor has a slightly lengthened critical path. However, since it comfortably met the target timing, no optimisation work was performed, and there is no reason to believe that the frequency worsening is fundamental.

The critical path in both the baseline and CHERI cases takes the loaded value from the data cache, passes it through the Load/Store Queue, and seems to interact with register rename and a write to the register file, possibly via a control dependency. Once again, substantial further work would be required to optimise the baseline processor to the point that the CHERI impact for high-frequency cores could be investigated more thoroughly, but it can be seen that CHERI has not significantly altered the critical path as it stands.

6.4 Performance

To measure application-class performance, the SPEC benchmarks are run atop CheriBSD. See Appendix B for a brief description of each benchmark. Although the synthesised design is dual-core, the benchmarks are single-threaded. They are pinned to a single core to minimise benchmarking noise. CheriBSD running on the other core introduces some variability, especially in the shared L2 cache, so this is commented on in the results.

Benchmarks are run in the `train` configuration, compiled with the (capability-augmented) LLVM compiler with `-O3`.

6.4.1 Legacy performance

We first examine the performance of the modified hardware without using the CHERI protections in software. As in the microcontroller case, this should be cycle-identical within the core, with only memory subsystem changes affecting performance. Only the tag controller is fundamental, since it interposes on memory accesses, adding latency in the event of a miss. Once again, the difference is small, and is in fact negative in some cases. This may be caused by changes in AXI interconnect latency characteristics. The average difference is less than 1%. This confirms that capability hardware does not incur performance overhead unless the security features are used.

6.4.2 Capability performance

We next compare the performance of the benchmarks on the same CHERI-enabled hardware with and without CHERI enabled in the benchmark software. Note that the overheads of pure capability software are dependent on compiler optimizations, such as deciding when bounds subsetting can be elided based on static analysis. The SPEC pure capability benchmarks are compiled without capability-aware compressed instructions, so some instruction cache overhead is expected. The results also depend on the effectiveness of the microarchitecture. Overheads are expected from increased instruction cache pressure from the extra capability instructions, as well as the execution of these instructions themselves. Data cache pressure from the storing of metadata alongside pointers will also contribute to cycle overheads. As the CHERI Toooba implementation liberally extended all structures with metadata, most instructions within the CHERI core should perform the same as the baseline, with the added capability instructions completing and forwarding their results in a single cycle.

The results are shown in Figure 6.5: mean instruction and cycle overheads match at 9%. I now investigate the overheads, with analysis of potential causes based on the output of performance counters. Note that, since Toooba is out-of-order and superscalar, accounting of cycles by performance counters is only approximate. For example, while the number of cycles spent by caches waiting for load misses is recorded, the core can proceed with other non-dependent instructions while the corresponding load resolves. Appendix B describes the benchmarks, including whether they are expected to be pointer-heavy, which is a significant factor in determining the capability overhead.

The `bzip2`, `h264ref`, `astar`, and `hmmer` benchmarks all see the smallest instruction overheads and even lower cycle overheads. These were all expected to be relatively pointer-light benchmarks. This is confirmed by the results: between them, on average only 17%

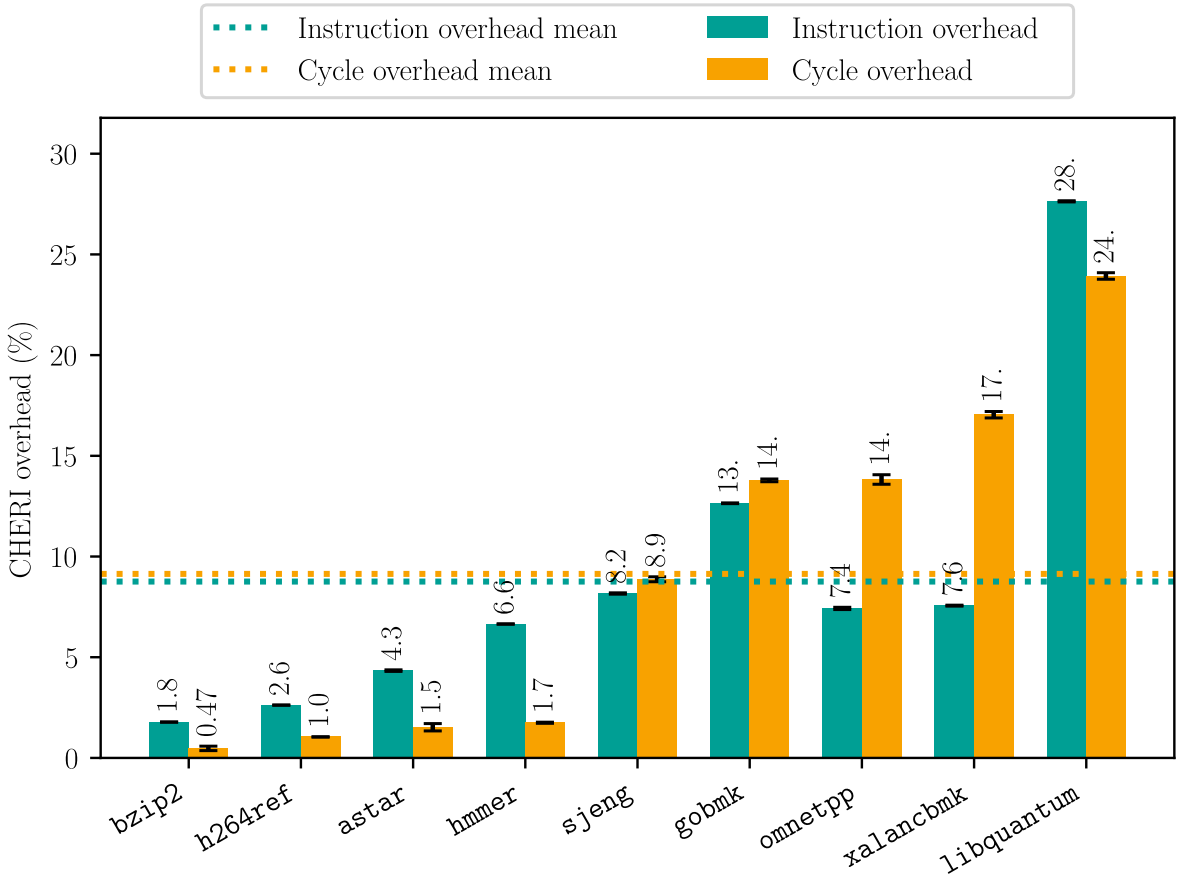

Figure 6.5: SPEC overhead of CHERI for the Toooba core. Arithmetic mean instruction overhead is 9% and cycle overhead is 9%. All benchmarks were run three times, excluding one anomalous run of `xalancbmk`. Error bars show one standard deviation (at most 0.24%).

of their loaded values are tagged capabilities, explaining their low overheads. The added instructions are likely to be mostly capability manipulations, which Toooba is able to execute quickly due to its superscalarity, even hiding this time behind memory latency. This effect is demonstrated by dividing the absolute instruction overhead by the absolute cycle overhead. Across these four benchmarks, this gives an average IPC of 3.1 for the added capability operations. Since Toooba is configured to be two-way superscalar, the maximum IPC is 2, so this proves that capability instruction overhead is being masked behind other latency. This explains the reduction in cycle overhead compared to instruction overhead for these benchmarks.

`sjeng` and `gobmk` both see relatively high instruction overheads and matching cycle overheads. These benchmarks are similar to each other: both are game engines, so explore game trees, meaning they are relatively pointer-dense. In both cases, approximately 30% of their loaded values are tagged capabilities. A large fraction of the overhead is caused by instruction cache misses: between them there is an average 70% increase in misses. The time spent by the instruction cache waiting to resolve misses accounts for 28% of the cycle overhead in both cases. This should be improved when the benchmarks are compiled with

support for compressed CHERI instructions. The data caches see less overhead: a 50% miss increase in both cases, which only accounts for 9% of the extra cycles for `sjeng` and 16% for `gobmk`.

`omnetpp` and `xalancbmk` see much higher cycle overheads than their instruction overheads. These are the most pointer-heavy benchmarks: in both cases approximately 60% of their loads are of tagged capabilities. Both see an approximate 100% increase in data cache misses due to extra bounds metadata: for `omnetpp` this corresponds to 26% of the gained cycles and 70% for `xalancbmk`. The instruction cache also sees more pressure, with 138% more misses for `omnetpp` and 64% more misses for `xalancbmk`. This corresponds to 48% and 32% of the added cycles respectively[1]. This should reduce when the benchmarks are compiled with CHERI-aware compressed instructions.

`libquantum` sees by far the highest instruction overhead, despite being very pointer-light: less than 2% of its loaded values are tagged capabilities. The cycle overhead is similar, and all other microarchitectural overheads are tiny (while instruction cache misses grow by 61%, less than 0.1% of the cycles are spent waiting for instruction cache loads). This points towards a code generation issue in a tight loop that is not fundamental to capabilities, similar to the `rc4` embedded benchmark (see Figure 4.11). Given the purpose of the `libquantum` code, it is likely that the tight loop is the inner loop of matrix multiplication. However, since the SPEC benchmarks are significantly larger in static instructions than the MiBench benchmarks, analysing code generation difficulties is trickier, and beyond the scope of this thesis.

In summary, performance overheads mostly vary as expected with pointer-density. With more pointers, data cache pressure increases due to metadata cache pollution. Instruction cache pressure also increases, though this may in part be caused by a lack of compressed capability instructions. Within the pipeline, there appears to be little unexplained cycle overhead that would indicate a microarchitectural issue.

Comparing to the microcontrollers' performance overheads, the instruction and cycle overheads are smaller for Toooba, supporting Hypothesis H.3. For pointer-light benchmarks, Toooba is able to have significantly lower cycle overheads than instruction overheads, indicating that the out-of-order engine is able to hide the capability operations behind memory latency. However, application-class workloads with high pointer-densities are able to pressure the data caches in a way that was not seen for the microcontroller benchmarks.

[1]Note the percentages for `xalancbmk` sum to more than 100% due to out-of-order execution allowing the processor to continue on a cache miss.

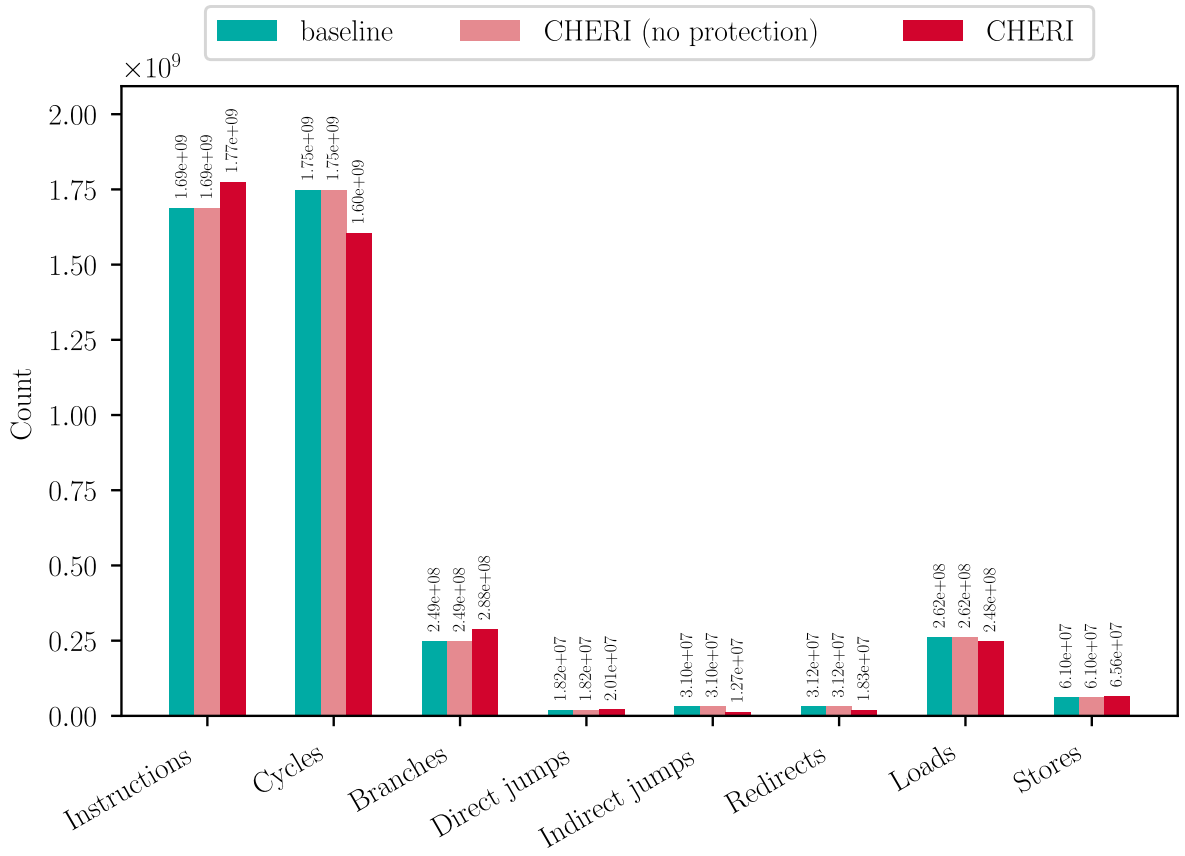

Figure 6.6: CoreMark run statistics for the Toooba baseline core, the CHERI core running the baseline software (no protection), and the CHERI core running pure capability code, including relevant performance counters. The working set fits in cache, so negligible cache misses are observed.

6.4.3 Microcontroller benchmarks

To enable direct comparison with the Piccolo and Flute microcontrollers, the same CoreMark and MiBench benchmarks are run atop Cheri-FreeRTOS on Toooba. The overheads of these benchmarks on the microcontrollers was described in Section 4.4.

Figure 6.6 shows the results of running CoreMark on the Toooba core. Despite a 5% instruction overhead, Toooba sees a -8% cycle overhead: a *decrease*. This is a result of the CHERI code generation's preference for branches over indirect jumps, leading to much more successful branch prediction on Toooba. This benefit is due to a code generation choice, and is not fundamental to capabilities.

The MiBench overheads are shown in Figure 6.7. The instruction overhead is the same as for Flute (since the binaries are identical) at 16%, while the cycle overheads are lower at 10%. Most of the benchmarks tell the same story of cycle overheads somewhat lower than their instruction overheads as Toooba hides the added instructions behind memory accesses. Since these benchmarks have small memory footprints, the caching effects seen for SPEC are not repeated here. `qsort` sees a more significant reduction in cycle overhead: since the large instruction overhead is caused by the copying granularity issues discussed

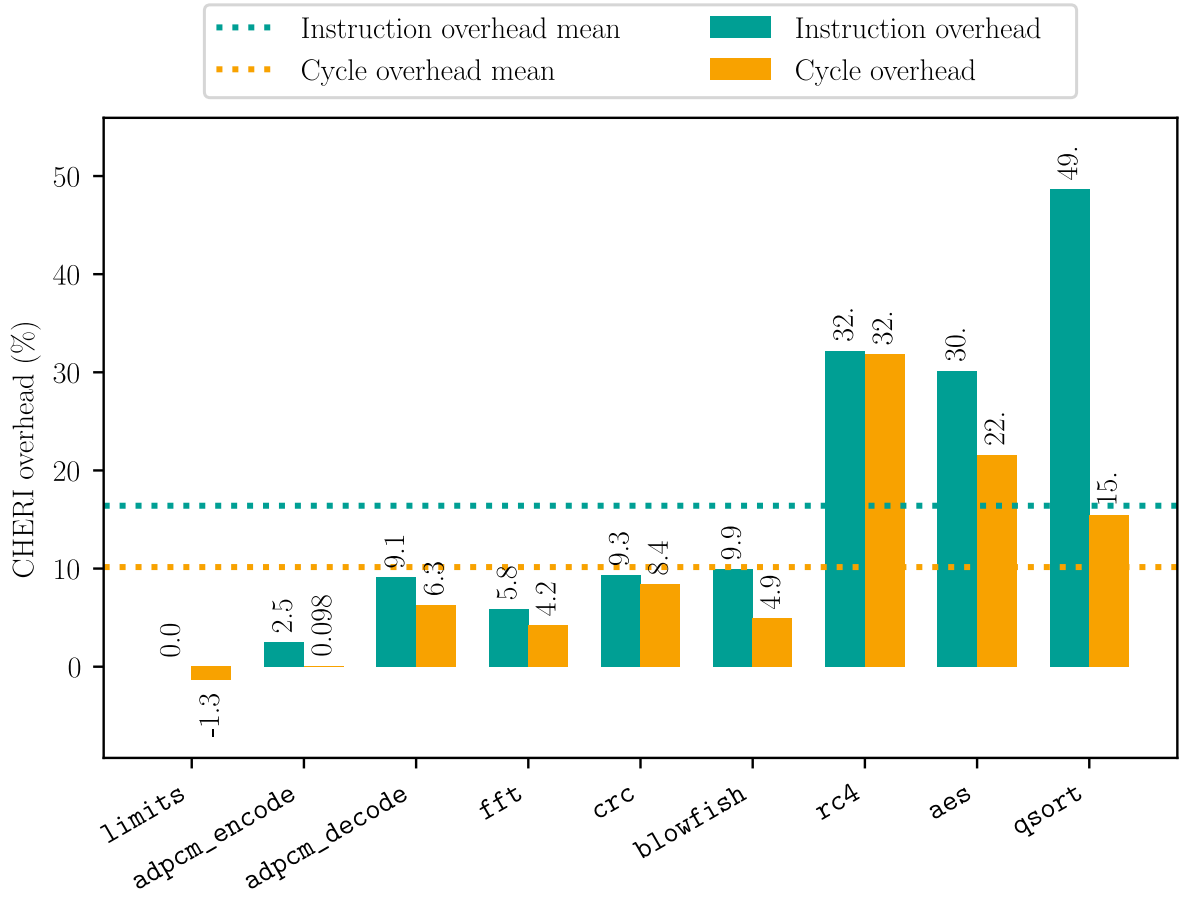

Figure 6.7: MiBench overhead of CHERI for the Toooba core. Arithmetic mean instruction overhead is 16% and cycle overhead is 10%.

in Section 4.4, the smaller accesses can be forwarded within the Load/Store Queue. `rc4` does not see a reduction in its cycle overhead compared to its instruction overhead, despite the added instructions all being simple arithmetic (see Figure 4.11). This is because the baseline achieves almost maximum performance—an IPC of 1.84—so there is no latency in which to hide the added instructions. While the lower Toooba frequency would be expected to reduce its cycle overhead compared to Flute due to reduced DRAM latency, this effect is not relevant since the benchmark data all comfortably fits in the caches so L2 misses are negligible.

These results match the predictions of Hypothesis H.3: the larger processor sees lower performance overheads.

Power usage	Baseline (mW)	CHERI (mW)	Overhead
Toooba (logic only)	252	406	61%
Toooba (overall)	808	1,289	63%

Figure 6.8: Power usage of the Toooba processor synthesised for the VCU-118, as reported by Vivado.

6.5 Power

As with the microcontroller evaluation, I report the power estimates generated by Vivado as well as DRAM traffic overheads, in this case for SPEC.

Figure 6.8 shows the power report for Toooba (dual-core) generated by Vivado. Toooba's overall CHERI power overhead is 63%, significantly exceeding its area overhead. As noted in Section 4.5, these power numbers can only be approximate, as dynamic usage patterns of the different logic elements are not known to the synthesis tools.

Figure 6.9 shows the L2 cache miss overhead due to CHERI on the Toooba core: an average of 21%. As can be seen in the graph, the number of cache misses, and so the significance of this overhead, varies between benchmarks. The L2 cache miss overhead estimates the DRAM traffic overhead, except that tag controller accesses are excluded. The tag cache is much larger for Toooba (128 KiB) than the microcontrollers (4 KiB). This avoids anomalies due to having so few tag cache lines. Therefore, the DRAM traffic overhead due to tag lookups can be expected to be significantly smaller. As a very loose upper bound, a miss in the tag cache on every lookup would imply a doubling of DRAM traffic overheads compared to L2 cache miss overheads.

The increase in Vivado power overhead for Toooba compared to Flute defies the trend proposed by Hypothesis H.3. This is at least in part due to the area-heavy design choices made, but the reason that the power overhead exceeds the area overhead by so much is unclear. The power overhead incurred by added DRAM traffic is smaller for Toooba than the microcontrollers, however. This is true even taking a conservative estimate for the traffic generated by tag lookups. DRAM traffic overheads scale well because the larger and more sophisticated caches can absorb much of the extra capability metadata traffic. For example, the write-through nature of the microcontroller caches led to significant additional DRAM traffic overhead from capability writes.

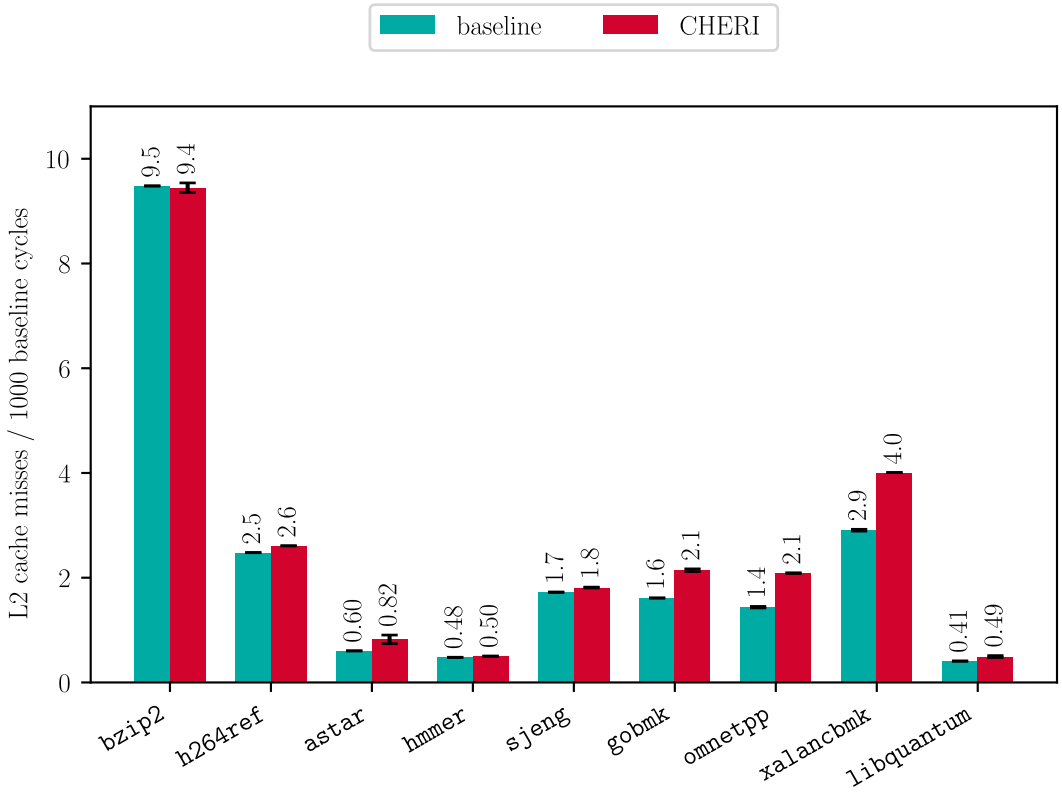

Figure 6.9: Toooba core L2 cache misses per thousand cycles for SPEC with and without CHERI. CHERI incurs an arithmetic mean overhead of 21% cache misses per thousand cycles. Note that the misses are normalised per baseline cycle, which avoids an apparent reduction in misses for CHERI due to cycle overhead. This estimates DRAM traffic overhead, but excludes the traffic from the tag controller. All benchmarks were run three times, excluding one anomalous run of `xalancbmk`. Error bars show one standard deviation.

6.6 Security

Toooba was evaluated using the same methodology as Piccolo and Flute, although was not part of the Fett bug-bounty program.

Due to implementing the same architecture as Flute, the Toooba core performed the same against the CWE test suite [30], demonstrating defence against spatial safety vulnerabilities and numeric errors that impact pointer arithmetic. It also does not share Piccolo's bounds overflow failures due to rounding, as the 128-bit capability format allows high bounds precision.

As discussed in Section 5.3.6, evaluation by others of the processor for speculative side-channel security revealed the serious Meltdown Capability Forgery vulnerably [48]. Prompted by this, an audit of the microarchitecture (described in Section 5.3.6) characterised all possible instances of this attack, confirming that they are mitigated by switching to a tag-clearing architecture.

6.7 Future work

Two further aspects of evaluation could be measured as future work: SPEC overheads with capability-aware compressed instructions, and additional DRAM accesses from the tag controller. However, these seem unlikely to change the overall conclusions drawn. In addition, the Toooba processor evaluated does not include the change to tag-clearing discussed in Section 5.4. This change cannot affect performance, as it does not affect any operations unless there is a security violation, which does not occur for the benchmarks used. The area overhead is expected to be small, or even negative as some checks are removed. Confirming this is future work.

As with the microcontrollers, all evaluation was carried out on FPGA. Synthesis using ASIC tools may give different insights into area and timing overheads. This would also allow the timing impact to be evaluated meaningfully at higher frequencies. The difference in relative DRAM frequency, especially given Toooba's low FPGA frequency, may also make a significant performance difference. Another approach to mitigate this effect might be to delay memory accesses on FPGA artificially.

6.8 Summary

This chapter has given answers for the quantitative aspects of Hypothesis H.2, confirming that CHERI is plausible for application-class processors. We have seen that the CHERI extensions help to provide spatial safety for Toooba. The LUT area overhead was 45%. Again, the capability manipulation logic is a significant contributor to the overhead, but almost all components (apart from the FPU) see significant overhead to support capabilities. This confirms the potential for the optimisations suggested in Figure 5.3 to produce significant area savings. The critical path of the design is not significantly affected, though no timing optimisation was carried out. Performance overheads are negligible for unmodified code, and 9% (average SPEC cycles) for pure capability code. Power overheads are mixed, with Vivado reporting a 63% overall power overhead, but with average L2 cache miss overhead across SPEC at 21%.

Hypothesis H.3 holds in that performance, DRAM traffic overhead, and timing (which required no effort to optimise) have scaled well compared to the microcontrollers. The performance improvement is due to out-of-order execution allowing new operations to run in parallel with memory accesses. However, area does not show such a strong trend and logic power shows the reverse trend due to the widespread extension of processor components with capability metadata. It is expected that the trend will be restored after applying further optimisations to the CHERI Toooba implementation.

As noted in this chapter and Chapter 5, CHERI poses a tradeoff between performance, power, and area in its implementation. However, the underlying core already offers many

parameters that allow these metrics to be tailored, for example superscalar width and reorder buffer size. Therefore, it is not possible to offer a definitive evaluation without being aware of the particular constraints of an application. In order to keep application-class processors fed with instructions and data, a large fraction of their silicon die area is used for caches. This may make a relatively high CHERI area overhead less of a problem in practice, as the core occupies a small area of the chip and the caches can be modestly scaled down without significant performance impact.

Chapter 7

Accelerating temporal safety

This chapter discusses heap-based temporal safety atop CHERI, including developments to the architecture-neutral model for revocation and microarchitectural work to adapt and accelerate this for RISC-V, addressing Hypothesis H.4. While the discussion applies to all scales of RISC-V core, implementation is only carried out for Toooba, which has the most sophisticated memory subsystem so highlights the most obstacles. Temporal safety on CHERI is still an area of active research, with the revocation algorithms still being developed, so this chapter surveys a wide range of features, only some of which have been implemented.

7.1 High-level approach

Temporal safety can refer to a wide variety of properties. For example, race conditions allow behaviour to deviate from what was intended based on different interleavings of concurrently executing threads [92]. The difference in behaviour may be exploitable. For instance, time-of-check to time-of-use vulnerabilities occur when a property is checked and the result is used to authorise some operation non-atomically, allowing the property to be changed before the operation is performed [19]. It is possible to prevent race conditions in certain contexts: for example, Rust prevents data races with its "ownership" enforcement [83]. This problem is out-of-scope for this chapter. As defined in Section 2.3, temporal safety in this chapter only refers to enforcement that objects be used only within their lifetime. However, it should be noted that CHERI's atomic enforcement of bounds checks in hardware prevents time-of-check to time-of-use vulnerabilities where the check is that a pointer is in-bounds, and the use is dereferencing it.

Within this definition of temporal safety, we further restrict to discussion of temporal safety on the heap. Other early work has shown promise of capabilities for enforcing temporal safety on the stack [52]. Temporal safety violations on the heap take the form of use-after-free violations. To avoid vulnerabilities due to heap use-after-free errors, dangling

pointers, i.e. pointers to freed memory, must be invalidated before that memory is reused. This prevents confusion between the old and new object. Double frees occur when the same pointer is passed to `free` twice [31]. These can be considered a subset of use-after-free violations, requiring only that the allocator checks that the pointer is still valid on `free`.

Many attempts have been made to mitigate temporal safety issues in this way: these are discussed in Section 2.3.1. Some approaches, such as *Boehm GC*, are conservative, sweeping memory to ensure there are no pointers that alias with allocations before freeing them. This means they suffer from both false positives, as memory cannot be freed if an integer holds a value that aliases with its address, and false negatives, as pointers to freed memory may be hidden and recovered, or even synthesised from nowhere. Approaches that avoid being conservative, such as *DangNull*, track pointers by instrumenting loads and stores of pointer types. This contributes to their large slowdowns by making loads and stores of pointers incur significant additional memory pressure. They can also suffer from false negatives in the presence of pointer arithmetic. Type-agnostic operations, most notably `memcpy`, also cause problems that must be addressed in various ways. Capabilities offer a way of avoiding the issues with these two approaches. Pointers are tracked precisely by the capability integrity tag, avoiding the need for conservatism. In addition, since the memory hierarchy already keeps track of what can be a pointer, memory instructions do not require further instrumentation from the compiler, avoiding this performance penalty.

7.1.1 Sweeping revocation

Compared to its direct spatial safety benefits, CHERI makes revocation of granted authority difficult since capabilities can be liberally copied and are not indirected. Present revocation techniques over CHERI therefore require all of memory to be scanned to find and invalidate all copies of the capability to be revoked. This overhead can be hidden, such as by amortising over a large number of collected frees [138], or sweeping on a parallel hardware core [132]. Parallel sweeping requires an invariant to be maintained: running applications must not be able to bypass the sweep by copying data from unswept pages to swept pages. This can be ensured either by prohibiting stores to unswept pages or loads from swept pages. This is discussed more in Section 7.2.1.

Prior to my project, proposals for CHERI revocation revolved around "filter registers": a small number of registers describing contiguous ranges currently being revoked. This was to allow the sweeping invariant to be easily maintained as memory accesses into swept regions could be quickly checked against the filter register to determine if they copied revoked allocations. The number of filter registers had to be kept small due to the associative checks—analogous to a PMP—that had to be performed on every memory access. This restricted revocation sweeps to revoking access to very small regions at a time, despite sweeping all of memory each time. I proposed instead to maintain a shadow bitmap: a region of memory containing bits corresponding to all of userspace, where each

bit describes whether a 16-byte allocation granule at the corresponding address is within a revoked allocation. This allowed each sweep of memory to process an unlimited number of revoked capabilities. The results of this approach, including a software implementation of the revoker (and additional optimisations) by other members of the CHERI team, are described in our *CHERIVoke* paper [138]. The resulting algorithm broadly works as follows:

- Mark in the shadow bitmap when memory is freed.
- Quarantine the memory in the allocator, preventing it from being reused until all capabilities to it have been revoked.
- Wait until a large amount of memory has been freed.
- Perform stop-the-world sweeps, searching over the entire address space for capabilities into freed memory. Any matching capabilities have their tag cleared, revoking them.

The approach is divided between the allocator, running in userspace, and the revocation service provided as a syscall into the kernel. This syscall takes the revocation bitmap as an argument. The revocation loop itself then runs as a concurrent kernel process. The allocator is responsible for quarantining freed allocations until the kernel reports the relevant revocation sweep is complete. This means that the application gains use-after-reallocation guarantees simply by using a CHERI-enabled allocator. The allocator is already trusted by the application for this kind of correctness, such as to ensure that allocations do not alias spatially. Therefore, the trust relationship between the application and the allocator is not significantly changed.

The results of this approach are discussed in Section 7.4.

7.2 Optimising sweeping revocation

Given the changes to the sweeping algorithm described above, the core of the intra-page revocation loop is as shown in Figure 7.1. This loop is critical for optimisation since it must be run for a large fraction of pages in the system on revocation sweeps. This section discusses optimisations that can be made to improve the performance of the sweep, and how they have been or could be applied to the Toooba core. Optimisations for finding tags, discussed in Section 7.3, can also accelerate sweeping.

```
for (cap_t *cline = start; cline < end; cline += STRIDE) {
  int tags = CLoadTags(cline);
  size_t cline_offset = 0;
  while (tags > 0) {
    if (tags & 1) {
      cap_t cap = cline[cline_offset];
      if (CGetTag(cap) && !(CGetPerms(cap) & VM_PROT_PERM)) {
        if (getShadowMap(cap)) {
          cap = CClearTag(cap);
          cline[cline_offset] = cap;
        }
      }
    }
    tags >>= 1; ++cline_offset;
  }
}
```

Figure 7.1: C-like pseudocode implementation of the core CHERI revocation loop. Reproduced based on the algorithm in our *CHERIVoke* paper [138]. `CLoadTags`, `CGetTag`, `CGetPerms`, and `CClearTag` each use intrinsics to execute the corresponding CHERI instruction. `cap_t` is a type to hold a capability: in real code, this can be a `void*` pointer. `STRIDE` is detected dynamically by the OS on startup, and indicates the number of tags returned by `CLoadTags`. `VM_PROT_PERM` is a mask extracting the capability permission indicating the capability belongs to the allocator itself, so should not be revoked. `getShadowMap` performs a simple transformation of the base of the input capability to find whether it points to revoked memory. To enable concurrent sweeping, care is required to avoid race conditions with other threads.

7.2.1 Virtual memory

Virtual memory can be used to track which pages have been swept during concurrent revocation and can allow entire pages to be omitted from sweeps if they are known not to contain any capabilities that need revoking.

In *CHERIVoke*, we described a store-side barrier, where hardware prevents processes from storing revoked pointers to pages that have already been swept. The recent CheriBSD revocation kernel has moved to instead maintaining a load-side barrier, preventing processes from ever loading in a revoked capability from an unswept page. The load-side barrier requires altered virtual memory support to raise an exception on capability-width loads from such pages, so that the revoker can check the loaded capability and sweep the page in case it is accessed again. This prevents the application from repeatedly taking exceptions due to accessing the same page. I have implemented these virtual memory changes in Toooba by augmenting the TLB to raise exceptions when capability-width loads are attempted from a page with this bit set. A further optimisation would only raise the exception when the loaded value is tagged. However, this would prevent the instruction from retiring until the value is returned from memory, potentially harming performance.

Evaluation is required to determine whether this is worthwhile.

Furthermore, we used capability-dirty tracking to exclude pages containing no capabilities from the sweep. This again requires virtual memory support to track which pages have been written with capabilities. A bit is added to the PTE to record this information. The tracking can be implemented either by changing the PTE atomically in hardware, or by raising an exception whenever a capability is stored to a capability-clean page. I have implemented the second approach in Toooba. Note that the first implementation poses difficulty on multicore processors to write the changed PTE from the TLB to memory. For this reason, the baseline Toooba implementation also does not atomically update its analogous bits that track whether the page is accessed or dirty, instead taking the exception approach.

7.2.2 As-user memory accesses

In addition, hardware support is required to allow the revoker—running in supervisor mode—to act as the user while examining the capabilities in memory. For example, RISC-V page-tables contain bits to prevent supervisor mode from mistakenly accessing user memory, either accidentally or by deceit. The revoker must bypass this protection. RISC-V has an existing mode, switched via a CSR, to allow supervisor mode to access these pages. However, a finer granularity is required for performance as user-mode accesses (to the heap) need to be regularly interleaved with supervisor-mode accesses (to the shadow map) while sweeping.

This required implementation of additional copies of load and store instructions: primarily adding the additional decode cases, and allowing the override of privilege mode to be routed to the TLBs, where the permission checks are performed. An as-user version of `CLoadTags` was also required.

7.2.3 Prefetching

The revocation loop is a prime candidate for optimisation using prefetching [138]. Since memory is scanned sequentially, the lines can be prefetched in iterations ahead of when they are needed. The sweeping loop accesses many lines that are unlikely to be resident in cache, so avoiding the latency for cache misses has the potential to speed it up significantly.

RISC-V prefetch instructions have only recently been specified [47], and are not present in the Toooba processor. However, a prefetch can be achieved by performing a normal load into register zero. On an out-of-order processor, this loads the value into the cache hierarchy, but does not have to delay execution for the cacheline to be returned, as no other instructions depend on the value. I have confirmed that this approach can improve performance by benchmarking a small prefetch loop (see Figure 7.2). However, this

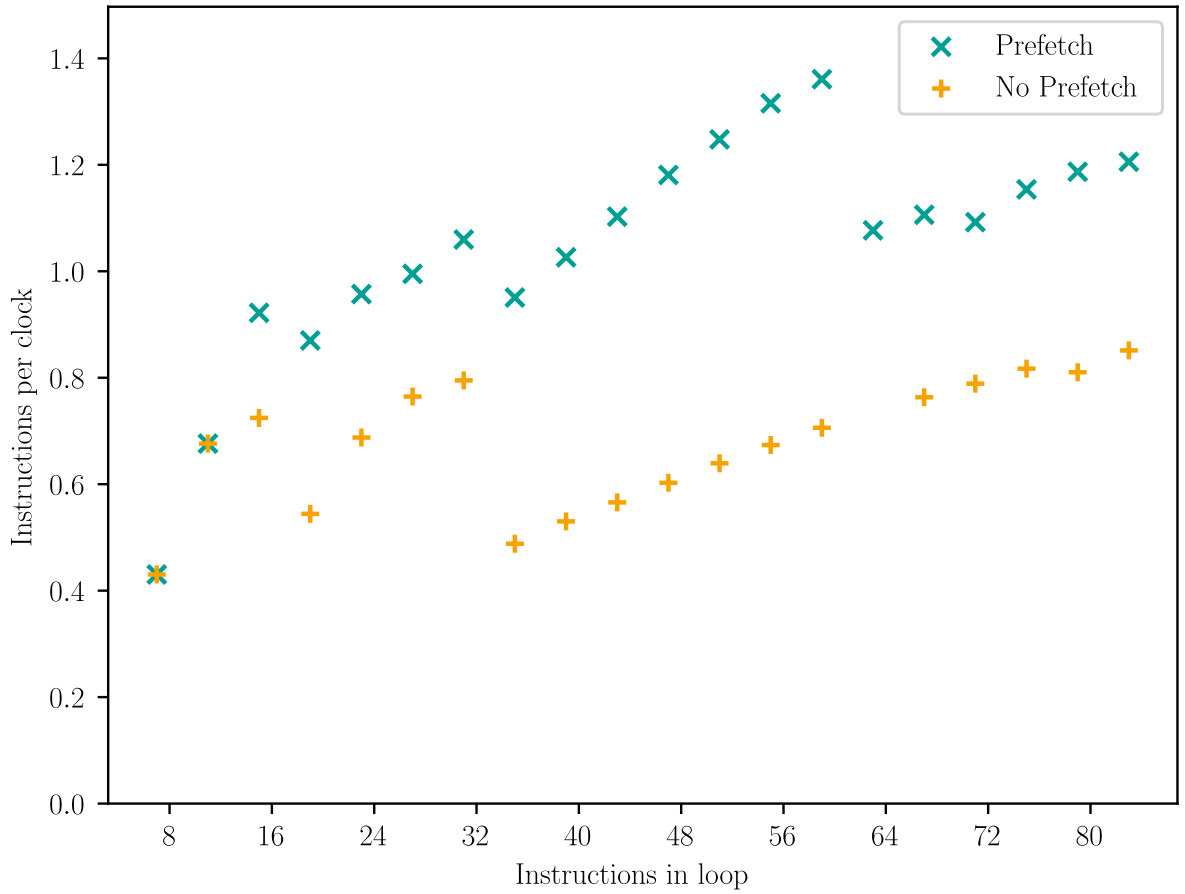

Figure 7.2: Performance of Toooba for an artificial benchmark loop with and without software prefetch. The loop iterates through memory, loading a value that misses in all caches each iteration. The prefetch loop has an additional load of the following cacheline into register zero. Each iteration then performs a variable number of dependent arithmetic instructions. Note that performance drops at factors of the reorder buffer size (64) as one fewer concurrent load can be issued. Prefetch does not help for small loops, as the reorder buffer has space for eight loads: the maximum that can be serviced concurrently. Executed on the VCU-118, as in Chapter 6.

prefetching approach potentially requires software to check before prefetching that the load will not trigger an exception due to running past the page being swept. Dedicated prefetch instructions would not raise exceptions in this case.

7.2.4 Dedicated sweeper

Cornucopia [132] confirmed that offloading the sweep onto a parallel hardware core significantly reduces application overhead. However, this prevents that core from doing other application work, harming multicore application performance. To address this, specialised hardware could be provided to perform revocation sweeps. This would have a much smaller area than an entire core, so could give the benefits of offloading without reducing the number of cores available for applications.

One possibility is to implement a device akin to a DMA controller, specifically dedicated to revocation. This would accept requests to sweep pages, perform the sweep, then report it has finished in a way that allows epochs to progress. However, as the shadow map is specified in virtual memory, such a device would require core-like page-table walk and TLB hardware. In addition, the OS kernel overhead from managing such a device and handling interrupts from it is likely to undermine most of the performance benefit, especially if pages are specified only on a 4 KiB granularity.

A more practical alternative would be to have a smaller core, such as Flute, with the revocation process pinned to it. This heterogeneous compute approach is reminiscent of Arm's big.LITTLE architecture [8], and indeed the core could be used for general-purpose compute when not required for sweeping. The smaller core could be specialised for efficient sweeping, for example with streaming caches and customised hardware prefetching.

7.3 Finding tags

CHERI MIPS implemented `CLoadTags` for *Cornucopia* [132]. This section investigates its implementation for RISC-V and in particular the implications for Toooba's more mature memory subsystem.

`CLoadTags` loads the capability tags of consecutive capabilities in memory from a given start address, returning the result as a bitmask. The number of tags returned is implementation-defined, but is expected to correspond to the width of a cacheline.

Regardless of the method of describing capabilities to revoke and the invariant maintained, memory addresses can quickly be excluded from the sweep if they do not contain a capability. Allowing tags to be queried separately from the corresponding data enables more efficient sweeping: this is the purpose of the `CLoadTags` instruction. An immediate saving is in the presentation of the information to software, allowing untagged values to be skipped over more quickly by loading multiple tags at a time and avoiding the requirement for `CGetTag` instructions. However, a more fundamental potential saving is that the data itself need not be loaded in, avoiding power and performance penalty from accessing DRAM. In addition, tags can be cached much more densely alone than alongside the data: Toooba's configuration of the tag controller holds 64 bits of tags (a quarter of a page's worth) per cacheline. Further, it may be possible to avoid polluting the cache with the unneeded data, provided tags can be read around the caches.

This section discusses a spectrum of alternative implementations for `CLoadTags`:

- Loading the cacheline in as normal and servicing the request in the L1 cache;
- Adding a separate tag-only state to the caches and fetching the tags from the tag controller without data;

- Adding way-restriction for tag-only lines to avoid cache pollution;
- Relaxing coherence requirements and fetching the tags directly from the tag controller.

While the latter two approaches are described in detail, implementing them is future work. An evaluation to quantitatively compare the approaches is also future work.

Apart from the relaxed coherence approach, these implementations are architecturally equivalent, making TestRIG invaluable in fuzzing them against one another to identify bugs quickly. By constructing a template, including a sub-template to ensure a cacheline is evicted from the cache, simple bugs were revealed quickly, while deeper bugs were caught with specific template design. Each implementation was also tested using `cheribsdtest` to ensure they allowed the revoker to detect tags correctly.

7.3.1 Toooba memory subsystem

This section describes the Toooba memory system, giving the required context to understand the implications for implementing `CLoadTags`.

Each Toooba core has a single memory pipeline, which makes requests to a Load/Store Queue. The Load/Store Queue in turn makes requests to a per-core L1 data cache. The L1 caches interact with a shared L2 cache, which in turn makes DRAM transactions. In the CHERI version, the tag controller is added on the path between the L2 cache and DRAM, as discussed in Section 3.4.5. The caches are all coherent, using the standard `MESI` [100] cache coherence protocol. Each L1 cache records the state of each cacheline, and the L2 cache keeps a directory of the states of lines in the child caches to coordinate the coherence protocol and broadcast required updates. Coherent DMA is supported via an additional interface of the L2 cache.

The Load/Store Queue contains entries for pending memory accesses. On a new access, the address is checked to determine if it overlaps with any other pending access. This allows forwarding from stores to loads. The size of the Load/Store Queue used throughout this chapter is shown in Figure 6.1.

The L1 cache can support up to eight concurrent transactions, while the L2 cache can support 16. This aligns with the number of ways per cache set in the two caches, guaranteeing an idle way is always available for each pending transaction.

7.3.2 Initial implementation

The many locations tags can be stored [63] implies a somewhat complex implementation of the `CLoadTags` instruction to ensure coherence. Ideally, a large array of tags could be fetched directly from the tag controller's tag cache (or its backing memory). However, this would cover multiple lines of data, each of which could be present in caches with tags updated to be more recent than those stored in the tag table. As such, in the initial implementation, `CLoadTags` requests are limited to providing tags for a single cacheline at a time: the smallest granularity on which tags are held in the memory hierarchy. The implementation must check for hits in all caches up the hierarchy, the same as for normal data loads.

The baseline implementation of `CLoadTags` I implemented simply loads the data into the L1 cache then presents the tags for the cacheline to software. This only requires changes on the interface between the pipeline and the L1 cache, and is sufficient to allow the revoker to run on the Toooba core. Care is required in the Load/Store Queue, as the `CLoadTags` can overlap with multiple existing stores. This approach does not provide the key benefit of `CLoadTags`: preventing data from being loaded into the caches for lines that contain no capabilities.

7.3.3 Avoiding data loads

Rather than avoiding some arithmetic instructions, the primary aim of the `CLoadTags` instruction is to allow optimisations within the caches. The main observation is that the data is completely unused if there are no tags in the cacheline, and thus a miss does not need the data to be fetched from DRAM. Since DRAM accesses use significant dynamic power, and since a large portion of cachelines contain no valid capabilities, this alone could mitigate a large fraction of the power overhead of sweeping. An improved implementation of `CLoadTags` is to fetch only the tags in the event of a cache miss, since the tags are cached densely in the tag cache and can be fetched independently of the data.

Several options present themselves to allow caches to deal with tags separately to the corresponding data. One option is just to read around each cache once it is confirmed that it does not contain the relevant cacheline. This is the approach taken by CHERI MIPS. However, this approach presents challenges for Toooba's more mature memory subsystem, which must support concurrent requests, since there is no existing means to make a request to a parent cache without allocating the cacheline in the child cache. Reading around the caches may cause coherence difficulties due to race conditions: without careful implementation, a tag-only read could overtake a write in the cache and read stale tags. This approach would also require adding an entirely new interface to the L2 cache and tag controller to allow the core to query them directly, rather than via the child caches.

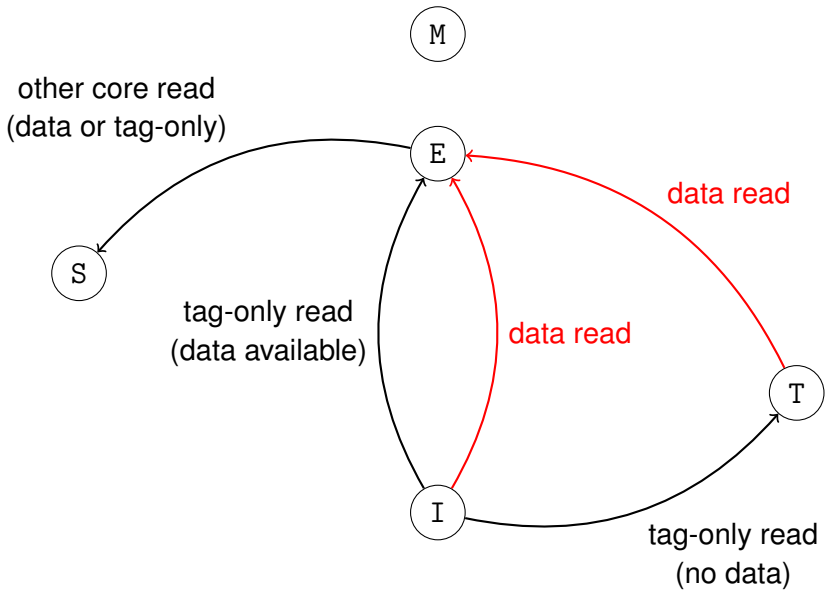

Figure 7.3: The `MESTI` cache coherence protocol for Toooba's L1 data caches. Red arrows show transitions that may incur a DRAM data read. Transitions are omitted that are not altered compared to `MESI`: from every state to `M` on a data write, from every state to `I` on another core's data write and also from `I` or `T` to `S` on a data read where the cacheline is already resident in another core's cache.

To avoid these problems, I use the existing cache coherence mechanism, augmenting it to inherit its coherence guarantees automatically, and avoid additional complexity. To allow caches to contain data for tags only, I augment Toooba's `MESI` cache coherence protocol with an additional state: tag-only (`T`). The `T` state is functionally the same as the `S` state, but indicates that only the tags and not the data are valid. Since capability tags cannot be modified within the ISA without writing the corresponding data, the tag-only state is necessarily read-only so fits between the `S` and `I` states in the protocol. I thus refer to this as the `MESTI` cache coherence model (see Figure 7.3).

A tag-only request hits if the cacheline is resident in any state other than `I`, i.e. `M`, `E`, `S`, or `T`. Care is required, since upgrading the state from `T` to `S` or any higher state requires an access to the next level of cache, since the data is not correct. This previously could not occur, since any non-`I` state implied the data was correct, and the only consequence of an upgrade was to send a message to invalidate sibling caches' copies of the cacheline. Thus, careful consideration of the cache state machine was required to ensure incorrect data could never be loaded into the processor.

Since the revocation use-case is expected to `CLoadTags` each cacheline in a page exactly once per sweep, keeping a cacheline in the `T` state is unlikely to be useful since the tags will not be needed again unless the data is also required. A possible further optimisation would be to make an upgrade request to the `E` state in L1 cache automatically if a tag-only request returns any non-zero tags, in anticipation of the revoker loading the full capability. If all-zero tags are returned, the cacheline could be prioritised for eviction (see Section 7.3.4).

The interface between the L2 cache and the tag controller also required augmentation to allow the caches to communicate a tag-only request. Rather than adopting the MIPS approach of adding an additional interface to the tag controller for this purpose, I instead opted to signal the request using the existing AXI interface. The `aruser` field is extended to one bit, with a value of 1 signalling that the request is tag-only. I modified the tag controller to convert such an AXI request into an internal tag-only request. Care was also required to ensure that tag-only responses would not be interleaved within data response bursts.

7.3.4 Avoiding cache pollution

In the course of sweeping through a page, tags will be loaded in for every cacheline, implying every cache set is touched and has its data evicted. This implies subsequent cache misses as the data is fetched back in.

Since the (hopeful) common case for a revocation sweep is to load the tags for a cacheline once, realise they are zero so nothing needs revoking, and move onto the next cacheline, streaming semantics are desirable. Tags are not expected to be reused, so there is no incentive to keep them around in the cache. Thus, an approach that avoids them evicting other data should reduce performance loss when switching back to the non-revocation thread. This optimisation is most significant when applied to the L2 cache as it is shared between the child cores.

A natural implementation might restrict tag-only lines to a single way in each cache set. This way would need to contain space for the data as well, to allow the cacheline to be upgraded in-place if the data is also needed. However, the Toooba caches fix the size of their operation queue equal to the number of ways per cache set, allowing the caches to make progress even if all outstanding operations access the same cache set. Way-limiting tag-only lines would break this invariant, as a second `CLoadTags` to the same set would have to either stall or overwrite the first's metadata. This should never happen in revocation as memory is scanned sequentially, so there will be an even alternation between sets, and as many concurrent `CLoadTags` can be supported as sets in the cache: eight in Toooba's default configuration. However, it is important to ensure correct behaviour even if the instruction is not used for revocation. This would mean adding logic to stall `CLoadTags` in this case.

As an alternative implementation, all ways could be permitted to hold tag-only lines, but the first way of each set prioritised for tags to constrain cache pollution when sweeping. This would mean that the common case will have the desired behaviour—only one cacheline in each cache set is used for tags—while behaviour will be correct even in the face of unexpected usage patterns. Note that, since the cacheline is resident after the `CLoadTags`, upgrades to the cacheline will occur in-place, so the cacheline will stay in the intended way even if it contains non-zero tags.

7.3.5 Relaxing consistency

An alternative `CLoadTags` implementation would be to significantly relax the consistency requirements on the `CLoadTags` instruction. The correctness of load-side sweeping relies only on finding all capabilities that existed at the start of the sweep. Thus, provided tags presented by `CLoadTags` are at least as recent as the start of the current sweep, revocation will be correct. This would allow `CLoadTags` to obtain its tags directly from the tag controller, bypassing the processor caches, provided tags are written back to the tag controller at the beginning of a sweep. As well as a simpler lookup state-machine, this would allow `CLoadTags` to return many more tags at a time than a cacheline's worth (64 rather than four).

As the epoch is switched, all processes must be suspended to scan their register files for revoked capabilities. This presents an opportunity to write back the contents of the caches in parallel while minimising the impact on application performance. The writing back could be performed asynchronously, triggered by an explicit instruction, with regular memory accesses prioritised. Conservatively, all dirty lines could be written back to memory. However, this may incur significant DRAM traffic. An alternative is to just write the tag bits back to the tag controller, allowing the tag cache to become inconsistent with the data in memory. Since the only case where this inconsistency occurs is where there is more recent data in the processor caches, this cannot lead to confusion. The processor will see only the content of the caches until the cacheline is written back, in which case the tag cache is brought back in line with the data in memory. Furthermore, only the tags of dirty lines with the tags set need be written back. This introduces a tradeoff between additional write traffic at the start of an epoch and redundant loading of data during sweeps of lines that actually contain no capabilities. The worst case would write back 16,384 lines for Toooba's 1 MiB L2 cache.

7.4 Evaluation

This section estimates the performance that can be expected for RISC-V revocation, based on the results of *Cornucopia* [132]. Actually running the revoking kernel on RISC-V hardware is future work.

Our *CHERIVoke* algorithm was implemented and evaluated by *Cornucopia*. The authors perform evaluation on the dual-core CHERI MIPS processor on a Stratix IV FPGA at 50 MHz. Overheads are relative to pure capability code, so assuming spatial safety is provided by the baseline. An allocator-agnostic wrapper performs the necessary shadow-map management on every call to `malloc` and `free`. The results, including ablation to determine the causes of overheads, are shown in Figure 7.4. The geometric mean run-time SPEC overhead of revocation by the CheriBSD kernel is 5.8% (1.9% with offloading)

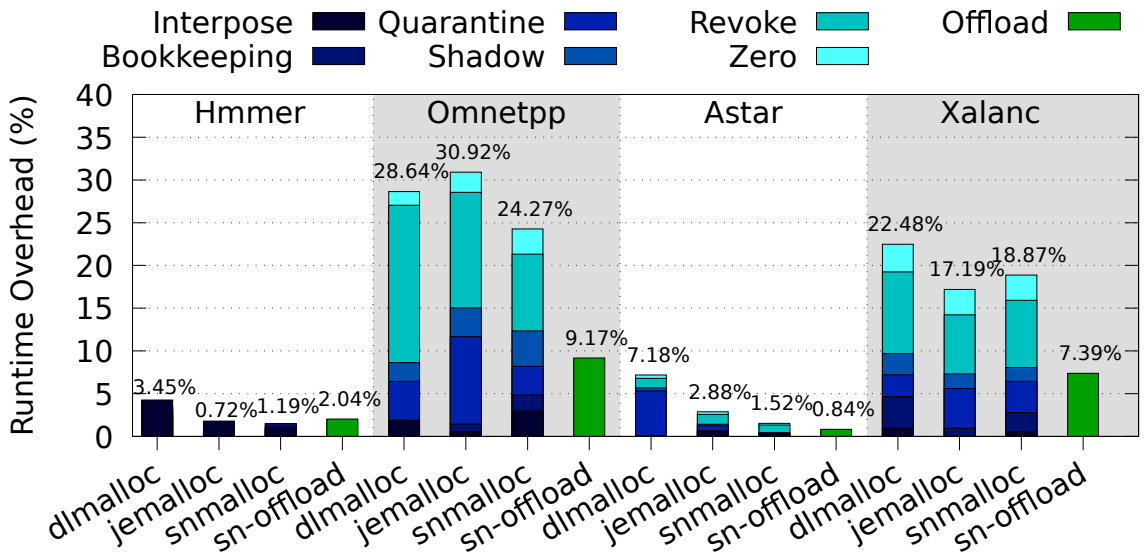

Figure 7.4: Performance overheads on **worst-performing** SPEC (`test`) for *Cornucopia* sweeping revocation using a generic wrapper across a range of allocators. The figure is extracted with permission from *Cornucopia* [132]. Results are included to show the application overhead when sweeping is offloaded to another hardware core.

with a worst case of 26.9% (8.9% with offloading). This is much lower than alternative approaches to temporal safety, as shown in Figure 7.5.

Optimisations performed on CHERI MIPS to enable these results included: use of page table bits to exclude pages that cannot contain capabilities; architectural support for a streaming `CLoadTags`; and performing sweeps without stopping the world, so that the pause time is reduced. The concurrent sweeps can then also be offloaded to another core. As discussed in this chapter, these optimisations can all be applied to the RISC-V Toooba core, so the results provide an estimate for the performance overheads of temporal safety in this new context. In fact, Toooba's out-of-order microarchitecture may significantly accelerate the sweeping loop. Further refinements have also since been made to the sweeping algorithm by others in the CHERI team, for example the load-side barrier discussed in Section 7.2.1.

7.5 Alternative capability semantics

As discussed above, the need for sweeping revocation stems from capabilities being both freely copied and not indirected, meaning that granted authority can spread throughout the system without mediation of its use. Relaxing either of these capability constraints has the potential to allow much cheaper revocation: *linear capabilities* cannot be copied, and *indirect capabilities* are indirected through another capability. These are both proposed as experimental extensions to the CHERI architecture [128]. They both allow revocation in constant-time, without having to sweep through memory. Unfortunately, either approach

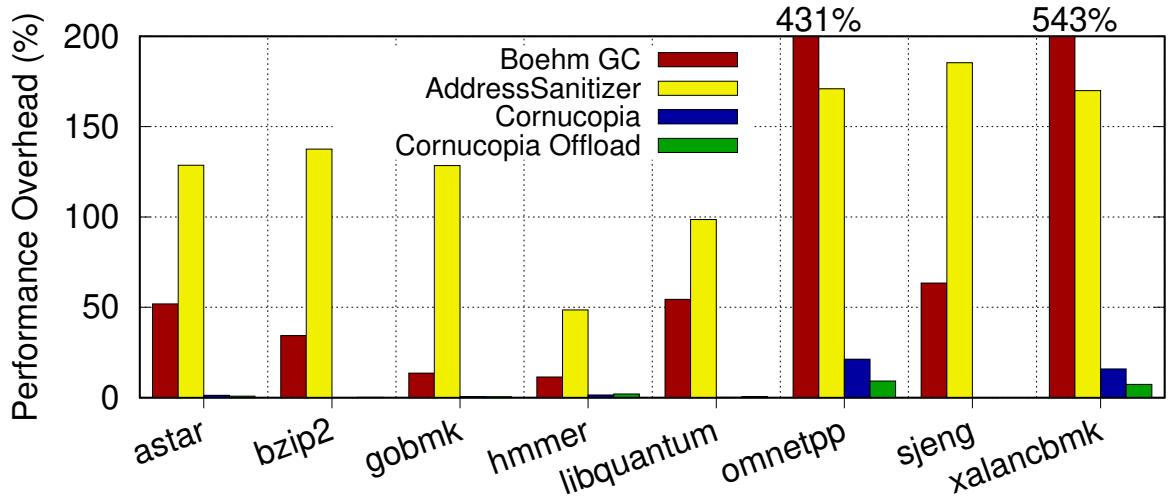

Figure 7.5: Performance overheads on SPEC (`test`) for *Cornucopia* sweeping revocation, compared to *Boehm GC* [20] and *AddressSanitizer* [115]. The figure is extracted with permission from *Cornucopia* [132]. Results are included to show the application overhead when sweeping is offloaded to another hardware core.

would require significant compiler and software upheaval, since they change the way in which code interacts with pointers. As such, they remain only theoretical: this section discusses how they might be implemented in RISC-V.

7.5.1 Linear capabilities

Linear capabilities cannot be copied: any move, store or load operation invalidates the original capability. This allows the allocator to confirm that the capability has been revoked by simply checking that the capability has been returned, since this guarantees no other copies can exist. Skorstengaard, Devriese, and Birkedal prove that enforcement of linear capabilities can guarantee control flow and data encapsulation properties [120].

Direct implementation requires multiple register writebacks to invalidate the source register while also updating the destination. This would complicate RISC-V microarchitecture, which otherwise only ever writes back one general-purpose register per instruction. This is particularly problematic for Toooba, which allocates a single physical register per instruction, allowing it to avoid the possibility of structural hazards by parametrising its physical register file size. The linearity must also extend to memory, meaning load instructions targeting linear capabilities would need to invalidate those capabilities in memory as a side effect. Since they cannot be copied, operations must be provided to split linear capabilities apart and merge them back together to allow nested use, for instance in incrementally dividing up the stack. This again requires multiple register writebacks, and is problematic given our compressed capability representation [135], as alignment restrictions on bounds prevent capabilities from being precisely divided.

RISC-V does already contain a mechanism to swap data atomically: CSR instructions both write the value of a general-purpose register to a CSR and read its previous value into a

destination register. This could be used to move a linear capability, for example using the `mscratch` CSR, guaranteeing it is overwritten at its source. For memory operations, one option is to require linear capabilities to be read and written only using (already existing) atomic swap instructions, again guaranteeing that the source capability is overwritten. To avoid circumvention, hardware would need to strip tags of any linear capability not moved in one of these permitted ways: an atomic access or CSR write with the destination register index the same as the source.

This approach would be very slow, since CSR writes incur a full pipeline flush on the microarchitectures discussed in this thesis. Adding the CSR to the set of renamed registers would not solve the problem, as then CSR instructions would again write to two physical registers. Its practicality would therefore depend on how often linear capabilities need to be moved. However, it may serve as an initial stepping stone to evaluate the feasibility of the modifications to software and compilers.

7.5.2 Indirect capabilities

Indirect capabilities do not themselves confer authority, but point to another capability that does. The indirect capability can then be freely copied, but the authority can simply be revoked by invalidating the indirected capability. This would allow a compartment to share a region of memory but quickly reclaim ownership. Another application could be to grant indirect capabilities from `malloc` calls and free by invalidating the indirected capability, preventing use-after-free vulnerabilities.

Indirect capabilities could be implemented in different ways, the primary decision being whether the indirection is opaque to software. One option that requires software to be aware of the indirection is to use a model similar to pairs of load-linked and store-conditional. Dereferences of the indirect capability could require the intermediate capability to be loaded immediately before use. Another option would be to have this occur transparently to software, with hardware performing the chained accesses akin to a page-table walk. A cache of recently used indirections could reduce the required memory accesses, similar to a TLB, although entries would need to be invalidated when the capability is revoked.

7.6 Future work

This chapter has estimated evaluation of performance overheads for revocation on RISC-V based on other work. However, future work would run the CheriBSD revoking kernel on the RISC-V hardware to determine concrete results. Preliminary evaluation was performed on an early prototype of a revocation-supporting kernel, confirming temporal safety is achieved, but a version of CheriBSD with full temporal safety support has only recently

been released. Ideally, this evaluation would include an ablation study of the effects of the individual optimisations discussed in this chapter.

Various of the optimisations discussed remain as future work to implement. The most significant is the investigation of heterogeneous cores for revocation, for example using a Flute core to perform the sweeps while applications run on Toooba cores. Additionally, it is future work to complete the options for `CLoadTags` implementations in the Toooba caches, including the relaxed coherence approach. Finally, an investigation into tag-dependent exceptions on loads in Toooba would reveal whether this is worthwhile to support the load-side barrier.

Future work could also perform further exploration and implementation of the linear and indirect capability types to provide constant-time revocation, albeit with a significant change to the software model.

7.7 Summary

A high-level revocation approach has been developed to bring revocation overheads down to acceptable levels, with a key component being the shadow-map of revoked capabilities.

Optimisations found by others to accelerate this approach to sweeping have been implemented for Toooba, including page-table and TLB modifications, as-user memory accesses, and prefetching. This confirms that application-class cores do not prohibit their implementation. Options have been enumerated for using heterogeneous compute to further reduce (multicore) overheads.

The `CLoadTags` instruction has been implemented for Toooba, again showing it can be applied to such a microarchitecture. This includes a naïve implementation, and a more optimised implementation that required modifying the cache coherence algorithm. Opportunities for further optimisation are elaborated in some detail, including way-restriction and relaxation of `CLoadTags` consistency requirements.

Linear and indirect capabilities have also been considered for RISC-V to allow constant-time revocation.

Overall, while concrete evaluation of temporal safety on RISC-V has not been performed, this chapter provides evidence that revocation is plausible, since much of the architectural support that was required for CHERI MIPS has been implemented. This supports Hypothesis H.4.

Chapter 8

Conclusion

This chapter uses the evidence throughout the thesis to give answers to the hypotheses posed. I then go on to draw overall conclusions from the answers to these hypotheses.

8.1 Answering hypotheses

The hypotheses presented in Section 1.2 are reproduced here. All have seen initial answers from this project. This section gathers the evidence for and against each hypothesis from throughout the thesis.

This work formed part of the DARPA System Security Integration Through Hardware and Firmware (SSITH) program [143]. Where relevant, the results are compared against the objective targets set by the program based on consultation with industry.

Hypothesis H.1

CHERI can be implemented to provide spatial safety for RISC-V microcontrollers, with a small area, power, clock frequency, and performance impact.

CHERI has been validated for RISC-V microarchitectures in a microcontroller context, addressing Hypothesis H.1. Careful management of bounds checking and capability compression are required to produce correct implementations. A library is provided to help future CHERI processor implementations, based on the existing *CHERI-Concentrate* [135] algorithms. While initial implementation reduced frequency in some cases, it was found this could be recovered with moderate optimisation. Security evaluation showed the modifications to be effective for improving spatial safety. Area overheads of 49-62% were observed in the least favourable metric: LUTs. Capability arithmetic logic and the tag controller comprised a large proportion of the overhead. Power overheads were similar, seeing a 36-61% increase in both logic overhead and mean DRAM traffic overheads. Performance overheads spanned 0.19-55% for the benchmarks investigated, with an average across MiBench of approximately 16%. The highest performance overheads were shown

to result at least in part from code generation and software inefficiencies that are not fundamental to capability code.

The final phase of the SSITH program gave targets of 30% area overhead, 10% performance overhead, and 0% power overhead (measured as DRAM traffic) [143]. Some tradeoff between metrics was allowed. The metrics were provided by the SSITH program manager after consultation with a range of possible consumers of the secure processor technology. These targets apply to the entire SoC, including components for DMA and interconnect, rather than just the cores themselves as provided here. This reflects the fact that often processors only occupy a small fraction of their containing SoC as a whole. On this level, the area overheads of CHERI for Piccolo and Flute are only 9% and 17% respectively, well below the target. A small fraction of the remaining budget can be invested to reduce the performance and DRAM traffic overheads to meet the target. This was done by increasing the size of the tag cache, and increasing associativity of select structures: Piccolo's L1 cache and Flute's TLB. This observation reinforces that CHERI overheads can be mitigated by exploring the overall processor design space.

Hypothesis H.2

CHERI can be implemented to provide spatial safety for RISC-V out-of-order superscalar application-class cores, with a small area, power, clock frequency, and performance impact.

To answer Hypothesis H.2, CHERI has been applied to an open, superscalar, out-of-order processor for the first time. Key implementation tradeoffs have been identified, with the initial implementation prioritising simplicity and performance over area in each case. Following discovery of Meltdown Capability Forgery by Fuchs [48], improvements are made to prevent this attack, and the core is audited to confirm the absence of similar vulnerabilities. This motivates a change to the CHERI RISC-V architecture to clear tags rather than raise exceptions on capability monotonicity violations, and corresponding implementation of a single-cycle algorithm to check whether new requested bounds are legal. Once again, evaluation confirms that the extensions mitigate spatial safety attacks. Timing is not significantly affected by CHERI so the target frequency is still comfortably achieved. A 45% area overhead is measured, with capability arithmetic again using a significant proportion of this. A 63% FPGA power overhead is reported, but DRAM traffic overheads appear much lower, with average L2 cache miss overheads of 21%. Performance overheads of 0.68-24% (average 9%) are measured across the SPEC benchmarks. The processor is able to use out-of-order execution to hide some of the instruction overhead. On the other hand, cache pressure increases due to capability metadata and instructions increase the overheads for pointer-heavy benchmarks.

The same SSITH targets applied for the application-class processor as for the microcontrollers. A 34% area overhead (single core) is seen on the SoC granularity for the version of CHERI Toooba evaluated in this thesis, slightly exceeding the 30% target. The SPEC

performance overheads shown meet the 10% performance target. DRAM traffic overhead significantly exceeds the target, and it seems relatively fundamental that capability metadata will increase cache pressure and so miss rates to some extent. Applications requiring low DRAM traffic overhead may therefore have to increase data cache sizes to compensate for this overhead, or simply tolerate the overhead as the price for increased security. It should be noted that negligible DRAM traffic overhead is incurred by the CHERI-enabled processor when running legacy code, allowing the application programmer to make tradeoffs between performance and security.

Hypothesis H.3

The CHERI area, power, and performance impact becomes less significant for larger cores.

The implementations performed for this thesis across the three different sizes of core provide the first evidence for Hypothesis H.3: it has largely been found to hold. The comparison between Piccolo and Flute shows that bigger caches can mask some of the CHERI cycle overhead and that area scaling is favourable due to added components, such as the FPU, that do not need to be changed for CHERI. The comparison between the microcontrollers and Toooba, in turn, show that out-of-order execution can mask some of the cycle overheads. The reduction in area overhead from Flute to Toooba is lower than expected, as the structures added to support the more complex microarchitecture are also extended with capability metadata. Power sees a similar effect, with Toooba's initial Vivado-reported power overhead *exceeding* that of Flute. It is expected that the optimisations identified can mitigate this effect.

Hypothesis H.4

Temporal safety can be implemented efficiently atop CHERI for RISC-V processors.

During the thesis, alongside work from others, overheads for temporal safety atop CHERI have been reduced to levels acceptable for many applications: a 5.8% (1.9% with offloading) geometric mean overhead across the SPEC benchmarks [138, 132]. I have contributed a change in high-level sweeping approach to adopt a shadow-bitmap, giving orders-of-magnitude benefits to the number of capabilities revoked per sweep compared to previous approaches based on CHERI. Furthermore, I have investigated microarchitectural optimisation to improve support for revocation software, including virtual memory optimisations and approaches to finding tags in a sophisticated memory subsystem, leading to the `MESTI` approach to coherence for tags. Optimisations required to reproduce *Cornucopia* [132] can be implemented for RISC-V hardware, though overheads have yet to be evaluated concretely in this context. This work gives evidence to support Hypothesis H.4, but more work is needed to further optimise revocation and evaluate it on RISC-V. Nevertheless, our existing published work [138, 132] already demonstrates that overheads are class-leading compared with prior temporal memory safety schemes.

8.2 Overall conclusions

The results show great promise for CHERI, both in the context of RISC-V and beyond. CHERI has been validated as fit for instantiation in hardware beyond its initial implementation in CHERI MIPS[1]. The validity of the hypotheses shows that CHERI suits a range of microarchitectures, with the overheads seeming to only decrease as the baseline processor becomes more sophisticated. This provides a route to efficient spatial and temporal memory safety. As with many other processor developments, new ideas and incremental improvements within and beyond the CHERI team are expected to bring the overheads ever lower. Historically, this trend has been seen in features such as ever-improving prefetching [85] and branch prediction [121].

Increasing pressure on the software industry to provide security in computer infrastructure, for example via the Global Data Protection Regulation (GDPR) [4], provides incentive to fundamentally improve processor security. Processor vendors have demonstrated that they are willing to invest area and sacrifice performance for security, for instance in their response to speculative execution attacks [14]. This gives reassurance that CHERI can be adopted if its benefits can be shown to outweigh its overheads, as this thesis has suggested.

As well as the implementation itself answering the hypotheses discussed, the Piccolo, Flute, and Toooba cores provide a good basis for the research community to perform further architectural security research. Being open-source, these processors allow others to perform work investigating hardware changes without completely reimplementing CHERI.

As part of the thesis, the first microarchitectures with various new features of the CHERI architecture have been implemented, validating them for both microcontrollers and application-class cores. This principally includes the merged register file, which was found to reduce the complexity in changing the baseline processor's control logic. In addition, clearing tags as opposed to raising exceptions has been motivated and the microarchitectures modified to support the alternative semantics. Tag clearing has therefore been shown to be the advisable approach, especially in light of the speculative side channel implications. Compressed capability-aware instructions, the Sentry mechanism, and the capability encoding mode bit, have also been implemented. The thesis has thus produced recommendations for how CHERI should be implemented going forward, from a microarchitectural perspective.

Finally, the TestRIG framework and QuickCheck Vengine implementation have been elaborated. I have designed and implemented features that significantly improve the utility of QuickCheck Vengine, including recursive templates and smart instruction shrinking. I also designed the first hardware implementations of the DII interface, requiring a new approach to handle pipeline flushes. TestRIG proved an indispensable tool to quickly bring-up the three processors with incremental testing of instructions, and correctness

[1]The concurrently developed Arm Morello prototype provides further assurance of this.

testing of optimisations to the cores.

8.3 Future work

Future work has been highlighted at the end of each chapter, but I summarise those I see as the most significant here.

The main topic for further investigation is optimisation of the processors' area and power: an approximate 50% overhead in these aspects seems higher than ideal, and may be a barrier to adoption in certain contexts. However, these provide only an initial stepping stone in the process of refining the microarchitecture. My work provides insights into where these optimisations could start for CHERI RISC-V. Initial avenues have been identified to reduce the overheads purely in microarchitecture for application-class cores, while optimisation for the microcontrollers may require architectural changes or further refinement to the *CHERI-Concentrate* [135] compression algorithms.

As set out in the scope of the thesis, all evaluation was performed on FPGA. I believe this is a useful first step, and reasonable given the resources available for this project. This approach provides more realism than instruction set simulators, as it ensures microarchitectural feasibility of the techniques used. However, it is not entirely clear how FPGA timing, area, and power overheads would map onto ASIC overheads. Future work would therefore investigate the designs using an ASIC synthesis flow. The lower FPGA frequencies also produce discrepancies in performance measurements, due to the relative speed of DRAM. Future work would go further to mitigate this effect, for example by artificially delaying DRAM transactions.

Finally, enough implementation was carried out to provide initial reassurance that RISC-V hardware, and in particular application-class processors, can be augmented for the *CHERIVoke* [138] approach to achieve temporal safety. However, many further opportunities for hardware acceleration were identified that future work could implement to further improve revocation performance. In addition, future work would explicitly evaluate the performance overheads of revocation using the RISC-V hardware. This would be enabled by the recently released version of CheriBSD with support for temporal safety in the kernel.

Appendix A

CHERI RISC-V Instructions

This section lists all CHERI instructions mentioned in the thesis, briefly explaining their semantics and intended purpose. Complete descriptions can be found in the CHERI architecture document [128]. Due to the change in error semantics throughout the project discussed in Section 5.4, it is left vague whether failed checks raise an exception or clear the resulting tag.

A.1 Capability inspection

These all take a single capability and extract some information about it.

`CGetTag` Returns whether the capability has a valid tag, i.e. whether it has legal provenance. While the tag is checked in hardware whenever a capability is dereferenced, this instruction exposes the information to software.

`CGetAddr` Returns the absolute address of the capability.

`CGetOffset` Returns the offset of the capability, i.e. its address relative to its base.

`CGetPerm` Returns the permissions of the capability, presented as a bit-mask. Permissions include read, write, execute, read-capability, load-capability, and several others.

`CGetFlags` Returns the `flags` field of the capability. Currently, this is just a single bit indicating whether to execute in capability encoding mode when the instruction is installed as PCC.

`CGetBase` Returns the base of the capability.

`CGetLen` Returns the length of the capability, i.e. the difference between its top and base, saturating on overflow.

`CGetType` Returns the `otype` of the capability, with special bit-patterns to indicate unsealed capabilities, the Sentry type, and reserved types.

`CGetSealed` Returns whether the capability is sealed, as a shortcut for `CGetType` followed by comparison with the unsealed type.

A.2 Capability modification

These instructions allow capabilities to be modified in restricted ways. All except `CMove` and `CUnseal` must check their input capability is not sealed.

`CMove` Takes a capability and copies it into the destination register. Unfortunately, none of the other modification instructions can reliably be used to do this simple operation, due to treatment of sealed capabilities.

`CClearTag` Takes a capability and clears its tag. Note that there is no option to set a tag, which would imply forging a capability without valid provenance. `CBuildCap` achieves a similar effect safely.

`CSetAddr` Takes a capability (`cap`) and a new address (`addr`) and sets the address of `cap` to `addr`. The instruction must check that the resulting capability is representable.

`CSetOffset` Takes a capability (`cap`) and a new offset (`off`) and sets the address of `cap` to its base plus `off`. The instruction must check that the resulting capability is representable. This accelerates addressing into arrays and structures, which is typically performed relative to their start.

`CIncOffset` Takes a capability (`cap`) and an increment (`inc`) and increases the address (and therefore the offset) of `cap` by `inc`. The instruction must check that the resulting capability is representable. This accelerates iterating through memory.

`CIncOffsetImm` Form of `CIncOffset` with `inc` a 12-bit immediate rather than a register. This accelerates iterating through memory in statically known increments, such as to loop through an array.

`CAndPerm` Takes a capability (`cap`) and a permissions mask (`mask`), setting the permissions of `cap` to the bitwise `AND` of its original permissions with `mask`. This allows a capability to have its permissions restricted, such as to remove the execute permission on the stack capability. Note that its interface prevents non-monotonic operations without requiring additional checks.

`CSetFlags` Takes a capability (`cap`) and an integer (`flags`) and sets the flags field of `cap` to the value of `flags`. Currently, this is just a single bit indicating whether to execute in capability encoding mode when the instruction is installed as PCC. Note that this can be performed freely without violating monotonicity.

`CSetBounds` Takes a capability (`cap`) and a new length (`len`) and restricts the bounds of `cap`. The new base is the address of `cap`, and the new top is this base plus `len`. Checks are performed that the new bounds are a subset of the old bounds. Due to bounds precision imposed by capability precision, the base may be rounded down, and the top rounded up. The instruction allows capabilities to be subset, for example to restrict a capability for the whole stack to a single stack-allocated variable.

`CSetBoundsImm` Form of `CSetBounds` with `len` a 12-bit immediate rather than a register. This accelerates the common case of statically known small stack allocations.

`CSetBoundsExact` Behaves identically to `CSetBounds`, but also checks that no rounding is required. This allows software to assert that no padding is required.

`CSeal` Takes a capability (`cap`) and an authorising capability (`auth`) and sets the `otype` of `cap` to the address of `auth`. It must check that `auth` gives permission to use its address as a type rather than a memory address (this is signified with a permission bit), and that `auth` is in-bounds. This makes the resulting capability immutable, except for use with `CUnseal` and `CInvoke`. This mechanism enables compartmentalisation as capabilities can be passed safely through untrusted compartments.

`CCSeal` Behaves identically to `CSeal`, but with different behaviours in error cases. This is required to accelerate restoring capabilities of known types with multiple possible authorising capabilities, such as when paging them back in.

`CUnseal` Takes a capability (`cap`) and an authorising capability (`auth`) and unseals `cap`, provided `auth` has the required permissions and bounds. The inverse of `CSeal`, this allows a compartment to recover its sealed capabilities.

A.3 Memory access

The capability-based memory instructions mirror the existing RISC-V memory instructions, but prefixed with `c`. Loads and stores of capabilities themselves use double the access width of `xlen`, such as `cld` on RV32 and `clq` on RV64.

A.4 Control flow

As well as versions—`CJAL` and `CJALR`—of the RISC-V control flow instructions that jump to and link capabilities, RISC-V also provides secure domain transition:

`CInvoke` Takes a sealed code capability and a data capability sealed with the same type, jumping to the code capability and installing the data capability into a fixed register.

Performs various checks, e.g, that the `otypes` match. This allows secure domain transition to a compartment, atomically jumping into its control while giving it access to its data.

A.5 Other instructions

`CTestSubset` Takes two capabilities and checks if the first is a subset of the second in bounds and permissions, returning 1 or 0 to indicate the result. Hardware has to perform subset checks in the background to ensure monotonicity, such as for `CSetBounds`. This instruction exposes this logic to software to allow it to perform custom checks.

`auipcc` Mirrors RISC-V's `auipc` instruction, adding a large immediate to the PCC and returning it. This is used to retrieve globals relative to the PC and in address calculations for long jumps.

`CLoadTags` Loads the capability tags starting from the input address, typically for a cacheline. This accelerates revocation sweeps, as discussed in Section 7.3.

`CClearRegs` An instruction to clear registers in bulk. This is intended to enable fast context switching without leaking data between compartments. A floating-point version is also specified. Note that these instructions were not implemented for the processors discussed in this thesis.

`CBuildCap` Takes two capabilities (`auth` and `bits`), and sets the tag on `bits` provided it can legally be derived from `auth`. This instruction is formally specified to modify `auth` to set every field equal to that of `bits`, clearing the tag if any of these transformations is illegal. This makes the flow of authority clearer. The instruction has applications for dynamic linking and paging.

Appendix B

Benchmarks

To evaluate the impact of the capability modifications, we use several benchmark suites, comparing their execution between the baseline and CHERI-enabled cores. In the microcontroller context, we use CoreMark and MiBench, and for application-class performance we use SPEC. This appendix describes these benchmarks, giving context for the likely effects of capabilities on their performance.

B.1 CoreMark

CoreMark [50] is an embedded benchmark that aims to indicate performance using data structures and algorithms common to most applications. It performs list operations, matrix processing, and state machine processing. The memory footprint is designed to fit within 2 KiB to target small microcontrollers, meaning it easily fits in the caches of all processors discussed in this thesis.

B.2 MiBench

MiBench [55] is a set of benchmarks aimed at covering representative applications for embedded processors, analogous to SPEC for application processors. In particular, we use MiBench2, which ports the benchmarks for IoT devices [84]. These were provided by Galois for the baseline Piccolo and Flute cores, and recompiled by the CHERI team to allow comparison with the CHERI cores.

Unfortunately, not all benchmarks were successfully ported by Galois, so some are excluded from evaluation. These are the `bitcount`, `lzfx`, `overflow`, `patricia`, `regress`, `stringsearch`, `susan`, and `vcflags` benchmarks that fail to compile for the baseline, and `dijkstra`, `picojpeg`, `rsa`, and `sha` that encounter run-time errors on the baseline. `randmath` also does not seem to run correctly on the baseline, executing fewer than ten

instructions per run. All benchmarks that run on the baseline successfully ran compiled for purecap.

To allow discussion of MiBench2 overheads, I give a description of the benchmarks that were run on the cores. Detail is given as required for the discussion in the thesis.

`limits` This tests a variety of fixed-iteration loops with their end conditions expressed differently, presumably mostly targeting branch prediction. There are 13 different loops, each running between 0 and 14 iterations. The end conditions are expressed as `extern` functions, preventing static analysis short-circuiting the loops, but also meaning that calls into very short functions are benchmarked.

`adpcm_encode` Performs ADPCM encoding on a 1.3 MiB input.

`adpcm_decode` Performs ADPCM decoding on a 330 KiB input.

`fft` Performs two Fast Fourier Transforms [97]: one forwards on 128 elements and one inverse on 256 elements.

`crc` Computes a cyclic redundancy check on a short string (10 bytes) in several different ways.

`blowfish` Encrypts and decrypts a 300 KiB string using the Blowfish block cipher.

`rc4` Rivest Cipher 4 (RC4) [62] is a (now insecure) simple fast stream-cipher, on which Wired Equivalent Privacy (WEP) was based. The stream cipher itself maintains 256 bytes of state (initialised based on the key), with each round swapping two bytes and replacing a third based on their values. The simplicity of the stream cipher is such that the encryption loop compiles to a single basic block of approximately 25 RISC-V instructions. This benchmark encrypts a test message of 2,048 bytes using the cipher, then decrypts it again, comparing against the original.

`aes` This runs the AES [33] block cipher in ECB and CBC modes: encrypting and decrypting with each for message lengths of up to 64 bytes.

`qsort` This computes the floating-point distance from the origin of 11,240 3D integer co-ordinates, then calls the C standard library `qsort` function to perform a quicksort based on the computed distance.

B.3 SPEC

The SPEC benchmark suite [29] is designed to cover a range of applications to assess the performance of high-end cores. We use the SPEC CINT 2006 suite.

Compiling the benchmarks for CHERI (with full protection) varies in difficulty across the benchmarks. Some require only small changes, while others would be large projects, for example those that themselves perform compilation: `gcc` and `perlbench`. `mcf` also requires further modification to support CHERI, so is excluded. The remainder had already been adapted for CHERI for CHERI MIPS, so were available to run on CHERI RISC-V.

A brief description of the characteristics of each benchmark that was run gives context for the benchmarking results. These are based on the official descriptions given by SPEC [29]. Particular focus is given to code patterns most relevant to capabilities such as memory access patterns and pointer density.

`bzip2` Tests compression and decompression using the common `bzip2` file-compression algorithm. The benchmarks therefore consist of linear accesses through memory to access the file, as well as accesses to internal data structures to exploit repeating patterns.

`h264ref` Performs AVC video compression and decompression. Similar to `bzip2`, this will involve linear memory accesses to the files and accesses to smaller internal data structures, so not many pointers.

`astar` Runs variants of the A* algorithm for pathfinding in game AI. There are two variants operating on 2D space, which will be pointer-light, and a variant adapted to work on graphs, which will be pointer-heavy.

`hmmer` Runs database searches and statistical tests on databases representing DNA sequence alignments. This involves repeated linear searching and very few pointers.

`sjeng` Analyses positions in chess using an engine. To represent the game tree, the code is likely relatively pointer-dense.

`gobmk` Analyses positions in the game Go using an engine. Similar to `sjeng`, this will involve storing and regularly traversing a pointer-heavy game tree.

`omnetpp` Performs discrete event simulation of an Ethernet network. This will require a pointer-heavy structure describing the edges of the network graph.

`xalancbmk` Transforms XML documents into different formats. Since the XML documents are represented as trees of nodes, this is relatively pointer-heavy.

`libquantum` Simulates a quantum computer to run Shor's factoring algorithm on the input. This mostly involves intense matrix computations, so is pointer-light.

Appendix C

TestRIG

This appendix contains our TestRIG paper, pending publication. It is the result of collaboration between myself and the listed authors.

Randomized Testing of RISC-V CPUs using Direct Instruction Injection

Alexandre Joannou, Peter Rugg, Jonathan Woodruff, Franz A. Fuchs, Marno van der Maas, Matthew Naylor, Michael Roe, Robert N. M. Watson, Peter G. Neumann, Simon W. Moore

I. Introduction

TestRIG (Testing with Random Instruction Generation) is a testing framework for RISC-V implementations. The RISC-V community has standardized a formal model of the architecture in the Sail language[1]. Ideally, a RISC-V implementor could formally prove equivalence between their implementation and the Sail model, but proof tools are not yet sufficiently automated to be routinely used on the whole-processor level. As a pragmatic compromise, we use TestRIG to check equivalence between the model and an implementation by generating random instruction sequences, executing the same sequences on the model and the implementation under test, and comparing execution traces (tandem execution). This approach does not prove equivalence but can demonstrate divergence, and is usable in all stages of development.

TestRIG uses the RISC-V Formal Interface (RVFI) standard to observe the change in state after each instruction of the implementation under test, and uses a novel technique that we are calling Direct Instruction Injection (DII) for test injection.

In normal program execution, the next instruction is fetched from program memory at an address determined by the program counter. With Direct Instruction Injection, the next instruction to be executed is provided by the test harness, regardless of the CPU's program counter.

We are not testing completed, fabricated chips. Rather, we are comparing executable formal models, software ISA simulators and simulated execution of hardware designs. This requires us to instrument the CPU design with an additional interface for Direct Instruction Injection used by the test harness during tandem verification.

We have added the Direct Instruction Injection interface to the Sail RISC-V formal model[2], and to two high-performance emulators: Spike[3], and QEMU[4]. We have also instrumented four RISC-V processor implementations with RVFI-DII, spanning from embedded to superscalar implementations. We have used TestRIG to test many standard RISC-V extensions, and the experimental CHERI security extension.

We found TestRIG to be easier to use than unit tests, and to give more thorough test coverage. It is effective at detecting not just issues in instruction semantics, but also the in pipeline and the data caches. As a result, TestRIG has completely replaced our instruction-set level unit testing for development.

[1]https://github.com/riscv/sail-riscv
[2]https://github.com/CTSRD-CHERI/sail-riscv
[3]https://github.com/riscv-software-src/riscv-isa-sim
[4]https://github.com/CTSRD-CHERI/qemu

II. The Dream – Model-based Verification

Architectural extensions are traditionally specified starting with a prose specification, and then four implementations are produced largely independently:

1) Assembler
2) Executable model (simulator)
3) Instruction-level unit-test suite
4) Hardware implementation

While this ensemble of implementation efforts is laborious when done once, its greatest cost lies in discouraging design exploration; design changes require consistency among five independent code bases.

A formal, executable instruction-set architecture (ISA) specification can greatly simplify this workflow. We use the Sail [3] domain-specific language, which features human-readable syntax. Sail excerpts serve as pseudocode in our ISA documents [14]. Sail also produces a simulator (item 2), and will eventually provide verification (item 3) and the assembler (item 1) to be derived from it automatically.

A. Model-based Formal Verification

Formal verification tools for RISC-V have often used the RVFI tracing (see Section IV) interface[5] along with tools like Cadence's JasperGold to prove that a series of traces from a simple HDL model is equivalent to a series of traces from a pipelined HDL implementation. Unfortunately, these tools can handle only simple pipelines, and require specialist knowledge. As a result, the formal-verification approach does not yet replace functional testing for entire processors.

B. Model-based Random Testing

While formally proving equivalence for complex microarchitectures has been elusive, pragmatic tools have used other ways to detect divergence from a model. These approaches cannot *prove* equivalence between a formal model and an implementation but can refute it with counterexamples.

For example, directed-random test-sequence generation has been used to debug pipeline and memory bugs, as well as to uncover unexpected divergences in implementation behavior [1], [12]. There exist multiple test generators for RISC-V, e.g., RISC-V RTG [13], but RISCV-DV[6] remains the most advanced such sequence generator for RISC-V, and it works well for these use cases, particularly where detailed traces can

[5]https://github.com/SymbioticEDA/riscv-formal
[6]https://github.com/google/riscv-dv

be compared. RISCV-DV generates assembly programs, ready to be converted to in-memory images for execution. RISCV-DV includes a number of test generators for RV32IMAFDC and RV64IMAFDC – including support for page-table interactions, privileged CSR use, and handling traps/interrupts. These generated test programs are executed on both a golden model and a processor in development. A RISCV-DV test framework would typically detect a divergence by comparing the execution traces.

Although randomly generating tests is a promising approach, it can have several drawbacks:

- Automatically generated counterexamples can be long and convoluted, while hand-written tests can be made short and easy to understand.
- The generator must ensure that useful instructions are found at the target of each randomly generated branch.

Automated reduction of failing test cases has previously been used in software testing. For example, *C-Reduce* [10] can take a program that triggers a bug in a C compiler and reduce it to a minimal example that triggers the bug.

PyH2P [7] applies automated test case reduction randomly generated RISC-V instruction sequences. PyH2P often produces test sequences that contain less than 5 instructions, with every instruction being meaningful for reproducing the error. Nevertheless, PyH2P has three shortcomings:

1) PyH2P does not perform full trace comparison with its internal PyMTL3 model, but only with final memory and register state.
2) PyH2P has difficulty shrinking through branches, as it must produce a valid in-memory program.
3) PyH2P does not use community-standard interfaces that have been proven across a range of implementations.

PyH2P points in an encouraging direction, and TestRIG matures the approach, proposing a standardized communication interface so that verification engines (VEngines), models, and implementations are interchangeable and can be improved independently. Additionally, instruction injection allows straightforward shrinking of sequences with branches. This has allowed us to completely replace instruction-level unit tests for the sophisticated CHERI extension [15], greatly improving both productivity and assurance, and enabling extension of an array of simulators and processors more efficiently than the CHERI implementations on MIPS or ARM.

III. TestRIG

Figure 1 gives an overview of the modular TestRIG ecosystem. In TestRIG, an interactive Verification Engine (VEngine) stimulates RISC-V implementations over RVFI-DII sockets, which are detailed in Section IV. An RVFI-DII compatible RISC-V implementation can reset, consume instruction sequences, and report execution traces via its RVFI-DII interface. A VEngine can drive one or more RVFI-DII compatible implementations; a VEngine might have an internal RISC-V model, similar to PyH2P, or could drive two independent implementations and compare their RVFI traces, as we have done with QCVEngine, which is presented in Section V. VEngine instruction sequences could be loaded from disk, generated randomly, or produced with interactive architecture-driven state-space exploration.

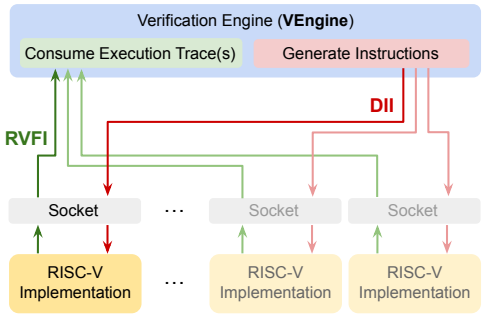

Fig. 1. An illustrative example of the TestRIG ecosystem with a Verification Engine communicating with any two RISC-V implementations over sockets. The Verification Engine injects instruction sequences and compares the execution traces until it finds a divergence.

The RVFI-DII bytestream interface allows models and implementations written in various languages to communicate through widely supported networking sockets. QCVEngine is written in Haskell, and the Sail RISC-V model is written in the Sail domain-specific language (either interpreted by an OCaml program or compiled into C). Spike and QEMU are RISC-V simulators written in C and C++. TestRIG also supports hardware implementations like RVBS, Ibex, Piccolo, Flute, and Tooba, which are written in either SystemVerilog or Bluespec. RVBS[7] is a reference implementation, Ibex[8] and Piccolo[9] are simple 32-bit implementations, Flute[10] is a 5-stage in-order pipeline processor implementing RV64, and Tooba[11] is a RISC-V 64-bit superscalar out-of-order processor.

Participants in the TestRIG ecosystem are expected to be identical in every architecturally visible way. Besides a RVFI-DII interface, TestRIG requires 8 MiB of memory accessible at address 0x80000000 (all other addresses returning an access fault), and must support resetting to a known state (zeroed registers, known default values for RISC-V control and status registers, zeroed 8 MiB of memory) upon injection of a "reset" DII packet.

IV. RVFI-DII

To participate in the TestRIG verification ecosystem, implementations must be extended with RVFI-DII instrumentation. To ease development, we distribute data structures and libraries in several languages to facilitate RVFI-DII connections over TCP ports.

The RISC-V Formal Interface (RVFI), specified by Claire Wolf, is an existing trace format for formal verification using symbolic instructions. RVFI exposes select architecturally significant signals such as the instruction encoding and any

[7] https://github.com/CTSRD-CHERI/RVBS
[8] https://github.com/CTSRD-CHERI/ibex
[9] https://github.com/CTSRD-CHERI/Piccolo
[10] https://github.com/CTSRD-CHERI/Flute
[11] https://github.com/CTSRD-CHERI/Tooba

memory address or value, as well as the numbers and values of the operand and writeback registers.

TestRIG extends RVFI with Direct Instruction Injection (DII). DII is for instruction input, RVFI is for trace output, and RVFI-DII supports full interactive verification. Interactive verification enables automated simplification and shrinking, as discussed in Section V-A. Existing RISC-V cores that implement RVFI can be augmented to participate in the TestRIG ecosystem by implementing DII, and conversely RVFI-DII designs may benefit from RVFI formal verification tooling.

Not all architectural updates are reported in the RVFI interface, e.g., floating-point registers and extended CHERI capability registers. While this is a limitation, PyH2P relies only on final register and memory state and is still able to usefully detect divergence. We found that occasional instructions that move unexposed values into RVFI-visible state could produce sufficiently succinct counterexamples. This strategy was also used in RVFI formal verification efforts.

An RVFI interface exports internal signals of an RTL design, or internal variables of a simulator or emulator. For more complex RTL designs, such as pipelined or superscalar microarchitectures, extracting the appropriate values may require preserving state for an RVFI report in a commit/write-back stage that did not previously have access to them. Extending the superscalar Toooba core for RVFI-DII required two extra records for each instruction in the Reorder Buffer.As these records are present only when built for simulation with RVFI, this is not a physical overhead for the design.

DII directly specifies the instruction sequence expected in the output trace, and does not associate instructions with memory addresses. This requires custom pipeline instrumentation, but enables greatly simplified sequence generation and shrinking, as the program counter does not affect the instruction stream.

A DII interface receives a reset command followed by a sequence of instructions. A Bluespec implementation of this interface is shown below:

```
typedef struct {
  Bool     rvfi_cmd;  // Instruction or reset command?
  Bit#(10) rvfi_time; // Time to inject instruction
  Bit#(32) rvfi_insn; // Instruction word (32/16 bit)
} RVFI_DII_Instruction
```

For an emulator, this interface simply replaces each fetched instruction with an encoding from the DII queue. For RTL designs, DII support is more complex. An RTL design can remove the instruction cache entirely (but not address translation of the PC, which is architecturally visible) to ensure maximal pipeline packing, or can exercise the instruction cache and replace the bytes of the instruction after they have been fetched. RISC-V compressed instructions present another choice: to substitute picked instructions before decode, or inject 16-bit instruction fragments from DII to exercise the picking logic. The simple single-issue design of Piccolo and Flute enabled us to replace the cache entirely with a DII queue that delivered one instruction every cycle, either compressed or uncompressed. For superscalar Toooba, we began with unmodified instruction-cache access, substituting the vector of picked instructions before decoding. In an effort to debug instruction picking itself, we later moved to bypassing the instruction cache and providing 16-bit instruction fragments to the pipeline, relying on the instruction picker and decode to reconstitute the correct DII instruction sequence.

Canceled instructions present a further challenge to DII. Synchronization is required when instructions are dropped in the pipeline, as RVFI-DII requires a single RVFI trace entry for each DII instruction injected. While adding RVFI-DII to Flute, we arrived at a mature design that attaches a sequence ID to each RVFI instruction and carries it with the PC through the pipeline. Instruction Fetch actively requests each instruction ID from the DII sequence (as with PC requests to the cache), allowing pipeline redirects to work naturally. We adapted this approach to Toooba by adding superscalar fetch and assigning IDs to compressed instruction fragments. This more capable DII unit is available in our RVFI-DII libraries[12], and has been backported to Flute. While DII instrumentation may appear daunting, we have found that beginning with this mature strategy greatly reduces both implementation effort and design disturbance. In retrospect, the few hours invested in this implementation have greatly streamlined the otherwise much longer testing phase.

V. QuickCheck VEngine

Our TestRIG Verification Engine, *QCVEngine*, leverages Haskell's QuickCheck library [5]. Due to the simplicity of Direct Instruction Injection execution, which decouples the instruction stream from control flow, QCVEngine can use unmodified QuickCheck utilities to generate, compare, and shrink instruction sequences.

QuickCheck receives a function with a pass/fail return value, and generates inputs in search of a failure. To facilitate this, we construct a function that receives a list of instructions, sends these over two DII sockets, collects RVFI traces back from these sockets, asserts that they match, and returns the result. We then provide a set of generators of *arbitrary* instruction sequences that are used by QuickCheck to produce inputs to this function.

We use convenience functions to define instructions in a syntax closely resembling the RISC-V ISA manual, and provide tailored generators for each instruction field to promote register reuse. QuickCheck automatically discovers and uses these generators through the type system and uses them to construct arbitrary instruction sequences. We also provide targeted generators for simple subsets of the instruction set, as well as generators that leverage templates of varying complexity to reach deeper states, including virtual memory mappings and cache conflicts. Templates are a common tool for random test generators; for example, IBM's Genesys-Pro [1] is built on templates to intelligently solve for desired deep states.

A. Smart Shrinking

While Direct Instruction Injection allows us to primarily rely on QuickCheck's builtin shrinking strategies, we augmented these with *smart shrinking* functions that not only eliminate instructions, but intelligently transform them to simplify the sequence.

[12]https://github.com/CTSRD-CHERI/BSV-RVFI-DII

Once a counterexample is found by QCVEngine, QuickCheck uses a builtin list-shrinking function that removes sequences from the list and tests again, hoping to eliminate instructions with no relevance to the errant behavior. Illustratively, here is an initial counterexample found for an artificial hardware bug where the LSB of the `add` instruction's result (but not `addi`'s) is stuck at zero:

```
addi x7, x4, 123 # Generate odd immediate
addi x5, x3, 42  # Generate even immediate
addi x6, x7, 0   # Move x7 to x6
xori x1, x5, 745 # Irrelevant
add  x1, x5, x6  # Perform buggy add
```

The builtin list shrinking results in:

```
addi x7, x4, 123 # Generate odd immediate
addi x6, x7, 0   # Move x7 to x6
add  x1, x5, x6  # Perform buggy add
```

The middle instruction can also be eliminated if the final `add` takes register 7 as an operand directly. To automate this functionality, we further augment shrinking to intelligently propagate an instruction's output register to future input operands. This allows another pass of list-shrinking to further reduce the counterexample:

```
addi x7, x4, 123 # Generate odd immediate
add  x1, x5, x7  # Perform buggy add
```

We also add a library of simplifications to be used during shrinking that eliminate esoteric instructions used to perform mundane functions that might distract from the root cause. For example, memory operations often trap; thus, we might attempt to simplify a memory operation to an *ecall*, an instruction that only traps, to make the error more obvious.

Any shrinking or simplification is safe to try for model-based testing; any change that still diverges is kept. In rare circumstances, the shrinking may reveal an alternative bug, obscuring the original, but still producing a useful result.

B. Sequence Import/Export

Instruction traces can be converted to (and from) a human-readable format both for terminal reporting, and for reading and writing trace files – individually or in bulk from a directory. This has enabled us to collect a library of regression tests to quickly check all previous counterexamples. Unlike hand-written tests with assertions, these do not require maintenance, as expected behavior updates naturally with the model as the instruction set evolves. We have also used this feature to replay recorded test-suite examples (including *riscv-tests* and *RISCV-DV*), adding full trace-equivalence check with shrinking. This feature has also allowed us to capture traces of an operating system booting on the model implementation, which we could then use to aid bring-up of the same operating system on implementations, with instruction shrinking rapidly highlighting any problems.

C. Non-shrinkable Sequences

Sequences can be annotated as non-shrinkable. This has been used to force initialization to cover divergences in initial state. For example, one implementation did not initialize floating-point registers, which produced trivial counterexamples. A non-shrinkable initialization sequence allowed us to progress to interesting divergences in exception conditions and rounding modes.

D. Assertions

Sequences can include assertions – e.g., that the value written by the previous instruction was non-zero. These make it possible to fail without a divergence. Unusually, sequences with assertions do not require tandem verification to discover a failure, and we have used these to test the limits of implementation-defined behavior.

VI. Evaluation

A. A Coverage Study

Architectural coverage is the first metric for basic verification. We evaluated coverage of the RISC-V architecture using *sailcov*[13], which measured how many branches of the RISC-V Sail model were explored during a run. We compared QCVEngine over TestRIG with the RISC-V test suite (*riscv-tests*[14]) and the RISCV-DV generator.

For our coverage study we conduct two runs of each testing framework (TestRIG, riscv-tests, and RISCV-DV) for RV32IMC and RV64IMAFDCZicsr. For RV32IMC, we take the Sail RISC-V model coverage of the I, M, and C extension instructions as well as the coverage of the general-purpose registers. For RV64IMAFDCZicsr, we measure the coverage of I, M, A, F, D, C, and CSR instructions as well as the coverage of the general-purpose and floating-point registers. For riscv-tests, we measure the coverage of the Sail RISC-V model running the test binaries. For RISCV-DV, we produce TestRIG traces from the Spike simulator executing the tests and replay them through RVFI-DII while measuring the coverage of the Sail RISC-V model. For QCVEngine, we configure it with the two architecture strings and let it run 500 sequences of each generator. The RV32IMC results are similar across all three testing frameworks, indicating that QCVEngine can support a suitable alternative to unit testing and torture testing, at least with respect to breadth of coverage. The RV64IMAFDCZicsr results more variance, but all three remain comparable except in floating point register coverage, in which RISCV-DV excels and CSR instructions in which TestRIG excels. QCVEngine chooses from a subset of floating-point registers to increase the probability of operand reuse, at a cost to overall coverage. As failures based on register number are rare, we have made a trade-off between increasing the probability of finding violations that require multiple permutations and increasing the overall FD register coverage.

B. Counterexample Complexity

Counterexample complexity is another useful metric. Our archive of QCVEngine traces comprises 3509 shrunken counterexamples discovered during development of our CHERI processor extensions. Figure 2 shows the distribution of our

[13] https://github.com/rems-project/sail/tree/sail2/sailcov
[14] https://github.com/riscv-software-src/riscv-tests

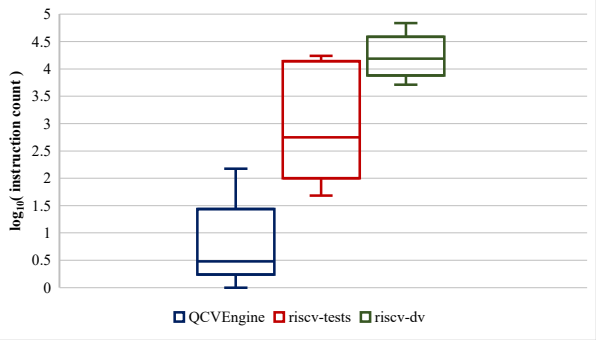

Fig. 2. Counterexample size complexity

archive of counterexample lengths versus riscv-test and riscv-dv trace lengths, which do not allow shrinking. The median value is 3 instructions, and the third quartile is only 5 instructions. The median for riscv-tests 561 instructions, which is more than 3 times the maximum counterexample found by QCVEngine, and the median riscv-dv sequence is 15339. QCVEngine's small counterexample size is facilitated by Direction Instruction Injection and smart shrinking as described previously. Single digit counterexample length greatly accellerated our discovery of failures and development of fixes compared to even a traditional unit test suite.

VII. Illustrative Cases

TestRIG is useful for a broader range of verification than instruction-level unit tests and improves productivity in all cases. Architectural bugs, which are traditionally targeted by hand-written test suites, are usually discovered quickly with TestRIG. Microarchitectural mistakes, such as register forwarding or pipeline-flush problems, are also discovered quickly and deterministically in TestRIG, but are difficult to anticipate and target in unit-test suites. Memory mistakes, such as cache bugs or memory speculation failures, have also been discovered efficiently with targeted generators, and are notoriously hard to discover using static unit tests. Finally, TestRIG has found *unexpected interactions*, where architectural features interact in unforeseen ways , while unit-test suites are unlikely to test these cases. We have applied TestRIG not only to RISC-V, but also to CHERI-RISC-V (our security extension, noted below).

A. CHERI Introduction

Capability Hardware Enhanced RISC Instructions (CHERI) is a security extension of conventional Instruction Set Architectures that adds *capabilities* – unforgeable and bounded tokens. A capability is a fat pointer [8] containing the address and metadata including permissions and bounds information. Furthermore, validity of capabilities is ensured by a hidden tag.A capability authorizes access to a region of memory, and no data or instruction access is possible without a valid capability. Furthermore, all capability operations are monotonic and therefore cannot increase the privileges a capability grants. As a result, CHERI enforces spatial safety, enables temporal safety, and supports fine-grained software compartmentalization.

B. Architectural Bug

When developing the compressed encoding of CHERI capabilities, we had a bug that unnecessarily cleared the tag of a pointer when setting an address that wrapped the address space. This bug was found with this shrunken counterexample:

```
cSetBoundsImmediate x3, x1, 1106 # Set a short bound
cIncOffsetImmediate x2, x4, -197 # Small negative integer
cSetAddr x4, x3, x2 # Set the integer as the address
cGetTag x1, x4 # XXX Untagged
```

While this case may have been covered in an extensive unit-test suite composed at significant effort, our TestRIG generator required only a list of CHERI instructions to produce a counterexample far more compact than most hand-written tests.

C. Microarchitectural Bug

We have also used TestRIG for discovering microarchitectural vulnerabilities in transient execution. One such generator produces a sequence of arbitrary instructions, followed by an assertion that no additional cache misses were counted, which would indicate a transient violation of the capability system. The following shrunken counterexample demonstrated a vulnerability in cSetBoundsImmediate:

```
.noshrink
... # initialize counters
... # bound x31 to 8 bytes
.shrink
# Illegally increase the bound on a pointer
cSetBoundsImmediate x31, x4, 797
# Load through this illegal capability
lb.cap lb.cap x31, x31[0]
... # Delay for counter to propagate
.noshrink
csrrs x30, hpmcounter3 (0xc03), x0 # Read L1 cache miss
.assert rd_wdata == 0x0 ""
```

Because CHERI only allows bounds to be reduced, the cSetBoundsImmediate instruction is illegal and throws an exception due to attempting to enlarge the bounds. Nevertheless, the capability that would have been produced is forwarded in the pipeline during the flush and causes a cache fill that could lead to side-channel attacks.

This sequence uses both *noshrink* and *assert*. Noshrink is required to initialize the state of the counters so that the final assertion on the L1 cache miss counter is deterministic.

D. Cache Bug

We received Flute as a working in-order RV64G design, and discovered that the data cache was direct-mapped and 4 KiB, rather than 2-way associative and 8 KiB – as specified. An experiment with parameters confirmed that the 2-way cache could not boot the operating system. This bug had not been found with the unit-test suite, so we used a generator that constructs addresses within the TestRIG memory range (see Section III), as well as random loads and stores. This generator quickly discovered the bug with the following shortened sequence, after 42 tests and 20 rounds of shrinking:

```
lui  x1, 262148
slli x1, x1, 1
lui x20, 262148 # Value used as data
ori  x3, x1, 1  # A page address
lui  x2, 262148
slli x2, x2, 1  # The same page address
lhu  x4, x3[1]  # Load from address
sh  x20, x2[2]  # Store to an overlapping byte
lhu  x2, x3[1]  # Divergence on reloading
```

The final sequence contains only three memory operations: two loads with a single store in between, all to overlapping addresses. This counterexample was found less than 10 seconds into the TestRIG run, and was fixed within the hour. The fix is reproduced below:

```
Bool hit = False;
for (Integer way = 0; way < num_ways; way = way + 1)
  begin
    Bool hit_at_way  = (tags[way].state != EMPTY)
                    && (tags[way].tag == pa_tag);
    hit = hit || hit_at_way;
    if (hit_at_way) // XXX This line was missing!
      way_hit = fromInteger (way);
  end
```

While this bug was trivial to resolve with a TestRIG counterexample, it had escaped the entire development process of the Flute processor. It was not found with the RISC-V unit-test suite and was overwhelmingly difficult to debug from a full software trace.

VIII. Future of TestRIG

Despite having an array of models, simulators, and implementations supporting RVFI-DII, the generators of our initial TestRIG verification engine, QCVEngine, are still rudimentary.

QCVEngine Generators: The Haskell infrastructure in QCVEngine supports rich and complex generators. However, the generators for virtual memory, cache testing, and floating-point operations can be enriched with more intelligent directed-random templates for reaching deeper states.

Memory Concurrency Testing: TestRIG should support memory-model testing. RVFI-DII instruction streams injected with specified timestamps into multiple shared memory cores should allow precise stimulation of concurrency behaviors. These would require a more advanced verification engine that tests RVFI traces not only for equivalence, but also against higher-level memory-model semantics – as in AXE [9].

Pipeline Performance: Similarly, a higher-level model of pipeline scheduling and performance could be used to analyze the timing of instruction traces committed in a pipeline to discover performance bugs and track performance improvements. The high level of control possible with direct instruction injection should enable precise detection of performance anomalies.

Model-derived Engine: TestRIG's modular design enables a variety of engines to drive RVFI-DII compatible RISC-V implementations. With QCVEngine, the test maintenance burden is greatly simplified, but not entirely eliminated. Past experience suggests that even deep-state tests can be automatically generated from a model specification, as with IBM's Genesys [1]. Previous CHERI work used tests generated from a formal model of our CHERI-MIPS ISA (written in the L3 [6] specification language), compiling from L3 to HOL4, and then using constraint solving to automatically generate instruction sequences to reach a desired state without triggering undefined behavior. This approach has also been applied to the CHERI ARM Morello instruction set starting from a Sail model [4], [11]. Brian Campbell, a key contributor to this work, has also begun on a Sail-OCaml VEngine with direct access to the data structures of our Sail RISC-V model. This eliminates independent encodings in the VEngine, and we expect this approach to be taken further to automate generation of templates that target specific deep states in the architectural model using constraint solving.

IX. Counterexample-driven Development

TestRIG's model-based testing leads to counterexample-driven development, an advancement over test-driven development, a widely known technique of software engineering. Typical test-driven development for processor design requires a basic working design before architectural unit tests can be used. Counterexample-driven development using TestRIG can automatically provide reduced stimulus for the most basic features and can carry development all the way to advanced interactions. The CHERI extension to Ibex was a striking example. After extending Ibex with RVFI-DII support, a summer intern was able to independently add full CHERI functionality to Ibex in a month, due to the tight cycle of reduced counterexamples provided by QCVEngine.

X. Conclusion

We have collated all the current TestRIG-compatible implementations and verification engines into the open-source TestRIG repository[15]. This repository includes documentation that has been followed and improved multiple times by new users. TestRIG accelerates development at all stages, providing a tighter debugging loop than we have experienced in any other processor development paradigm. We expect TestRIG to lead the way to a standardized testing framework for RISC-V that leverages instrumentation of open implementations, to greatly simplify verification. Such a framework improves upon traditional instruction-set-level unit testing in every way, and subsumes specialized random test generators into a cohesive community of easy-to-use verification tools.

References

[1] Allon Adir et al. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2), 2004.

[2] Merav Aharoni, Sigal Asaf, Laurent Fournier, Anotoly Koifman, and Raviv Nagel. FPgen–a test generation framework for datapath floating point verification. In *Eighth IEEE International High-Level Design Validation and Test Workshop*, pages 17–22, 2003.

[3] Alasdair Armstrong et al. Isa semantics for armv8-a, risc-v, and cheri-mips. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[4] Thomas Bauereiss et al. Verified security for the morello capability-enhanced prototype arm architecture. Technical report, University of Cambridge, Computer Laboratory, 2021.

[5] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4), 2011.

[6] Anthony Fox. Directions in ISA specification. In *ITP*, 2012.

[7] Shunning Jiang et al. PyH2: Using PyMTL3 to create productive and open-source hardware testing methodologies. *IEEE Design & Test*, 38(2), 2020.

[15] https://github.com/CTSRD-CHERI/TestRIG

[8] Trevor Jim et al. Cyclone: A safe dialect of C. In *ATEC 2002*, Berkeley, CA, USA. USENIX.
[9] Matthew Naylor et al. A consistency checker for memory subsystem traces. In *FMCAD 2016*. IEEE.
[10] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *33rd ACM SIGPLAN Conference om Programming Language Design and Implementation (PLDI 2012)*, 2012.
[11] Peter Sewell. Engineering with full-scale formal architecture: Morello, cheri, armv8-a, and risc-v. In *FMCAD 2021*. IEEE.
[12] Shajid Thiruvathodi and Deepak Yeggina. A random instruction sequence generator for arm based systems. In *2014 15th International Microprocessor Test and Verification Workshop*. IEEE.
[13] Dai Duong Tran, Thi Giang Truong, Truong Giang Do, and The Duc Do. Risc-v random test generator. In *2021 15th International Conference on Advanced Computing and Applications (ACOMP)*, pages 150–155, November 2021.
[14] Robert N. M. Watson et al. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, October 2020.
[15] Jonathan Woodruff et al. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA 2014, Minneapolis, MN, USA*. IEEE.

Bibliography

[1] Kathirgamar Aingaran, Sumti Jairath, Georgios Konstadinidis, Serena Leung, Paul Loewenstein, Curtis McAllister, Stephen Phillips, Zoran Radovic, Ram Sivaramakrishnan, David Smentek, and Thomas Wicki. "M7: Oracle's Next-Generation Sparc Processor". In: *IEEE Micro* 35.2 (2015), pp. 36–45. DOI: 10.1109/MM.2015.35.

[2] Sam Ainsworth and Timothy M. Jones. "MarkUs: Drop-in use-after-free prevention for low-level languages". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 578–591. DOI: 10.1109/SP40000.2020.00058.

[3] Periklis Akritidis et al. "Cling: A memory allocator to mitigate dangling pointers". In: *19th USENIX Security Symposium (USENIX Security 10)*. 2010.

[4] Jan Philipp Albrecht. "How the GDPR will change the world". In: *Eur. Data Prot. L. Rev.* 2 (2016), p. 287.

[5] Hesham Almatary. *CHERI compartmentalisation for embedded systems*. Tech. rep. UCAM-CL-TR-976. University of Cambridge, Computer Laboratory, Nov. 2022. DOI: 10.48456/tr-976. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-976.pdf.

[6] Ross Anderson, Chris Barton, Rainer Böhme, Richard Clayton, Michel JG Van Eeten, Michael Levi, Tyler Moore, and Stefan Savage. "Measuring the cost of cybercrime". In: *The economics of information security and privacy*. Springer, 2013, pp. 265–300.

[7] Arm. *AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite*. 2013. URL: https://developer.arm.com/documentation/ihi0022/e/.

[8] Arm. *Arm big.LITTLE*. URL: https://www.arm.com/technologies/big-little.

[9] Arm. *Armv8.5-A Memory Tagging Extension*. 2019. URL: https://developer.arm.com/documentation/102925/0100.

[10] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, KE Gray, R Norton-Wright, C Pulte, S Flur, and Peter Sewell. *The Sail instruction-set semantics specification language*. 2021. URL: https://raw.githubusercontent.com/rems-project/sail/sail2/manual.pdf.

[11] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. "Chisel: Constructing hardware in a Scala embedded language". In: *DAC Design Automation Conference 2012*. 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[12] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems". In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*. 2002, pp. 73–78. DOI: 10.1145/774789.774805.

[13] Arash Baratloo, Navjot Singh, and Timothy Tsai. "Libsafe: Protecting critical elements of stacks". In: *White Paper* (1999). URL: http://www.research.avayalabs.com/project/libsafe.

[14] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. "Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks". In: *USENIX Security*. Vol. 11. 2022.

[15] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert NM Watson, and Peter Sewell. "Verified security for the Morello capability-enhanced prototype Arm architecture". In: *European Symposium on Programming*. Springer, Cham. 2022, pp. 174–203.

[16] Emery D. Berger and Benjamin G. Zorn. "DieHard: Probabilistic Memory Safety for Unsafe Languages". In: *SIGPLAN Not.* 41.6 (June 2006), pp. 158–168. ISSN: 0362-1340. DOI: 10.1145/1133255.1134000. URL: https://doi.org/10.1145/1133255.1134000.

[17] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. "Address obfuscation: An efficient approach to combat a broad range of memory error exploits". In: *12th USENIX Security Symposium (USENIX Security 03)*. 2003.

[18] Anton Bikineev, Michael Lippautz, and Hannes Payer. *Retrofitting Temporal Memory Safery on C++*. May 2022. URL: https://security.googleblog.com/2022/05/retrofitting-temporal-memory-safety-on-c.html.

[19] Matt Bishop, Michael Dilger, et al. "Checking for race conditions in file accesses". In: *Computing systems* 2.2 (1996), pp. 131–152.

[20] Hand-J. Boehm, Alan Demers, and Mark Weiser. *A garbage collector for C and C++*. URL: https://hboehm.info/gc/.

[21] University of California. *RISC-V Tests*. 2014. URL: https://github.com/riscv-software-src/riscv-tests.

[22] University of Cambridge. *Ibex*. URL: https://github.com/ctsrd-cheri/ibex.

[23] University of Cambridge and SRI International. *CheriBSD*. URL: https://github.com/ctsrd-cheri/cheribsd.

[24] University of Cambridge and SRI International. *CheriFreeRTOS*. URL: https://github.com/CTSRD-CHERI/FreeRTOS-mirror.

[25] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A systematic evaluation of transient execution attacks and defenses". In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 249–266.

[26] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. "Hardware Support for Fast Capability-based Addressing". In: *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VI. San Jose, California, USA: ACM, 1994, pp. 319–327. ISBN: 0-89791-660-3. DOI: 10.1145/195473.195579. URL: http://doi.acm.org/10.1145/195473.195579.

[27] Christopher Celio, David Patterson, and Krste Asanovic. "The Berkeley Out-of-Order Machine (BOOM) Design Specification". In: *University of California, Berkeley* (2016).

[28] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *SIGPLAN Not.* 46.4 (May 2011), pp. 53–64. ISSN: 0362-1340. DOI: 10.1145/1988042.1988046. URL: https://doi.org/10.1145/1988042.1988046.

[29] Standard Performance Evaluation Corporation. *CINT2006 (Integer Component of SPEC CPU 2006)*. URL: https://www.spec.org/cpu2006/CINT2006/.

[30] The MITRE Corporation. *Common Weakness Enumeration*. 2022. URL: https://cwe.mitre.org.

[31] The MITRE Corporation. *CWE-415: Double Free*. 2018. URL: https://cwe.mitre.org/data/definitions/415.html.

[32] The MITRE Corporation. *CWE-416: Use After Free*. 2018. URL: https://cwe.mitre.org/data/definitions/416.html.

[33] Joan Daemen and Vincent Rijmen. *AES proposal: Rijndael*. 1999.

[34] Thurston HY Dang, Petros Maniatis, and David Wagner. "Oscar: A Practical {Page-Permissions-Based} Scheme for Thwarting Dangling Pointers". In: *26th USENIX security symposium (USENIX security 17)*. 2017, pp. 815–832.

[35] *DARPA Announces First Bug Bounty Program to Hack SSITH Hardware Defenses*. URL: https://www.darpa.mil/news-events/2020-06-08a.

[36] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. "CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: ACM, 2019, pp. 379–393. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304042. URL: http://doi.acm.org/10.1145/3297858.3304042.

[37] Jack B. Dennis and Earl C. Van Horn. "Programming Semantics for Multiprogrammed Computations". In: *Commun. ACM* 9.3 (Mar. 1966), pp. 143–155. ISSN: 0001-0782. DOI: 10.1145/365230.365252. URL: https://doi.org/10.1145/365230.365252.

[38] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. "Hardbound: Architectural Support for Spatial Safety of the C Programming Language". In: *SIGPLAN Not.* 43.3 (Mar. 2008), pp. 103–114. ISSN: 0362-1340. DOI: 10.1145/1353536.1346295. URL: http://doi.acm.org/10.1145/1353536.1346295.

[39] Alexander Dörflinger, Mark Albers, Benedikt Kleinbeck, Yejun Guan, Harald Michalik, Raphael Klink, Christopher Blochwitz, Anouar Nechi, and Mladen Berekovic. "A Comparative Survey of Open-Source Application-Class RISC-V Processor Implementations". In: *Proceedings of the 18th ACM International Conference on Computing Frontiers*. CF '21. Virtual Event, Italy: Association for Computing Machinery, 2021, pp. 12–20. ISBN: 9781450384049. DOI: 10.1145/3457388.3458657. URL: https://doi.org/10.1145/3457388.3458657.

[40] Alexander Dörflinger, Mark Albers, Benedikt Kleinbeck, Yejun Guan, Harald Michalik, Raphael Klink, Christopher Blochwitz, Anouar Nechi, and Mladen Berekovic. "A Comparative Survey of Open-Source Application-Class RISC-V Processor Implementations". In: *Proceedings of the 18th ACM International Conference on Computing Frontiers*. CF '21. Virtual Event, Italy: Association for Computing Machinery, 2021, pp. 12–20. ISBN: 9781450384049. DOI: 10.1145/3457388.3458657. URL: https://doi.org/10.1145/3457388.3458657.

[41] Durumeric, Kasten, Adrian, Halderman, Bailey, Li, Weaver, Amann, Beekman, Payer, and Paxson. "The Matter of Heartbleed". In: *ACM Conference on Internet Measurement* (2014).

[42] Michel J. G. van Eeten and Johannes M. Bauer. *Economics of Malware*. 2008. DOI: https://doi.org/https://doi.org/10.1787/241440230621. URL: https://www.oecd-ilibrary.org/content/paper/241440230621.

[43] Islam Elsadek and Eslam Yahya Tawfik. "RISC-V Resource-Constrained Cores: A Survey and Energy Comparison". In: *2021 19th IEEE International New Circuits and Systems Conference (NEWCAS)*. 2021, pp. 1–5. DOI: 10.1109/NEWCAS50681.2021.9462781.

[44] Márton Erdős, Sam Ainsworth, and Timothy M. Jones. "MineSweeper: A "Clean Sweep" for Drop-in Use-after-Free Prevention". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 212–225. ISBN: 9781450392051. DOI: 10.1145/3503222.3507712. URL: https://doi.org/10.1145/3503222.3507712.

[45] Lawrence G. Esswood. "CheriOS: Designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor". PhD thesis. University of Cambridge, July 2020.

[46] Matthew Field. "WannaCry cyber attack cost the NHS £92m as 19,000 appointments cancelled". In: *The Telegraph* (Aug. 2018). URL: https://www.telegraph.co.uk/technology/2018/10/11/wannacry-cyber-attack-cost-nhs-92m-19000-appointments-cancelled/.

[47] RISC-V Foundation. *RISC-V Base Cache Management Operation ISA Extensions*. Nov. 2021. URL: https://github.com/riscv/riscv-CMOs.

[48] Franz A Fuchs. "Analysis of Transient-Execution Attacks on the out-of-order CHERI-RISC-V Microprocessor Toooba". In: *Masters Thesis* (2021).

[49] Franz A Fuchs, Jonathan Woodruff, Simon W Moore, Peter G Neumann, and Robert NM Watson. "Developing a Test Suite for Transient-Execution Attacks on RISC-V and CHERI-RISC-V". In: *Fifth Workshop on Computer Architecture Research with RISC-V* (June 2021).

[50] Shay Gal-On and Markus Levy. "Exploring coremark a benchmark maximizing simplicity and efficacy". In: *The Embedded Microprocessor Benchmark Consortium* (2012).

[51] Dapeng Gao and Tom Melham. "End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers". In: *2021 Formal Methods in Computer Aided Design (FMCAD)*. IEEE. 2021, pp. 24–33.

[52] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. "Le Temps Des Cerises: Efficient Temporal Stack Safety on Capability Machines Using Directed Capabilities". In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (Apr. 2022). DOI: 10.1145/3527318. URL: https://doi.org/10.1145/3527318.

[53] Enes Göktaş, Angelos Economopoulos, Robert Gawlik, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. "Bypassing clang's safestack for fun and profit". In: *Black Hat Europe* (2016), p. 21.

[54] R. Grisenthwaite. "Arm Morello Evaluation Platform -Validating CHERI-based Security in a High-performance System". In: *2022 IEEE Hot Chips 34 Symposium (HCS)*. Los Alamitos, CA, USA: IEEE Computer Society, Aug. 2022, pp. 1–22. DOI: 10.1109/HCS55958.2022.9895591. URL: https://doi.ieeecomputersociety.org/10.1109/HCS55958.2022.9895591.

[55] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

[56] Norm Hardy. "The Confused Deputy: (or why capabilities might have been invented)". In: *ACM SIGOPS Operating Systems Review* 22.4 (1988), pp. 36–38.

[57] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Amsterdam: Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.

[58] Bluespec Inc. *Bluespec Compiler*. URL: https://github.com/B-Lang-org/bsc.

[59] Bluespec Inc. *Toooba*. URL: https://github.com/bluespec/Toooba.

[60] Dejice Jacob and Jeremy Singer. "Capability Boehm: Challenges and Opportunities for Garbage Collection with Capability Hardware". In: *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE 2022. Virtual, Switzerland: Association for Computing Machinery, 2022, pp. 81–87. ISBN: 9781450392518. DOI: 10.1145/3516807.3516823. URL: https://doi.org/10.1145/3516807.3516823.

[61] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. "Cyclone: a safe dialect of C." In: *USENIX Annual Technical Conference, General Track*. 2002, pp. 275–288.

[62] Poonam Jindal and Brahmjit Singh. "RC4 Encryption-A Literature Survey". In: *Procedia Computer Science* 46 (2015). Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace & Island Resort, Kochi, India, pp. 697–705. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.02.129. URL: https://www.sciencedirect.com/science/article/pii/S1877050915001933.

[63] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos. "Efficient Tagged Memory". In: *2017 IEEE International Conference on Computer Design (ICCD)*. Nov. 2017, pp. 641–648. DOI: 10.1109/ICCD.2017.112.

[64] Alexandre Joannou. "High-performance memory safety - Optimizing the CHERI capability machine". PhD thesis. University of Cambridge, May 2018.

[65] Nicolas Joly, Saif ElSherei, and Saar Amar. *Security analysis of CHERI ISA*. Oct. 2020. URL: https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf.

[66] Nikolai Joukov, Aditya Kashyap, Gopalan Sivathanu, and Erez Zadok. "An Electric Fence for Kernel Buffers". In: *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability*. StorageSS '05. Fairfax, VA, USA: Association for Computing Machinery, 2005, pp. 37–43. ISBN: 159593233X. DOI: 10.1145/1103780.1103786. URL: https://doi.org/10.1145/1103780.1103786.

[67] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 29–42. DOI: 10.1109/ISCA.2018.00014.

[68] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. "Hardware-based Always-On Heap Memory Safety". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 1153–1166. DOI: 10.1109/MICRO50266.2020.00095.

[69] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "SeL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[70] Paul C Kocher. "On certificate revocation and validation". In: *International conference on financial cryptography*. Springer. 1998, pp. 172–177.

[71] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[72] S. Kostya, S. Evgenii, S. Aleksey, T. Vlad, and V. Dmitry. *Hwasan: An aarch64-specific compiler-based tool*. 2018. URL: https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.

[73] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. "DangSan: Scalable Use-after-Free Detection". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: Association for Computing Machinery, 2017, pp. 405–419. ISBN: 9781450349383. DOI: 10.1145/3064176.3064211. URL: https://doi.org/10.1145/3064176.3064211.

[74] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. "Code-Pointer Integrity". In: *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. Association for Computing Machinery and Morgan & Claypool, 2018, pp. 81–116. ISBN: 9781970001839. URL: https://doi.org/10.1145/3129743.3129748.

[75] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. "Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-Based Security". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany: Association for Computing Machinery, 2013, pp. 721–732. ISBN: 9781450324779. DOI: 10.1145/2508859.2516713. URL: https://doi.org/10.1145/2508859.2516713.

[76] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. "Preventing use-after-free with dangling pointers nullification." In: *NDSS*. 2015.

[77] Kyung-Suk Lhee and Steve J. Chapin. "Buffer overflow and format string overflow vulnerabilities". In: *Software: Practice and Experience* 33.5 (2003), pp. 423–460. DOI: https://doi.org/10.1002/spe.515. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.515. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.515.

[78] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. "PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1901–1915. ISBN: 9781450394505. DOI: 10.1145/3548606.3560598. URL: https://doi.org/10.1145/3548606.3560598.

[79] Samuel Lindemer, Gustav Midéus, and Shahid Raza. "Real-time thread isolation and trusted execution on embedded RISC-V". In: *Proceedings of the International Workshop on Secure RISC-V Architecture Design Exploration (SECRISC-V)*. 2020.

[80] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[81] Daiping Liu, Mingwei Zhang, and Haining Wang. "A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1635–

1648. ISBN: 9781450356930. DOI: 10.1145/3243734.3243826. URL: https://doi.org/10.1145/3243734.3243826.

[82] Chris A Mack. "Fifty years of Moore's law". In: *IEEE Transactions on semiconductor manufacturing* 24.2 (2011), pp. 202–207.

[83] Nicholas D. Matsakis and Felix S. Klock II. "The Rust Language". In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641. DOI: 10.1145/2692956.2663188. URL: http://doi.acm.org/10.1145/2692956.2663188.

[84] *MiBench2 for SSITH*. URL: https://github.com/ctsrd-cheri/mibench2.

[85] Sparsh Mittal. "A Survey of Recent Prefetching Techniques for Processor Caches". In: *ACM Comput. Surv.* 49.2 (Aug. 2016). ISSN: 0360-0300. DOI: 10.1145/2907071. URL: https://doi.org/10.1145/2907071.

[86] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. "Watchdog: Hardware for safe and secure manual memory management and full memory safety". In: *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 2012, pp. 189–200. DOI: 10.1109/ISCA.2012.6237017.

[87] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. "CETS: Compiler Enforced Temporal Safety for C". In: *SIGPLAN Not.* 45.8 (May 2010), pp. 31–40. ISSN: 0362-1340. DOI: 10.1145/1837855.1806657. URL: https://doi.org/10.1145/1837855.1806657.

[88] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: ACM, 2009, pp. 245–258. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542504. URL: http://doi.acm.org/10.1145/1542476.1542504.

[89] Matthew Naylor and Simon Moore. "A generic synthesisable test bench". In: *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE. 2015, pp. 128–137.

[90] George C. Necula, Scott McPeak, and Westley Weimer. "CCured: Type-safe Retrofitting of Legacy Code". In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '02. Portland, Oregon: ACM, 2002, pp. 128–139. ISBN: 1-58113-450-9. DOI: 10.1145/503272.503286. URL: http://doi.acm.org/10.1145/503272.503286.

[91] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100. ISSN: 0362-1340. DOI: 10.1145/1273442.1250746. URL: http://doi.acm.org/10.1145/1273442.1250746.

[92] Robert HB Netzer and Barton P Miller. "What are race conditions? Some issues and formalizations". In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.1 (1992), pp. 74–88.

[93] Peter G Neumann. "Fundamental trustworthiness principles". In: *New Solutions for Cybersecurity* (2018).

[94] RISC-V Community News. *Xuantie IOMMU from T-Head for RISC-V*. URL: https://riscv.org/blog/2022/04/xuantie-iommu-from-t-head-for-risc-v-chong-ren-alibaba-cloud.

[95] R. Nikhil. "Bluespec System Verilog: efficient, correct RTL from high level specifications". In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 2004, pp. 69–70. DOI: 10.1109/MEMCOD.2004.1459818.

[96] Gene Novark and Emery D. Berger. "DieHarder: Securing the Heap". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: Association for Computing Machinery, 2010, pp. 573–584. ISBN: 9781450302456. DOI: 10.1145/1866307.1866371. URL: https://doi.org/10.1145/1866307.1866371.

[97] Henri J Nussbaumer. "The fast Fourier transform". In: *Fast Fourier Transform and Convolution Algorithms*. Springer, 1981, pp. 80–111.

[98] Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. "Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure Progress Report". In: *Software Security — Theories and Systems*. Ed. by Mitsuhiro Okada, Benjamin C. Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 133–153. ISBN: 978-3-540-36532-7.

[99] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2018).

[100] Mark S. Papamarcos and Janak H. Patel. "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories". In: *Proceedings of the 11th Annual International Symposium on Computer Architecture*. ISCA '84. New York, NY, USA: Association for Computing Machinery, 1984, pp. 348–354. ISBN: 0818605383. DOI: 10.1145/800015.808204. URL: https://doi.org/10.1145/800015.808204.

[101] Mark Patton, Eric Gross, Ryan Chinn, Samantha Forbis, Leon Walker, and Hsinchun Chen. "Uninvited Connections: A Study of Vulnerable Devices on the Internet of Things (IoT)". In: *2014 IEEE Joint Intelligence and Security Informatics Conference*. 2014, pp. 232–235. DOI: 10.1109/JISIC.2014.43.

[102] Vaughan Pratt. "Anatomy of the Pentium bug". In: *TAPSOFT '95: Theory and Practice of Software Development*. Ed. by Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 97–107. ISBN: 978-3-540-49233-7.

[103] Qualcomm. *Pointer Authentication on ARMv8.3*. Jan. 2019. URL: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.

[104] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. "Precise Garbage Collection for C". In: *Proceedings of the 2009 International Symposium on Memory Management*. ISMM '09. Dublin, Ireland: Association for Computing Machinery, 2009, pp. 39–48. ISBN: 9781605583471. DOI: 10.1145/1542431.1542438. URL: https://doi.org/10.1145/1542431.1542438.

[105] Arm Research. *University of Cambridge: CHERI blossoms*. URL: https://community.arm.com/arm-research/b/articles/posts/university-of-cambridge-cheri-blossoms.

[106] Microsoft Research. *What's the smallest variety of CHERI?* URL: https://msrc-blog.microsoft.com/2022/09/06/whats-the-smallest-variety-of-cheri/.

[107] Alexander Richardson. *Complete spatial safety for C and C++ using CHERI capabilities*. Tech. rep. UCAM-CL-TR-949. University of Cambridge, Computer Laboratory, June 2020. DOI: 10.48456/tr-949. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-949.pdf.

[108] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. "Return-Oriented Programming: Systems, Languages, and Applications". In: *ACM Trans. Inf. Syst. Secur.* 15.1 (Mar. 2012), 2:1–2:34. ISSN: 1094-9224. DOI: 10.1145/2133375.2133377. URL: http://doi.acm.org/10.1145/2133375.2133377.

[109] J. H. Saltzer and M. D. Schroeder. "The protection of information in computer systems". In: *Proceedings of the IEEE* 63.9 (Sept. 1975), pp. 1278–1308. ISSN: 0018-9219. DOI: 10.1109/PROC.1975.9939.

[110] Hiroshi Sasaki, Miguel A. Arroyo, M. Tarek Ibn Ziad, Koustubha Bhat, Kanad Sinha, and Simha Sethumadhavan. "Practical Byte-Granular Memory Blacklisting Using Califorms". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 558–571. ISBN: 9781450369381. DOI: 10.1145/3352460.3358299. URL: https://doi.org/10.1145/3352460.3358299.

[111] Herbert Schildt. *The Annotated ANSI C Standard: American National Standard for Programming Languages: C*. 1990, pp. xvi + 219. ISBN: 0-07-881952-0.

[112] F. B. Schneider. "Least privilege and more [computer security]". In: *IEEE Security Privacy* 1.5 (Sept. 2003), pp. 55–59. ISSN: 1540-7993. DOI: 10.1109/MSECP.2003.1236236.

[113] David Seal. *ARM architecture reference manual*. Pearson Education, 2001. Chap. A6.

[114] Abderrahmane Sensaoui, Oum-El-Kheir Aktouf, David Hely, and Stephane Di Vito. "An In-depth Study of MPU-Based Isolation Techniques". In: *Journal of Hardware and Systems Security* 3 (4 Dec. 2019), pp. 365–381. DOI: 10.1007/s41635-019-00078-6. URL: https://doi.org/10.1007/s41635-019-00078-6.

[115] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "{AddressSanitizer}: A Fast Address Sanity Checker". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 2012, pp. 309–318.

[116] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. "On the Effectiveness of Address-Space Randomization". In: *Proceedings of the 11th ACM Conference on Computer and Communications Security*. CCS '04. Washington DC, USA: Association for Computing Machinery, 2004, pp. 298–307. ISBN: 1581139616. DOI: 10.1145/1030083.1030124. URL: https://doi.org/10.1145/1030083.1030124.

[117] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yunheung Paek, et al. "CRCount: Pointer invalidation with reference counting to mitigate use-after-free in legacy C/C++". In: *Network and Distributed System Security Symposium*. 2020, pp. 1–15.

[118] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. "FreeGuard: A Faster Secure Heap Allocator". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2389–2403. ISBN: 9781450349468. DOI: 10.1145/3133956.3133957. URL: https://doi.org/10.1145/3133956.3133957.

[119] Nicholas Wei Sheng Sim. "Strengthening memory safety in Rust: exploring CHERI capabilities for a safe language". In: *Masters Thesis* (2020).

[120] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. "StkTokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation Using Linear Capabilities". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 19:1–19:28. ISSN: 2475-1421. DOI: 10.1145/3290332. URL: http://doi.acm.org/10.1145/3290332.

[121] James E Smith. "A study of branch prediction strategies". In: *25 years of the international symposia on Computer architecture (selected papers)*. 1998, pp. 202–215.

[122] L. Szekeres, M. Payer, T. Wei, and D. Song. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. May 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[123] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. "SecFuzz: Fuzz-testing Security Protocols". In: *Proceedings of the 7th International Workshop on Automation of Software Test*. AST '12. Zurich, Switzerland: IEEE Press, 2012, pp. 1–7. ISBN: 978-1-4673-1822-8. URL: http://dl.acm.org/citation.cfm?id=2663608.2663610.

[124] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. "CODOMs: Protecting Software with Code-Centric Memory Domains". In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 469–480. ISSN: 0163-5964. DOI: 10.1145/2678373.2665741. URL: https://doi.org/10.1145/2678373.2665741.

[125] Philipp Wagner. *An update on Ibex, our microcontroller-class CPU core*. June 2019. URL: https://www.cl.cam.ac.uk/~jrrk2/blog/2019/06/an-update-on-ibex-our-microcontroller-class-cpu-core/.

[126] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Mar. 2019.

[127] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume II: Privileged ISA*. Nov. 2021.

[128] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. Tech. rep. UCAM-CL-TR-951. University of Cambridge, Computer Laboratory, Oct. 2020. DOI: 10.48456/tr-951. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf.

[129] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. *CHERI C/C++ Programming Guide*. Tech. rep. UCAM-CL-TR-947. University of Cambridge, Computer Laboratory, June 2020. DOI: 10.48456/tr-947. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf.

[130] Robert N. M. Watson, Jonathan Woodruff, Michael Roe, Simon W. Moore, and Peter G. Neumann. *Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks*. Tech. rep. UCAM-CL-TR-916. University of Cambridge, Computer Laboratory, Feb. 2018. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-916.pdf.

[131] WCCFTech. *x86 and Arm rival, RISC-V architecture ships 10 billion cores*. URL: https://wccftech.com/x86-arm-rival-risc-v-architecture-ships-10-billion-cores.

[132] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. "Cornucopia: Temporal Safety for CHERI Heaps". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 608–625. DOI: 10.1109/SP40000.2020.00098.

[133] Emmett Witchel, Josh Cates, and Krste Asanović. "Mondrian Memory Protection". In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS X. San Jose, California: ACM, 2002, pp. 304–316. ISBN: 1-58113-574-2. DOI: 10.1145/605397.605429. URL: http://doi.acm.org/10.1145/605397.605429.

[134] Claire Wolf. *End-to-end formal ISA verification of RISC-V processors with riscv-formal*. URL: http://www.clifford.at/papers/2017/riscv-formal.

[135] J. Woodruff, A. Joannou, H. Xia, B. Davis, P. G. Neumann, R. N. M. Watson, S. Moore, A. Fox, R. Norton, D. Chisnall, and A. Fox. "CHERI Concentrate: Practical Compressed Capabilities". In: *IEEE Transactions on Computers* (2019), pp. 1–1. ISSN: 0018-9340. DOI: 10.1109/TC.2019.2914037.

[136] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. "The CHERI capability model: Revisiting RISC in an age of risk". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. June 2014, pp. 457–468. DOI: 10.1109/ISCA.2014.6853201.

[137] Jonathan D. Woodruff. *CHERI: A RISC capability machine for practical memory safety*. Tech. rep. UCAM-CL-TR-858. University of Cambridge, Computer Laboratory, July 2014. DOI: 10.48456/tr-858. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-858.pdf.

[138] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. "CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 545–557. ISBN: 9781450369381. DOI: 10.1145/3352460.3358288. URL: https://doi.org/10.1145/3352460.3358288.

[139] Hongyan Xia, Jonathan Woodruff, Hadrien Barral, Lawrence Esswood, Alexandre Joannou, Robert Kovacsics, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alexander Richardson, Simon W. Moore, and Robert N. M. Watson. "CheriRTOS: A Capability Model for Embedded Devices". In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 2018, pp. 92–99. DOI: 10.1109/ICCD.2018.00023.

[140] AMD Xilinx. *UltraScale Architecture and Product Data Sheet: Overview*. Nov. 2022. URL: https://docs.xilinx.com/v/u/en-US/ds890-ultrascale-overview.

[141] AMD Xilinx. *Vivado Design Suite User Guide: Power Analysis and Optimization (UG907)*. Oct. 2022. URL: https://docs.xilinx.com/r/en-US/ug907-vivado-power-analysis-optimization.

[142] Shengjie Xu, Wei Huang, and David Lie. "In-Fat Pointer: Hardware-Assisted Tagged-Pointer Spatial Memory Safety Defense with Subobject Granularity Protection". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 224–240. ISBN: 9781450383172. DOI: 10.1145/3445814.3446761. URL: https://doi.org/10.1145/3445814.3446761.

[143] Lok Yan. *System Security Integration Through Hardware and Firmware (SSITH)*. May 2021. URL: https://www.darpa.mil/program/ssith.

[144] Yves Younan. "FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers." In: *NDSS*. 2015.

[145] S. Zhang, A. Wright, T. Bourgeat, and A. Arvind. "Composable Building Blocks to Open up Processor Design". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Aug. 2018, pp. 68–81. DOI: 10.1109/MICRO.2018.00015.

[146] Tong Zhang, Dongyoon Lee, and Changhee Jung. "BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 631–644. ISBN: 9781450362405. DOI: 10.1145/3297858.3304017. URL: https://doi.org/10.1145/3297858.3304017.

[147] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. "Debloating Address Sanitizer". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 4345–4363. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen.