UNIVERSITY OF
CAMBRIDGE

Capability-Based Memory Protection for Scalable Vector Processing

Samuel Stark

Clare College

This dissertation is submitted on June 2022 for the degree of Master of Philosophy

Declaration

I Samuel Stark of Clare College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total Word Count: 12,944 words

Word Count Method: `texcount -1 -sum -merge -q chapters.tex`
(Covers pages 13-63)

Samuel Stark
June 2022

Abstract

Capability-Based Memory Protection for Scalable Vector Processing

Samuel Stark

[TODO1 My abstract ...]

Acknowledgements

[TODO2 My acknowledgements ...]

Simon Moore

Jessica Clarke

Jon Woodruff

Karl Mose

Gavin Stark

Elizabeth Yallop (diagram inspection)

Contents

1 Introduction **13**

1.1 Motivation 14

1.2 Hypotheses and Aims 15

2 Background **17**

2.1 RISC-V 17

2.2 A brief history of vector processing 18

2.3 The RVV vector model 18

2.3.1 `vtype` 19

2.3.2 `vl` and `vstart` — Prestart, body, tail 20

2.3.3 Masking — Active/inactive elements 21

2.3.4 Exception handling 22

2.3.4.1 Imprecise vector traps 22

2.3.4.2 Precise vector traps 22

2.3.4.3 Other modes 23

2.3.5 Summary 24

2.4 RVV memory instructions 25

2.4.1 Segmented accesses 25

2.4.2 Unit and Strided accesses 26

2.4.3 Unit fault-only-first loads 27

2.4.4 Indexed accesses 28

2.4.5 Unit whole-register accesses 29

2.4.6 Unit bytemask accesses 29

2.5 Previous RVV implementations 30

2.6 CHERI 31

2.6.1 CHERI-RISC-V ISA 32

2.6.2 Instruction changes 32

2.6.3 Capability and Integer encoding mode 33

2.6.4 Pure-capability and Hybrid compilation modes 34

2.6.5 Capability relocations 34

3 Hardware emulation investigation 35
3.1 Developing the emulator . . . 35
3.1.1 Emulating CHERI . . . 36
3.1.1.1 `rust-cheri-compressed-cap` . . . 36
3.1.1.2 Integrating into the emulator . . . 37
3.1.2 Emulating vectors . . . 38
3.1.2.1 Decoding phase . . . 38
3.1.2.2 Fast-path checking phase . . . 39
3.1.2.3 Execution phase . . . 39
3.1.2.4 Integer vs. Capability encoding mode . . . 40
3.2 Fast-path calculations . . . 40
3.2.1 Possible fast-path outcomes . . . 40
3.2.2 Masked accesses . . . 41
3.2.3 Unit accesses . . . 42
3.2.4 Strided accesses . . . 42
3.2.5 Indexed accesses . . . 43
3.3 Going beyond the emulator . . . 43
3.3.1 Misaligned accesses . . . 43
3.3.2 Atomicity of accesses/General memory model . . . 44
3.3.3 Relaxed access ordering and precise traps . . . 44
3.4 Testing hypotheses . . . 45
Hypothesis H-1 - Feasibility . . . 45
Hypothesis H-2 - Fast-path checks . . . 45

4 The CHERI-RVV software stack 49
4.1 Compiling vector code . . . 49
4.1.1 Required command-line options . . . 49
4.1.2 Automatic vectorization . . . 50
4.1.3 Vector intrinsics . . . 50
4.1.4 Inline assembly . . . 51
4.1.5 vs. Arm SVE . . . 52
4.2 Compiling vector code with CHERI-Clang . . . 54
4.2.1 Required command-line options . . . 54
4.2.2 Adapting vector assembly instructions to CHERI . . . 55
4.2.3 Adapting vector intrinsics to CHERI . . . 55
4.2.4 Storing scalable vectors on the stack . . . 56
4.3 Testing hypotheses . . . 57
Hypothesis H-3 - Compiling legacy code for integer mode . . . 57
Hypothesis H-4 - Converting legacy code to pure-capability code . . . 58

Hypothesis H-5 - Saving vectors on the stack 59

Hypothesis H-6 - Running CHERI-RVV code in a multiprocessing system . . . 59

4.4 Recommended changes for CHERI-Clang 60

5 Capabilities-in-vectors 61

5.1 Emulator changes 61

5.2 Hardware implications 61

5.3 Possible software improvements 61

5.4 Testing hypotheses 61

Hypothesis H-7 - Holding capabilities in vectors 61

Hypothesis H-8 - Sending capabilities between vectors and memory 61

Hypothesis H-9 - Manipulating capabilities in vectors 61

6 Evaluation 63

6.1 Vectorized memcpy test and results 63

6.2 Capabilities-in-vector test and results 63

6.3 Conclusion 63

7 References 65

A Building the `riscv-gnu-toolchain` with vector support 69

CHAPTER 1

Introduction

Since 2010, the Cambridge Computer Lab (in association with SRI) has been developing the CHERI[1] architecture extension, which improves the security of any given architecture by checking all memory accesses in hardware. The core impact of CHERI, on a hardware level, is that memory can no longer be accessed directly through raw addresses, but must pass through a *capability*[1]. Capabilities are unforgeable tokens that grant fine-grained access to ranges of memory. Instead of generating them from scratch, capabilities must be *derived* from another capability with greater permissions. For example, a capability giving read-write access to an array of structures can be used to create a sub-capability granting read-only access to a single element. This vastly reduces the scope of memory-related security issues, such as buffer overflows**[TC1[2]]**, and creates interesting opportunities for software compartmentalization[2].

Industry leaders have recognized the value CHERI provides. Arm Inc have manufactured the Morello System-on-Chip, based on their Neoverse N1 CPU, which incorporates CHERI capabilities into the Armv8.2 ISA. While this represents a great step forward, there are still elements on the SoC that haven't fully embraced CHERI (e.g. the GPU), and architecture extensions that haven't been investigated in the context of CHERI. One such example is Arm's Scalable Vector Extension (introduced in Armv8.2 but not included in Neoverse N1), which is designed to remain in use well into the future[3]. Supporting this and other scalable vector ISAs in CHERI is essential to CHERI's long-term relevance.

In the context of modern computer architecture, vector processing is the practice of dividing a large hardware register into a *vector* of multiple *elements* and executing the same operation on each element in a single instruction[3]. This data-level parallelism can drastically increase throughput, particularly for arithmetic-heavy programs. **[TODO3 explain Scalable vectors]** However, before computing arithmetic, the vectors must be populated with data.

[1] Capability Hardware Enhanced RISC Instructions

[2] **something something heartbleed?**

[3] This is a SIMD (Single Instruction Multiple Data) paradigm.

1.1 Motivation

Modern vector implementations all provide vector load/store instructions to access a whole vector's worth of memory. These range from simple contiguous accesses (where all elements are next to each other), to complex indexed accesses (where each element loads from a different location based on another vector). They can also have per-element semantics, e.g. "elements must be loaded in order, so if one element fails the preceding elements are still valid"**[TC2[4]]**. If CHERI CPUs want to benefit from vector processing's increased performance and throughput, they must support those instructions at some level. But adding CHERI's bounds-checking to the mix may affect these semantics, and could impact performance (e.g. checking each element's access in turn may be slow).

Vector memory access performance is more critical than one may initially assume, because vectors are used for more than just computation. A prime example is `memcpy`: for `x86_64`, `glibc` includes multiple versions of the function[5] taking advantage of vector platforms, then selects one to use at runtime[6]. These implementations are written in assembly and heavily optimized. If the memory accesses are hitting the cache, a few extra cycles of bounds-checking for each access could actually make a noticeable difference.

`memcpy` also raises the important question of how the vector model interacts with capabilities. In non-CHERI processors, `memcpy` will copy pointers around in memory without fuss. For a CHERI-enabled vector processor to support this, it would need to be able to load/store capabilities from vectors without violating any security guarantees. This may require more guarantees than otherwise necessary - for example, each vector register likely needs to be as large or larger than a single capability.

To explore this topic, we chose to focus on the RISC-V Vector extension (shortened to RVV throughout this dissertation, and specified in [4]). As of November 2021 this has been ratified by RISC-V International[7], and will be RISC-V's standard vector instruction set moving forward. Choosing it has two key benefits. Firstly, the CHERI project maintains three open-source cores (Piccolo, Flute, and Tooba**[TC3[8]]**) implementing CHERI-RISC-V, none of which support vector processing. Studying RVV will allow reference "CHERI-RVV" implementations to be built for these cores. Secondly, RVV is a *scalable* vector model. This has more potential roadblocks than a fixed-length vector model, and investigating them here will make life easier if Arm wish to integrate their Scalable Vector Extension with CHERI later down the road.

[4] **RISC-V V fault-only-first**

[5] It appears memcpy is implemented as a copy of memmove.

[6] `sysdeps/x86_64/multiarch/ifunc-memmove.h` in `bminor/glibc` on GitHub

[7] `https://wiki.riscv.org/display/HOME/Recently+Ratified+Extensions`

[8] **cheri risc-v page?**

1.2 Hypotheses and Aims

The goal of this project is to investigate the impact of, and the roadblocks for, integrating a scalable vector architecture with CHERI's memory protection system. In particular, we focus on integrating RVV with the CHERI-RISC-V ISA, with the aim of enabling a future CHERI-RVV implementation and informing the approach for a future CHERI Arm SVE implementation.

The investigation was carried out by designing and testing a CHERI-RVV emulator written in Rust, but that is only a single implementation. To show that RVV can be integrated with CHERI-RISC-V for a wide range of processors, we use information gathered from the emulator to prove **[TODO4 five]** hypotheses (Table 1.1). Hyps. H-1 and H-2 take the point of view of the hardware, considering basic feasibility and potential performance issues. Hyps. H-3 to H-6 examine the current CHERI-RVV software stack, checking if adding CHERI to vector programs causes compatibility problems. Hyps. H-7 to H-9 considers capabilities-in-vectors: the conditions under which vector registers can hold capabilities, and the conditions under which vectorized memory accesses and other instructions can manipulate them.

Along with testing these hypotheses, this document serves as a practical explanation of the following:

- The changes to RISC-V required by CHERI-RISC-V
- The RISC-V-V vector model, and all memory-related instructions
- How to compile CHERI-RVV code for a baremetal platform (with no operating system)
- How to execute CHERI-RVV code compiled for a baremetal platform
- The limitations of the current CHERI-RVV software stack

[TODO5 fill out this para more, have an actual end sentence]

Hardware Hypotheses — Chapter 3	
H-1	It is possible to use CHERI capabilities as memory references in all vector instructions.
H-2	The cost of capability bounds checks can be amortised over multiple per-vector-element accesses.
Software Hypotheses — Chapter 4	
H-3	Vector code can be compiled in legacy forms (with integer addressing) and still function correctly on CHERI with no source code changes.
H-4	Vector code can be compiled into a pure-capability form from a legacy form with no source code changes.
H-5	Vector code that saves/restores variable-length vectors to/from the stack can be compiled on CHERI-RVV with no source code changes.
H-6	CHERI-vector code can run correctly in multiprocessing systems, where execution may be paused and resumed on interrupts or context switches.
Capabilities-in-Vectors — Chapter 5	
H-7	It is possible for vector registers to hold capabilities to enable copying without violating CHERI's security principles.
H-8	It is possible for vector memory accesses to load and store capabilities from vector registers without violating CHERI's security principles.
H-9	It is possible for vector instructions to manipulate capabilities in vector registers without violating CHERI's security principles.

Table 1.1: Project Hypotheses

CHAPTER 2

Background

This chapter describes RISC-V (Section 2.1), RVV (Sections 2.2 to 2.5), and CHERI (Section 2.6) to the detail required to understand the rest of the dissertation. It summarizes the relevant sections of the RISC-V unprivileged spec[5], the RISC-V "V" extension specification v1.0[4, Sections 1–9, 17], the TR-951 CHERI ISAv8 technical report[6, Chapters 5, 8], and the TR-949 technical report about C/C++ safety on CHERI[7, Section 4.4, Appendix C]. Both vectors and CHERI are described, because this dissertation caters to those who may be familiar with one but not the other.

2.1 RISC-V

RISC-V is an open family of ISAs which defines "base integer ISAs" (e.g. all 64-bit RISC-V cores implement the RV64I Base Integer Instruction Set) and extensions (e.g. the "M" extension for integer multiplication). A base instruction set combined with a set of extensions (**[TODO6 example of a architecture/feature string]**) is known as a RISC-V ISA. Because RISC-V is open, anyone can design, manufacture, and sell chips implementing any RISC-V ISA.

Each RISC-V implementation has a set of constant parameters. The most common example is `XLEN`, the length of an integer register in bits, which is tied to the base integer ISA (e.g. 64-bit ISA implies `XLEN=64`). Other constant parameters include `CLEN`, the length of a capability in bits, defined by CHERI relative to `XLEN`; and `VLEN` and `ELEN`, which are used by RVV and entirely implementation-defined.

The extensions of most relevance to this project are the "V" vector extension (RVV) and the CHERI extension. RVV recently became the officially ratified vector extension for RISC-V, which all RISC-V vector processing chips should implement going forward. The following sections summarize the vector extension, how it accesses memory, and previous implementations in academia.

2.2 A brief history of vector processing

Many vector implementations (Intel SSE/AVX, Arm's Advanced SIMD and Neon) use fixed-length vectors - e.g. 128-bit vectors which a program interprets as four 32-bit elements. As the industry's desire for parallelism grew, new implementations had to be designed with longer vectors of more elements. For example, Intel SSE/SSE2 (both 128-bit) was succeeded by AVX (128 and 256-bit), then AVX2 (entirely 256-bit), then AVX-512 (512-bit). Programs built for one extension, and hence designed for a specific vector size, could not automatically take advantage of longer vectors.

Scalable vectors address this by not specifying the vector length, and instead calculating it on the fly. Instead of hardcoding "this loop iteration uses a single vector of four 32-bit elements", the program has to ask "how many 32-bit elements will this iteration use?". This gives hardware designers more freedom, letting them select a suitable hardware vector length for their power/timing targets, while guaranteeing consistent execution of programs on arbitrarily-sized vectors. RVV uses a scalable vector model.

2.3 The RVV vector model

Summarizes [4, Sections 1-6, 17]

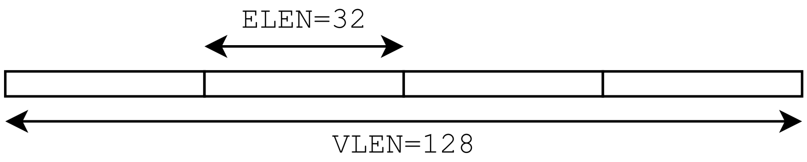
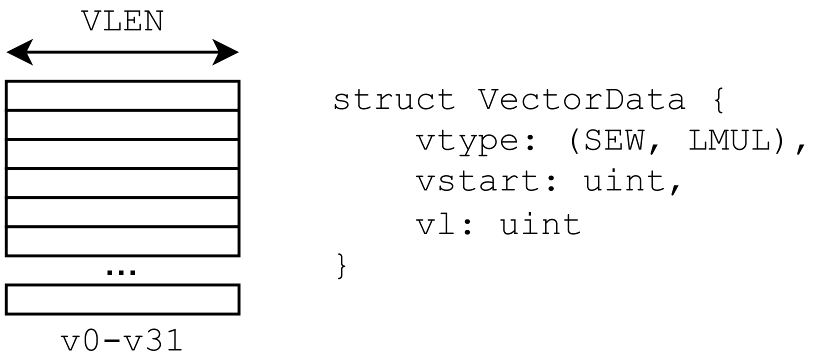


Figure 2.1: Example of a RVV vector register, with length of 128 bits and a maximum element width of 32 bits



```
struct VectorData {
    vtype: (SEW, LMUL),
    vstart: uint,
    vl: uint
}
```

Figure 2.2: Summary of additional state used by RVV

RVV defines thirty-two vector registers, each of an implementation-defined constant width

VLEN. These registers can be interpreted as *vectors* of *elements.* The program can configure the size of elements, and the implementation defines a maximum width ELEN. Fig. 2.1 shows a simple example of a 128-bit vector, where the maximum element length is 32-bits.

RVV also adds some state that defines how the vector registers are used (see Fig. 2.2). These are stored in RISC-V Control and Status Registers (CSRs), which the program can read. vtype (Section 2.3.1) defines how the vector registers are split into elements. vstart and vl (Section 2.3.2) divides the elements into three disjoint subsets: *prestart*, the *body*, and the *tail.* Masked accesses (Section 2.3.3) further divide the *body* into *active* and *inactive* elements. This section also describes the vector exception model (Section 2.3.4).

2.3.1 vtype

The vtype CSR contains two key fields that describe how vector instructions interpret the contents of vector registers. The first is the Selected Element Width (SEW), which is self-explanatory. It can be 8, 16, 32, or 64. 128-bit elements are referenced a few times throughout but haven't been formally specified (see [4, p32]).

The second field is the Vector Register Group Multiplier (LMUL). Vector instructions don't just operate over a single register, but over a register *group* as defined by this field. For example, if LMUL=8 then each instruction would operate over 8 register's worth of elements. These groups must use aligned register indices, so if LMUL=4 all vector register operands should be multiples of 4 e.g. v0, v4, v8 etc. In some implementations this may increase throughput, which by itself is beneficial for applications.

However, the true utility of LMUL lies in widening/narrowing operations (see Fig. 2.3). For example, an 8-by-8-bit multiplication can produce 16-bit results. Because the element size doubles, the number of vector registers required to hold the same number of elements also doubles. Doubling LMUL after such an operation allows subsequent instructions to handle all the results at once. At the start of such an operation, fractional LMUL (1/2, 1/4, or 1/8) can be used to avoid subsequent results using too many registers.

vtype also encodes two flags: mask-agnostic and tail-agnostic. If these are set, the imple-

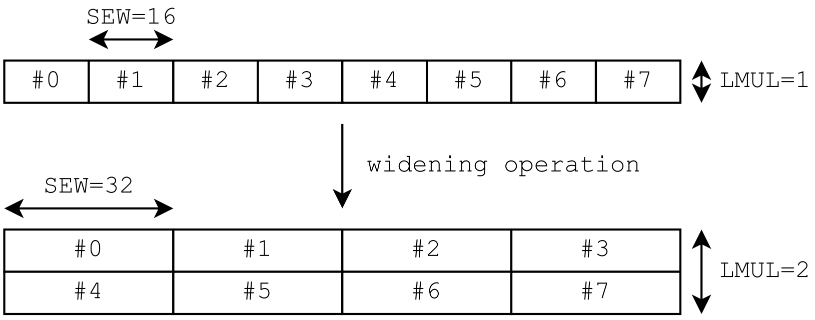
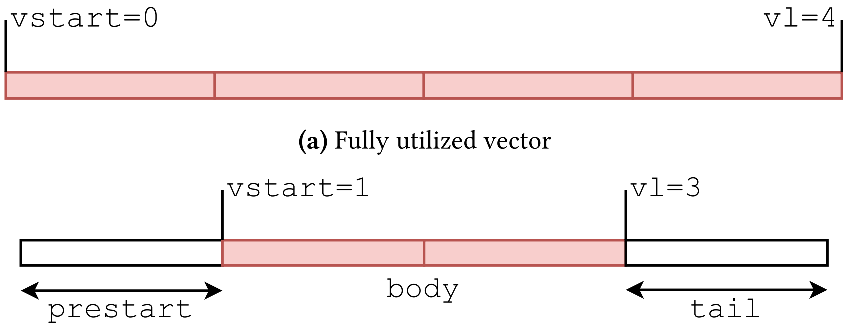


Figure 2.3: Example of using LMUL to access results of widening operations



(a) Fully utilized vector

(b) Partially utilized vector

Figure 2.4: Examples of vector utilization with `vl` and `vstart`

mentation is *allowed* to overwrite any masked-out or tail elements with all 1s.

Most vector instructions will interpret their operands using `vtype`, but this is not always the case. Some instructions (such as memory accesses) use different Effective Element Widths (`EEW`) and Effective LMULs (`EMUL`) for their operands. In the case of memory accesses, the `EEW` is encoded in the instruction bits and the `EMUL` is calculated to keep the number of elements consistent. Another example is widening/narrowing operations, which by definition have to interpret the destination registers differently from the sources.

Programs update `vtype` through the `vsetvl` family of instructions. These are designed for a "stripmining" paradigm, where each iteration of a loop processes some elements until all elements are processed. `vsetvl` instructions take a requested `vtype` and the number of remaining elements to process (the Application Vector Length or `AVL`), and return the number of elements that will be processed in this iteration. This value is saved in a register for the program to use, and also saved in the internal `vl` CSR.

2.3.2 `vl` and `vstart` — Prestart, body, tail

The first CSR is the Vector Length `vl`, which holds the number of elements that could be updated from a vector instruction. The program updates this value through fault-only-first loads (Section 2.4.3) and more commonly `vsetvl` instructions.

In the simple case, `vl` is equal to the total available elements (see Fig. 2.4a). It can also be fewer (see Fig. 2.4b), in which case vector instructions will not write to elements in the "tail" (i.e. elements past `vl`). This eliminates the need for a 'cleanup loop' common in fixed-length vector programs.

In a similar vein, `vstart` specifies "the index of the first element to be executed by a vector instruction". Elements before `vstart` are known as the *pre-start* and are not touched by executed instructions. It is usually only set by the hardware whenever it is interrupted mid-instruction (see Fig. 2.5 and Section 2.3.4) so that the instruction can be re-executed later without corrupting completed values. Whenever a vector instruction completes, `vstart` is

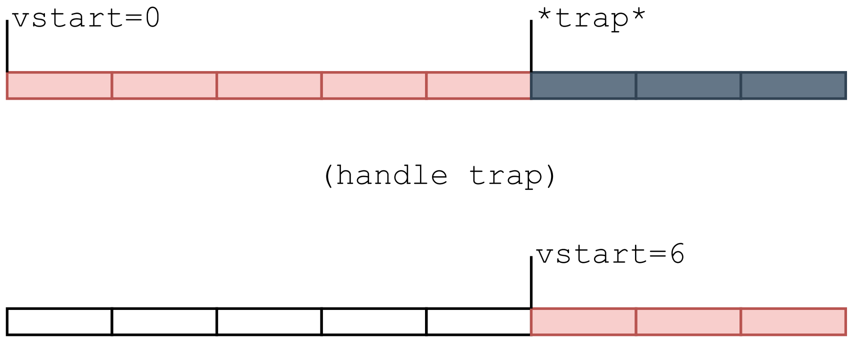


Figure 2.5: Example of the hardware setting `vstart` after a trap

reset to zero.

The program *can* set `vstart` manually, but it may not always work. If an implementation couldn't arrive at the value itself, then it is allowed to reject it. The specification gives an example where a vector implementation never takes interrupts during an arithmetic instruction, so it would never set `vstart` during an arithmetic instruction, so it could raise an exception if `vstart` was nonzero for an arithmetic instruction.

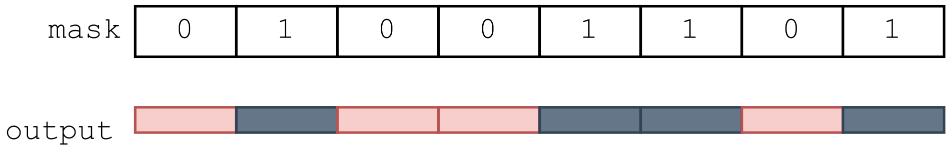


Figure 2.6: Example of masking a vector operation

2.3.3 Masking — Active/inactive elements

Most vector instructions allow for per-element *masking* (see Fig. 2.6). When masking is enabled, register `v0` acts as the 'mask register', where each bit corresponds to an element in the vector[1]. If the mask bit is 0, that element is *active* and will be used as normal. If the mask bit is 1, that element will be *inactive* and not written to (or depending on the mask-agnostic setting, overwritten with 1s). When masking is disabled, all elements are *active*.

[1] A single vector register will always have enough bits for all elements. The maximum element count is found when SEW is minimized (8 bits) and LMUL is maximized (8 registers), and is equal to `VLEN * LMUL / SEW = VLEN * 8 / 8 = VLEN`.

2.3.4 Exception handling

Summarizes [4, Section 17]

During the execution of a vector instruction, two events can prevent an instruction from fully completing: a synchronous exception in the instruction itself, or an asynchronous interrupt from another part of the system. Implementations may choose to wait until an instruction fully completes before handling asynchronous interrupts, making it unnecessary to pause the instruction halfway through, but synchronous exceptions cannot be avoided in this way (particularly for those performing memory accesses).

The RVV specification defines two modes for 'trapping' these events, which implementations may choose between depending on the context (e.g. the offending instruction), and notes two further modes which may be used in further extensions. All modes start by saving the PC of the trapping instruction to a CSR `*epc`.

2.3.4.1 Imprecise vector traps

Imprecise traps are intended for events that are not recoverable, where "reporting an error and terminating execution is the appropriate response". They do not impose any extra requirements on the implementation. For example, an implementation that executes instructions out-of-order does not need to guarantee that instructions older than `*epc` have completed, and is allowed to have completed instructions newer than `*epc`.

If the trap was triggered by a synchronous exception, the `vstart` CSR must be updated with the element that caused it. The specification calls out synchronous exceptions in particular, but does not mention asynchronous interrupts. It's likely that imprecise traps for asynchronous interrupts should also set `vstart`, but this issue has been raised with the authors for further clarification[2]. The specification also states "There is no support for imprecise traps in the current standard extensions", meaning that the other standard RISC-V exceptions do not use and have not considered imprecise traps.

2.3.4.2 Precise vector traps

Precise vector traps are intended for instructions that can be resumed after handling the interrupting event. This means the architectural state (i.e. register values) when starting the trap could be saved and reloaded before continuing execution. Therefore it must look like instructions were completed in-order, even if the implementation is out-of-order:

- Instructions older than `*epc` must have completed (committed all results to the architectural state)
- Instructions newer than `*epc` must **not** have altered architectural state.

[2] https://github.com/riscv/riscv-v-spec/issues/799

On a precise trap, regardless of what caused it, the `vstart` CSR must be set to the element index on which the trap was taken. The save-and-reload expectation then add two constraints on the trapping instruction's execution:

- Operations affecting elements preceding `vstart` must have committed their results
- Operations affecting elements at or following `vstart` must either
  - not have committed results or otherwise affected architectural state
  - be *idempotent* i.e. produce exactly the same result when repeated.

The idempotency option gives implementations a lot of leeway. Some instructions **[TODO7 examples]** are specifically prohibited from overwriting their inputs to make them idempotent. If an instruction is idempotent, an implementation is even allowed to repeat operations on elements *preceding* `vstart`. However for memory accesses the idempotency depends on the memory being accessed. For example, reading or writing a memory-mapped I/O region may not be idempotent.

Another memory-specific issue is that of *demand-paging*, where the OS needs to step in and move virtual memory pages into physical memory for an instruction to use. This use-case is specifically called out by the specification for precise traps. Usually, this is triggered by some element of a vector memory access raising a synchronous exception, invoking a precise trap, and writing the "Machine Trap Value" scalar register with the offending address[8, Section 3.1.21]. `vstart` must be set to an element at (or before[3]) the one that demanded the page, because that element must perform the access after reloading. If an implementation sets `vstart` to the offending element, because operations preceding `vstart` must have completed, any elements that could potentially trigger demand-paging *must* wait for the preceding elements to complete.

2.3.4.3 Other modes

The RVV spec mentions two other potential trap modes. First is "Selectable precise/imprecise traps", where an implementation provides a bit that selects between precise or imprecise traps. The intent is to allow precise traps to be selected for e.g. debugging purposes, and for imprecise traps to be selected for extra performance.

The second mode is "Swappable traps", where a trap handler could use special instructions to "save and restore the vector unit microarchitectural state". The intent seems to be to support context switching with imprecise traps, which could also require the *opaque* state (i.e. internal

[3] If the memory region is idempotent, then `vstart` could any value where all preceding elements had completed. It could even be zero, in which case all accesses would be retried on resume, as long as it could guarantee forward progress.

state not visible to the program) to be saved and restored. Right now, it seems that context switching always requires a precise trap.

Neither of these modes are actually defined, but they are simply noted as possibilities for the future.

2.3.5 Summary

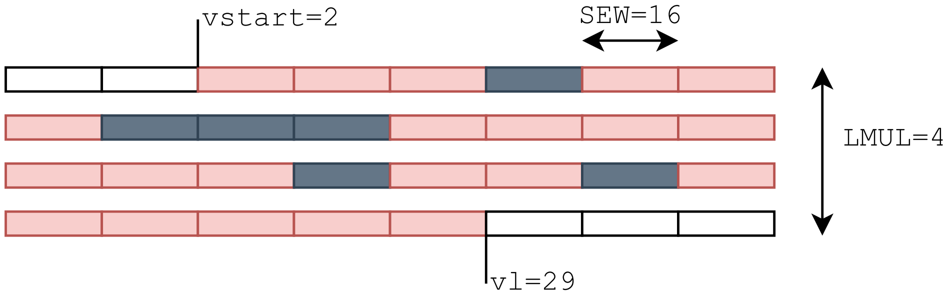


Figure 2.7: Combined examples for RVV vector

Fig. 2.7 shows all of the above features used in a single configuration:

- The instruction was previously interrupted with a precise trap and restarted, so `vstart=2`
- Elements are 16-bit
- `LMUL=4` to try and increase throughput
- Only 29 of the 32 available elements were requested, so `vl=29` (3 tail elements)
- Some elements are masked out/inactive (in this case seemingly at random)
- Overall, 21 elements are active

2.4 RVV memory instructions

Summarizes [4, Sections 7-9]

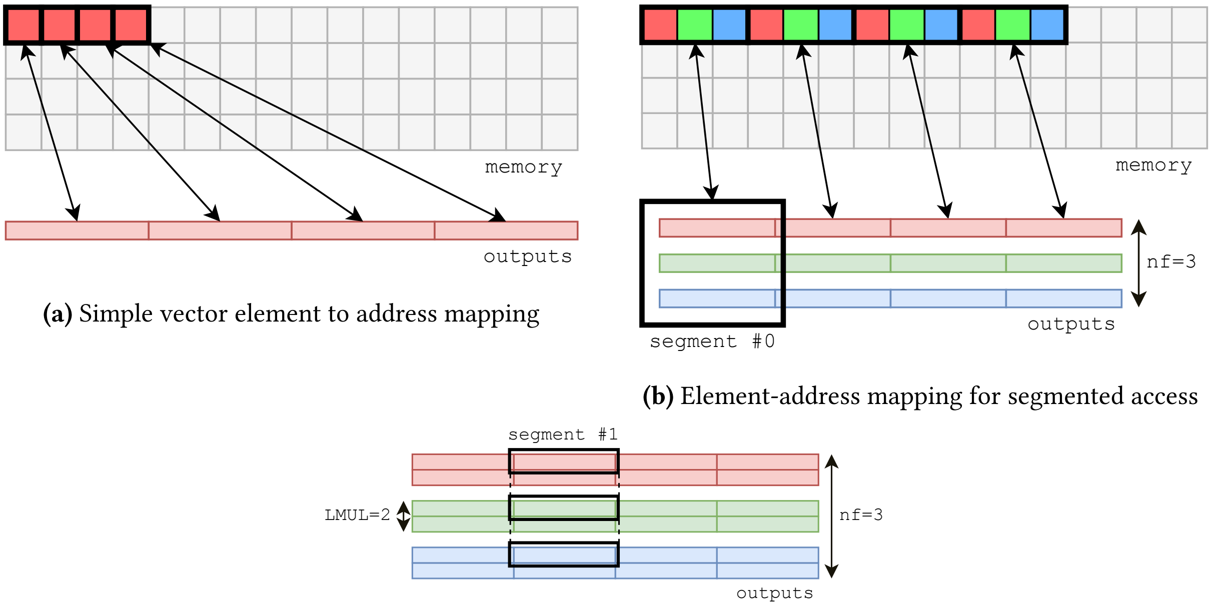
RVV defines three broad categories of memory access instructions, which can be further split into five archetypes with different semantics. This section summarizes each archetype, their semantics, their assembly mnemonics, and demonstrates how they map memory accesses to vector elements.

For the most part, memory access instructions handle their operands as described in Section 2.3. `EEW` and `EMUL` are usually derived from the instruction encoding, rather than reading the `vtype` CSR. In a few cases the Effective Vector Length `EVL` is different from the `vl` CSR, so for simplicity all instructions are described in terms of `EVL`.

2.4.1 Segmented accesses

Three of the five archetypes (unit/strided, fault-only-first, and indexed) support *segmented* access. This is used for unpacking contiguous structures of $1 \leq$ `nf` ≤ 8 *fields* and placing each field in a separate vector. In these instructions, the values of `vl`, `vstart`, and the mask register are interpreted in terms of segments.

Fig. 2.8 demonstrates a common example: the extraction of separate R, G, and B components from a color. Without segmentation, i.e. $n = 1$, each consecutive memory address maps to a consecutive element in a single vector register group. With segmentation, elements are grouped into segments of $n > 1$ fields, where each field is mapped to a different vector register group. This principle extends to `LMUL > 1` (Fig. 2.8c).



(a) Simple vector element to address mapping

(b) Element-address mapping for segmented access

(c) Example of segment mapping for `LMUL > 1`

Figure 2.8: Comparison between segmented and unsegmented accesses

2.4.2 Unit and Strided accesses

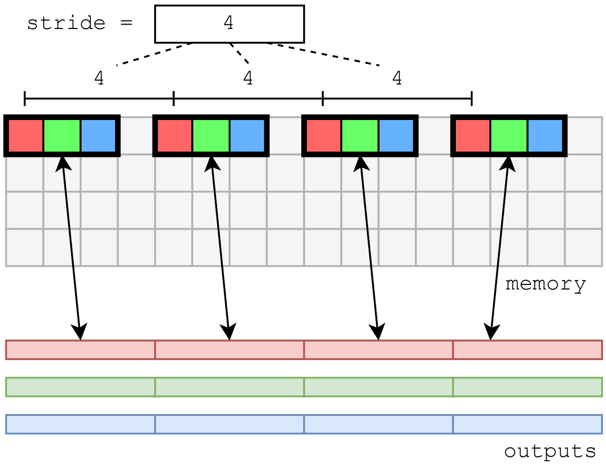
(Unit) `vlseg<nf>e<eew>.v vd, (rs1), vm`

(Strided) `vlsseg<nf>e<eew>.v vd, (rs1), rs2, vm`

(a) Instruction

Masked?	`vm == 0`
`stride`	(Unit) `<eew> * <nf>` (Strided) From `rs2`
EEW	`<eew>`
EVL	`vl`
EMUL	`VLEN * <eew> / EVL`
NFIELDS	`<nf>`

(b) How fields are interpreted



(c) Example of a segmented strided access
`EEW=8-bits`, `nf=3`, `stride=4`,`EVL=4`

Figure 2.9: Segmented Unit/Strided Access Information

Moves active elements of `nf` vector register groups to/from contiguous segments of memory, where each segment is separated by `stride` bytes.

- The start of each segment is separated by `stride` bytes.
  - The Unit version (short for Unit-stride) tightly packs segments, equivalent to selecting `stride = nf * eew / 8`.
  - `stride` may be negative or zero.
  - If `rs2` is register `x0`, implementations may perform fewer than EVL memory accesses. Otherwise, they must appear to perform all memory accesses, even if the value of `rs2` is zero.
- This instruction doesn't do anything if the `vstart >= EVL`.

Ordering

There are no ordering guarantees, other than those required by precise vector traps (if used).

Exception Handling

If any element within segment i triggers a synchronous exception, `vstart` is set to i and a precise or imprecise trap is triggered. Load instructions may overwrite active segments past the segment index at which the trap is reported, but not past EVL.[4, Section 7.7] Upon entering a trap, it is implementation-defined how much of the faulting segment's accesses are performed.

2.4.3 Unit fault-only-first loads

`vlseg<nf>e<eew>ff.v vd, (rs1), vm`	
Masked?	`vm == 0`
EEW	`<eew>`
EVL	`vl`
EMUL	`VLEN * <eew> / EVL`
NFIELDS	`<nf>`

Table 2.1: Unit Fault-only-First Information

This is equivalent to a unit load in all respects but exception handling. If any access in segment 0 raises an exception[4], `vl` is not modified and the trap is taken as usual. If any access in any active segment > 0 raises an exception, the trap is not taken, `vl` is reduced to the index of the offending segment, and the instruction finishes. If an asynchronous interrupt is encountered at any point, the trap is taken and `vstart` is set as usual.

Similar to plain loads, if an exception is encountered the instruction is allowed to update segments past the offender (but not past the original `vl`). If any synchronous exception or asynchronous interrupt occurs, regardless of the segment index, it is implementation-defined how much of the faulting segment's accesses are performed.

[4] Segment 0 may be masked out, in which case this is impossible.

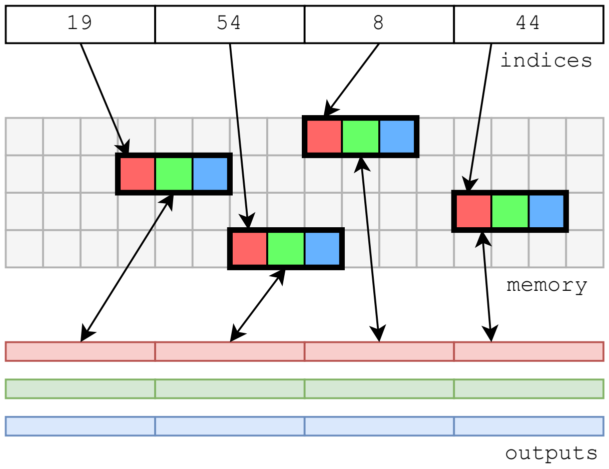
2.4.4 Indexed accesses

```
vl<u|o>xseg<nf>e<eew>.v vd, (rs1), vs2, vm
```

(a) Instruction

Masked?	`vm == 0`
Element EEW	`vtype.SEW`
Element EMUL	`vtype.LMUL`
Ordered?	`<u|o>`
Index Vector	`vs2`
Index EEW	`<eew>`
Index EMUL	`VLEN * <eew> / EVL`
`NFIELDS`	`<nf>`
`EVL`	`vl`

(b) How fields are interpreted



(c) Example of a segmented indexed access `EEW=8-bits`, `nf=3`

Figure 2.10: Segmented Indexed Access Information

Moves elements of `nf` vector register groups to/from contiguous segments of memory, where each segment is offset by an index (in bytes) taken from another vector.

- The start of each segment is defined by `base address + index_vector[i]`.
- This instruction doesn't do anything if `vstart >= EVL`.

Ordering

Accesses within each segment are not ordered relative to each other. If the ordered variant of this instruction is used, then the segments must be accessed in order (i.e. 19, 54, 8, 44 for Fig. 2.10c). Otherwise, segment ordering is not guaranteed.

Exception Handling

If any element within segment i triggers a synchronous exception, `vstart` is set to i and a precise or imprecise trap is triggered. Load instructions may overwrite active segments past the segment index at which the trap is reported, but not past `EVL`[4, Section 7.7]. Upon entering a trap, it is implementation-defined how much of the faulting segment's accesses are performed.

2.4.5 Unit whole-register accesses

`vl<nreg>re<eew>.v vd, (rs1)`

Masked?	False
Number of Registers	`<nreg>`
EEW	`<eew>`
EVL	`NFIELDS * VLEN / EEW`
EMUL	1

Table 2.2: Unit Whole Register Information

Moves the contents of `nreg` vector registers to/from a contiguous range in memory. Equivalent to a unit-stride access where EVL equals the total number of elements in `nreg` registers.

- `nreg` must be a power of two.
- Doesn't support segmented access.
- This instruction doesn't do anything if `vstart >= EVL`.

Ordering and exception handling are identical to unit-stride accesses (Section 2.4.2).

2.4.6 Unit bytemask accesses

`vlm.v vd, (rs1)`

Masked?	False
EEW	8-bits
EVL	`ceil(vl/8)`
EMUL	1

Table 2.3: Unit Bytemask Information

Moves the contents of a mask register to/from a contiguous range of memory. This instruction transfers at least `vl` bits, one bit for each element that could be used in subsequent vector instructions. This will always fit in a single vector register (see Section 2.3.3), hence `EMUL = 1` in all cases.

- This instruction always operates as if the tail-agnostic setting of `vtype` is true.
- This instruction doesn't support segmented access.
- This instruction doesn't do anything if `vstart >= EVL`.

Ordering and exception handling are identical to unit-stride accesses (Section 2.4.2).

2.5 Previous RVV implementations

Even before v1.0 of the RVV specification was released, multiple implementations were released in academia and industry. These implementations showcase the diversity allowed by a scalable model: Johns and Kazmierski integrated a minimal vector processor into a scalar pipeline meant for microcontrollers (`VLEN=32`) [9], Di Mascio et al. employed a RVV implementation for deep learning in space[10], and AndesCode, SiFive, and Alibaba have released cores with `VLEN`s up to 512[11][12][13]. Other academic examples include Ara[14], Arrow[15], RISC-V^2[16], and Vicuna[17], which all decouple the vector processing from the scalar pipeline[5].

This is only going to continue: multiple implementations were just recently presented at RISC-V Week in Paris (May 2022). Vitruvius[18] uses extremely long vectors `VLEN=256*64=16384`, is implemented as a decoupled processor, and is the first RISC-V processor to support the Open Vector Interface (OVI)[6]. VecProM[19] splits its approach into two, where vectors beyond a certain length are strip-mined and processed in hardware using a scratch memory, using OVI to connect multiple heterogeneous vector processors to a scalar core. Both were produced from the Barcelona Supercomputing Center under the European Processor Initiative.

[5] These implementations are not examined further as they do not go into detail on their load/store implementations.

[6] `semidynamics/OpenVectorInterface` on Github

2.6 CHERI

In CHERI, addresses/pointers are replaced with capabilities: unforgeable tokens that provide *specific kinds of access* to an *address* within a *range of memory*. The above statement is enough to understand what capabilities contain[7]:

- Permission bits, to restrict access
- The *cursor*, i.e. the address it currently points to
- The *bounds*, i.e. the range of addresses this capability could point to

A great deal of work has gone into compressing capabilities down into a reasonable size (see [20], **[TODO8 add diagram from TR-941?]**), and using the magic of floating-point all of this data has been reduced to just 2x the architectural register size. For example, on 64-bit RISC-V a standard capability is 128-bits long. The rest of this dissertation assumes capabilities are 128-bits long for simplicity.

A CHERI implementation has to enforce three security properties about its capabilities[6, Section 1.2.1]:

- Provenance — Capabilities must always be derived from valid manipulations of other capabilities.
- Integrity — Corrupted capabilities cannot be dereferenced.
- Monotonicity — Capabilities cannot increase their rights.

Integrity is enforced by tagging registers and memory. Every 128-bit register and aligned 128-bit region of memory has an associated tag bit, which denotes if its data encodes a valid capability[8]. If any non-capability data is written to any part of the region the tag bit is zeroed out. Instructions that perform memory accesses can only do so if the provided capability has a valid tag bit. As above, significant work has gone into the implementation to reduce the DRAM overhead of this method (see [21] for an example).

Provenance and Monotonicity are enforced by all instructions that manipulate capabilities. If an implementation detects a violation of either property, it will zero out the tag bit and rely on Integrity enforcement to ensure it is not dereferenced. Some CHERI-enabled architectures, such as CHERI-RISC-V, also raise a synchronous exception when this occurs.

[7] This is a slight simplification. For the purposes of vector memory accesses the *otype* of a capability can be ignored, as any type other than `UNSEALED` cannot be dereferenced anyway.

[8] This has the side-effect that capabilities must be 128-bit aligned in memory.

2.6.1 CHERI-RISC-V ISA

The Cambridge Computer Lab's TR-951 report[6] describes the latest version of the CHERI architecture (CHERI ISAv8) and proposes applications to MIPS, x86-64, and RISC-V. CHERI-RISC-V is a mostly straightforward set of additions to basic RISC-V ISAs. It adds thirty-two general-purpose capability registers, thirty-two Special Capability Registers (SCRs), and many new instructions.

The new general-purpose capability registers are each of size `CLEN = 2 * XLEN` plus a tag bit. These registers store compressed capabilities. While there is always a logical distinction between the pre-existing *integer* registers `x0-x31` and the *capability* registers `cx0-cx31`, the architecture may store them in a Split or Merged register file. A Split register file stores the integer registers separately from capability registers, so programs can manipulate them independently. A Merged register file stores thirty-two registers of length `CLEN`, using the full width for the capability registers, and aliases the integer registers to the bottom `XLEN` bits. Under a merged register file, writing to an integer register makes the capability counterpart invalid, so programs have to be more careful with register usage.

[TODO9 diagram for split/merged register file?]

Many of the new SCRs are intended to support the privileged ISA extensions for e.g. hypervisors or operating systems. The emulator doesn't use these, so their SCRs are not listed here, but there are two highly relevant SCRs for all modes: the Program Counter Capability and the Default Data Capability.

The PCC replaces the program counter and adds more metadata, ensuring instruction fetches have the same security properties as normal loads and stores. The DDC is used to sandbox integer addressing modes. CHERI-RISC-V includes new instructions which use integer addressing, and allows legacy (i.e. integer addressed) code to function on CHERI systems without recompiling for CHERI-RISC-V. These instructions all use integer addresses relative to the DDC, and the DDC controls the permissions those instructions have.

2.6.2 Instruction changes

TR-951[6, Chapter 8] specifies a suite of new instructions, as well as a set of modifications to pre-existing instructions. Many of the new instructions are unrelated to pre-existing instructions, and implement capability-specific operations like accessing fields of capability registers. The most relevant new instructions for our case are the various loads/stores.

CHERI-RISC-V adds new instructions for loading integer and capability data, either via capabilities or using integer addressing through the DDC (Table 2.4). These instructions are slightly more limited than the pre-existing counterparts, because they do not support immediate offsets. **[TODO10 is there rationale for that documented somewhere?]** To maintain compatibility with legacy programs that haven't been compiled for CHERI-RISC-V, the behaviour of basic RISC-V load/store opcodes changes to either use capabilities as memory references or use integer addressing via the DDC, depending on the encoding mode.

Name	Direction	Data type	Address calculation
L[BHWD][U].CAP	Load	Integer	via capability register
L[BHWD][U].DDC	Load	Integer	via DDC
LC.CAP	Load	Capability	via capability register
LC.DDC	Load	Capability	via DDC
S[BHWD].CAP	Store	Integer	via capability register
S[BHWD].DDC	Store	Integer	via DDC
SC.CAP	Store	Capability	via capability register
SC.DDC	Store	Capability	via DDC

Table 2.4: New CHERI load/store instructions

Name	Direction	Data type	Address calculation (Capability/Integer mode)
[C]LC[1]	Load	Capability	via capability/DDC
[C]SC[2]	Store	Capability	via capability/DDC
L[BWHD][U]	Load	Integer	via capability/DDC
S[BWHD]	Store	Integer	via capability/DDC
FL[WDQ]	Load	Float	via capability/DDC
FS[WDQ]	Store	Float	via capability/DDC
LR	Load	Integer	via capability/DDC
SC	Store	Integer	via capability/DDC
AMO[3]	-	Integer	via capability/DDC

[1] Replaces RV128 LQ [2] Replaces RV128 SQ [3] All atomic memory operations

Table 2.5: Preexisting RISC-V load/store instructions modified by CHERI-RISC-V

2.6.3 Capability and Integer encoding mode

CHERI-RISC-V specifies two encoding modes, selected using a flag in the PCC `flags` field. *Capability mode* modifies the behaviour of pre-existing instructions to take address operands as capabilities. This makes the basic load/store instruction behaviour exactly equivalent to newly introduced counterparts: e.g. `L[BWHD][U] == L[BWHD][U].CAP`. The DDC may still be used in this mode via the new instructions e.g. `S[BWHD].DDC`.

Integer mode seeks to emulate a standard CHERI-less RISC-V architecture as much as possible. All pre-existing RISC-V memory access instructions take address operands as integers, which are dereferenced relative to the DDC[9]. This makes the basic load/store instruc-

[9] Of course, the DDC must be valid when it is used in this mode, and all bounds checks etc. must still pass.

tion behaviour exactly equivalent to newly introduced counterparts: e.g. `L[BWHD][U] == L[BWHD][U].DDC`. The new instructions may still be used to dereference and inspect capability registers, but all other instructions access registers in an integer context i.e. ignoring the upper bits and tag from merged register files.

2.6.4 Pure-capability and Hybrid compilation modes

CHERI's de facto compiler, CHERI-Clang**[TC4[10]]**, supports two ways to compile CHERI-RISC-V which map to the different encoding modes.

Pure-capability mode treats all pointers as capabilities, and emits pre-existing RISC-V instructions that expect to be run in capability mode[11].

Hybrid mode treats pointers as integer addresses, dereferenced relative to the DDC, unless they are annotated with `__capability`. This mode allows programs to be gradually ported to CHERI, **[TODO11 which is important in ways I am too tired to express]** This mode emits pre-existing RISC-V instructions that take integer operands, and uses capabilities through the new instructions. All capabilities in hybrid mode are created manually by the program by copying and shrinking the DDC.

2.6.5 Capability relocations

Summarizes [7, Section 4.4, Appendix C]

Binary applications compiled in pure-capability mode require some "global" capabilities to exist at startup, e.g. the capability which points to the `main()` function. It would be a security risk to synthesize these capabilities from thin air, or to allow the binary file itself to contain tag bits.

Instead, CHERI ELF binaries contain a set of requested "relocations" (the `__cap_relocs` section) which instruct the runtime environment to create capabilities with specific permissions and bounds in specific places. This process uses the normal CHERI capability instructions, so any invalid requests will cause a program crash, maintaining security. Further complexity is introduced with dynamic linking, and in the future these relocations may change format, both described in [7], but the above description is sufficient to understand the rest of this paper.

[10]

[11] This wasn't derived from documentation, but instead from manual inspection of emitted code.

CHAPTER 3

Hardware emulation investigation

In order to experiment with integrating CHERI and RVV, we implemented a RISC-V emulator in the Rust programming language named `riscv-v-lite`. The emulator can partially emulate four unprivileged[1] RISC-V ISAs (Table 3.1), and was also used as the base for capabilities-in-vectors research (Chapter 5). This chapter explores the development of the emulator, the implementation of CHERI support (including supplementary libraries), the addition of vector support, and the conclusions drawn about the integration of CHERI and RVV (referred to as CHERI-RVV throughout).

	Architecture	Extensions
32-bit	`rv32imv`	Multiply, CSR, Vector[4]
64-bit	`rv64imv`	Multiply, CSR, Vector[4]
64-bit	`rv64imvxcheri`	Multiply, CSR, Vector[4], CHERI
64-bit	`rv64imvxcheri-int`	Multiply, CSR, Vector[4], CHERI (Integer)

[4] Floating-point parts of the vector extension are not supported.

Table 3.1: `riscv-v-lite` supported architectures

3.1 Developing the emulator

The emulator for each architecture follows a similar pattern. A `Processor` struct stores the register file and the available RAM. A separate `ProcessorModules` struct holds all ISA modules the processor can execute (e.g. the base RV64 Integer ISA, the Multiply extension, and the Vector extension).

The gap between the `Processor` and the ISA modules is bridged by a module-specific “connector” struct, which holds references to data in the `Processor` that is required by the

[1] i.e. entirely bare-metal without privilege levels for OSs or hypervisors.

ISA module. For example, the RV64 Integer ISA's connector contains the current PC, a virtual reference to a register file, and a virtual reference to memory. This allows different `Processor` structs (e.g. a normal RV64 and a CHERI-enabled RV64) to reuse the same ISA modules despite using different register file implementations.

Each `Processor` implements a single stage pipeline. Instructions are fetched, decoded with a common decoder function[2], and executed. The processor asks each ISA module in turn if it wants to handle the instruction, and uses the first module to say yes. If the ISA module returns a new PC value it is immediately applied, otherwise it is automatically incremented. This structure easily represents basic RISC-V architectures, and can scale up to support many different new modules.

3.1.1 Emulating CHERI

Manipulating CHERI capabilities securely and correctly is a must for any CHERI-enabled emulator. Capability encoding logic is not trivial by any means, so the `cheri-compressed-cap` C library was re-used rather than implementing it from scratch. Rust has generally decent interoperability with C, but some of the particulars of this library caused issues.

3.1.1.1 `rust-cheri-compressed-cap`

`cheri-compressed-cap` provides two versions of the library by default, for 64-bit and 128-bit capabilities, which are generated from a common source through extensive use of the preprocessor. Each variant defines a set of preprocessor macros (e.g. the widths of various fields) before including two common header files `cheri_compressed_cap_macros.h` and `cheri_compressed_cap_common.h`. The latter then defines every relevant structure or function based on those preprocessor macros. For example, a function `compute_base_top` is generated twice, once as `cc64_decompress_mem` returning `cc64_cap_t` and another time as `cc128_decompress_mem` returning `cc128_cap_t`. Elegantly capturing both sets was the main challenge for the Rust wrapper.

[TODO12 table of relevant structures/functions?]

One of Rust's core language elements is the Trait - a set of functions and "associated types" that can be *implemented* for any type. This gives a simple way to define a consistent interface for two different data types: define a trait `CompressedCapability` with all of the functions from `cheri_compressed_cap_common.h`, and implement it for both. In the future, this would allow the Morello versions of capabilities to be added easily. A struct `CcxCap<T>` is also defined which uses specific types for addresses and lengths pulled from a `CompressedCapability`. For example, the 64-bit capability structure holds a 32-bit address value, and the 128-bit capability a 64-bit address.

[2] The decoder, and therefore all emulated processors, doesn't support RISC-V Compressed instructions.

128-bit capabilities can cover a 64-bit address range, and thus can have a length of 2^{64}. Storing this length requires 65-bits, so all math in `cheri_compressed_cap_common.h` uses 128-bit length values. C doesn't have any standardized 128-bit types, but GCC and LLVM provide so-called "extension types" which are used instead. However, the x86-64 ABI doesn't define any rules for how 128-bit values must be stored or passed as arguments**[TC5[3]]**, which causes great pain to anyone who needs to pass them across a language boundary i.e. us[4]. While this can be resolved through careful examination[6], we instead rely on the Rust and Clang compilers using compatible LLVM versions and having identical 128-bit semantics.

[TODO13 C code wasn't documented, Rust is] [TODO14 move markdown documentation into rust :)]

The CHERI-RISC-V documentation contains formal specifications of all the new CHERI instructions, expressed in the Sail architecture definition language[8]. These definitions are used in the CHERI-RISC-V formal model ([9]), and require a few helper functions (see [6, Chapter 8.2]). To make it easier to port the formal definitions directly into the emulator the `rust-cheri-compressed-cap` library also provides those helper functions through a wrapper trait.

[TODO15 documentation is available on a github.io, not crates.io yet because I don't have access to CSTRD-CHERI and they'd likely want to host it]

3.1.1.2 Integrating into the emulator

Integrating capabilities into the emulator was relatively simple thanks to the modular emulator structure. A capability-addressed memory type was created, which wraps a simple integer-addressed memory in logic which performs the relevant capability checks. For integer encoding mode, a further integer-addressed memory type was created which wraps the capability addressed mode, where all integer addresses are bundled with the DDC before passing through to the capability-addressed memory. Similarly, a merged capability register file type was created that exposed integer-mode and capability-mode accesses. This layered approach meant code for basic RV64I operations did not need to be modified to handle CHERI at all - simply passing the integer-mode memory and register file would perform all relevant checks.

[TODO16 diagram]

Integrating capability instructions was also simple. Two new ISA modules were created: `XCheri64` for the new CHERI instructions, and `Rv64imCapabilityMode` to override the behaviour of legacy instructions in capability-encoding-mode. **[TODO17 show the program flow for using modules]** The actual Processor structure was left mostly unchanged. Integer ad-

[3] **x86-64 ABI rules**

[4] Rust explicitly warns against passing 128-bit values across FFI, and the Clang users manual even states that passing `i128` by value is incompatible with the Microsoft x64 calling convention**[TC6[5]]**.

[6] For example, on LLVM 128-bit values are passed to functions in two 64-bit registers**[TC7[7]]**. This could be replicated in Rust by passing two 64-bit arguments rather than one 128-bit one.

[8] `rems-project/sail` on Github

[9] `CTSRD-CHERI/sail-cheri-riscv` on Github

dresses were changed to capabilities throughout, memory and register file types were changed as described above, and the PCC/DDC were added.

The final hurdle was the capability relocations outlined in Section 2.6.5. Because we're emulating a bare-metal platform, there is no operating system to do this step for us. A bare-metal C function has been written to perform the relocations[10], which could be compiled into the emulated program. However, I wasn't sure how to find the addresses of the generated relocations in C, so I performed the relocations in Rust by examining the compiled ELF file before starting emulation. In future research it should be doable to perform the relocations entirely in bare-metal C.

3.1.2 Emulating vectors

Vector instructions are executed by a Vector ISA module, which stores all registers and other state. `VLEN` is hardcoded as 128-bits, and the maximum `ELEN` is 128-bits[11]. To support both CHERI and non-CHERI execution pointers are separated into an address and a *provenance* - **[TODO18 short definition of provenance]**. The vector unit retrieves an address + provenance pair from the base register, generates a stream of addresses to access, then rejoins each address with the provenance to access memory. When using capabilities, provenance is defined in terms of the base register e.g. "the provenance is provided by capability register X". On non-CHERI platforms, or when emulating a CHERI processor in integer mode[12], the vector unit doesn't check provenance.

Arithmetic and configuration instructions are generally simple to implement, so aren't covered here. The emulator splits vector memory accesses into three phases: decoding, checking, and execution. A separate decoding stage may technically not be necessary in hardware (especially the parts checking for errors and reserved instruction encodings, which a hardware platform could simply assume won't happen), but it allows each memory access instruction to be classified into one of the five archetypes outlined in Section 2.4. It is then easy to define the checking and execution phases separately for each archetype, as the hardware would need to do.

3.1.2.1 Decoding phase

Decoding is split into two steps: finding the encoded **`nf`** and **`eew`** values, then interpreting them based on the encoded archetype. Vector memory access instructions are encoded similarly to the F extension's floating-point load/store instructions, which include an "element width". The vector extension adds four extra "element width" values which imply the access is vectorized.

[10] `src/crt_init_globals.c` in `CTSRD-CHERI/device-model` on GitHub

[11] This is not technically supported by the specification, but is used by capabilities-in-vectors (Chapter 5).

[12] See Section 3.1.2.4 for the reasoning behind this decision.

If any of these values are found, the instruction is interpreted as a vector access and `nf` is extracted.

Once the generic parameters are extracted, the `mop` is checked to determine the indexing method (Unit, Strided, Indexed-Ordered, or Indexed-Unordered). If a unit access is selected, the second argument field encodes an extra value to choose between different unit-stride archetypes (normal unit access, fault-only-first, whole register, or bytemask). Strided and indexed accesses just use their dedicated archetypes. Once the archetype is found, supplemental calculations can be performed (e.g. computing `EVL = ceil(vl/8)` for bytemask accesses), and the relevant information is returned as a `DecodedMemOp` enumeration.

[TODO19 diagram of floating point ld/st vs. vector ld/st] [TODO20 Decision tree for operation decoding]

3.1.2.2 Fast-path checking phase

The initial motivation for this project was investigating the impact of capability checks on performance. One approach that we immediately hit upon was the concept of a "fast-path", where certain instructions could check their whole access range against a capability immediately rather than check each individual element. Section 3.2 describes methods for calculating the "tight bounds" for each access type, i.e. the minimum range of bytes that must be accessible, and ways that architectural complexity can be traded off to calculate *wider* bounds.

The emulator calculates tight bounds for all accesses. If this bounds doesn't pass the capability check, the emulator raises an imprecise trap and stops immediately. In the case of fault-only-first loads, where synchronous exceptions (e.g. capability checks) are explicitly handled, the access continues regardless and elements are checked individually. This is also the expected behaviour if a capability check for *wider* bounds fails. The emulator deviates from the spec in that `vstart` is *not* set when the tight bounds check fails, as it does not know exactly which element would have triggered the exception. As noted in Section 3.2, a fully compliant machine must check each access to find `vstart` in these cases.

3.1.2.3 Execution phase

If the fast-path check deems it appropriate, the emulator continues execution of the instruction in two phases. First, the mapping of vector elements to accessed memory addresses is found. The code for this step is independent of the access direction, and an effective description of how each type of access works. It and can be found in **[TODO21 appendix XYZ]**. The previously computed tight bounds are sanity-checked against these accesses, and the accesses are actually performed.

3.1.2.4 Integer vs. Capability encoding mode

As noted in Section 2.6.3 CHERI-RISC-V defines two execution modes that the program can switch between. In Integer mode "address operands to existing RISC-V load and store opcodes contain integer addresses" which are implicitly dereferenced relative to the default data capability, and in Capability mode those opcodes are modified to use capability operands. Integer mode was included in the interests of maintaining compatibility with legacy code that hasn't been adapted to capabilities. As similar vector code may also exist, CHERI-RVV treats vector memory access instructions as "existing RISC-V load and store opcodes" and requires that they respect integer/capability mode.

3.2 Fast-path calculations

Because CHERI, and indeed the vector extension, target all levels of computer architecture from embedded systems to cloud servers, it's important for fast-paths to be scalable and adjust to the implementation complexity. To that end, we propose a method of generating the address range for accesses of each archetype, noting where architectural complexity can be traded off for tighter coverage.

[TODO22 fast-path could be split up? i.e. for LMUL = 8, could execute a fast-path for each register in the group rather than all 8 at once]

3.2.1 Possible fast-path outcomes

In some cases, a failed address range check may not mean the access fails. The obvious case is fault-only-first loads, where capability exceptions may be handled without triggering a trap. Implementations may also choose to calculate wider bounds for the sake of simplicity, or even forego a fast-path check altogether. Thus, a fast-path check can have three outcomes depending on the circumstances:

- Success - All accesses will succeed
- Likely-Failure - At least one access *may* raise an exception
- Failure - At least one access *must* raise an exception
- Unchecked

[TODO23 if an address range is calculated then: if capability contains it: SUCCESS else if it was wide or FoF: Likely-Failure else: FAILURE] [TODO24 if an address range isn't calculated: Unchecked] [TODO25 Put the above into an algorithm format]

A Success means no per-access capability checks are required. Likely-Failure and Unchecked results mean each access must be checked, to see if any of them actually raise an exception.

Unfortunately, accesses still need to be checked under Failure, because both precise and imprecise traps need to report the offending element in `vstart`[13].

Because all archetypes may have Failure or Likely-Failure outcomes, hardware must provide a fallback slow-path for each archetype which checks/performs each access in turn. In theory, a CHERI-RVV specification could relax the `vstart` requirement for imprecise traps, and state that all capability exceptions trigger imprecise traps. In this case, only archetypes that produce Likely-Failure outcomes need the slow-path. However, it is likely that for complexity reasons all masked accesses will use wide ranges, thus producing Likely-Failure outcomes and requiring slow-paths for all archetypes anyway. Because the Likely-Failure and Failure cases require the slow-path anyway, computing the fast-path can only be worthwhile if Success is the common case.

3.2.2 Masked accesses

For all masked accesses, masked-out/inactive segments should not trigger capability exceptions. Therefore, a tight bounds must include only the smallest and largest active segments. These segments can be found by inspecting the mask vector: either checking each bit in turn or using parallel logic to find the lowest/highest set bits. Care must be taken with these checks to ensure elements outside the range $[vstart, evl)$ are not counted.

$$\texttt{vstart}_{active} = \min(i\ \forall\ \texttt{vstart} \leq i < \texttt{evl}\ \text{where}\ mask[i] = 1) \tag{3.2.1}$$

$$\texttt{evl}_{active} = \max(i\ \forall\ \texttt{vstart} \leq i < \texttt{evl}\ \text{where}\ mask[i] = 1) + 1 \tag{3.2.2}$$

Tradeoffs

If using parallel logic to find the lowest/highest bits, it could be difficult to account for $[vstart, evl)$. An implementation could choose to only calculate tight bounds when the mask is fully utilized, i.e. $vstart = 0$, $evl = VLEN$, and assume wider bounds otherwise.

Accounting for masked accesses at all may not be worth the extra complexity. Only elements masked off on the edges make any difference, and it may be uncommon for long runs of edge elements to be masked off. Thus, an implementation could choose to ignore masking entirely when computing the ranges. This does mean that all failures become Likely-Failure when masking is enabled, because all elements outside the capability bounds may be masked off.

[TODO26 iterating over elements may still be more energy-efficient than doing individual capability checks?]

[13] In very particular cases, e.g. unmasked unit-strided accesses where `nf = 1`, the capability bounds could be used to calculate what the offending element must have been. We believe this is too niche of a use case to investigate further, particularly given the complexity of the resulting hardware.

3.2.3 Unit accesses

For unit segmented accesses, which includes fault-only first, the tight address range for an access is simple to calculate. Whole register and bytemask accesses can simplify this by fixing `nf = 1` and `eew = 8`.

$$base + [\texttt{vstart}_{active} * \texttt{nf} * \texttt{eew}, \texttt{evl}_{active} * \texttt{nf} * \texttt{eew}) \tag{3.2.3}$$

[TODO27 Note that the equation is a simplification of strided for `stride = nf * eew`]

Tradeoffs

`nf` is not guaranteed to be a power of two (except for the whole-register case), so calculating the 'tight' address range would require a multiplication by an arbitrary four-bit value between 1 and 8. If this multiplication is too expensive, implementations could choose to classify all `nf` > 1 cases as Unchecked.

Unless extra restrictions are placed on `vstart`, calculating the start of this range requires another arbitrary multiplication. To avoid this one could assume `vstart = 0` and treat failures as Likely-Failure for other cases. Once could also classify all nonzero `vstart` accesses as Unchecked.

Even if the previous two optimizations are applied, the final range still requires a multiplication `evl` $*$ `eew`. Thankfully, because `eew` may only be one of four powers-of-two, this can be encoded as a simple shift.

3.2.4 Strided accesses

Strided accesses bring further complication, especially as the stride may be negative.

$$bounds(\texttt{stride}) = base + \begin{cases} [\texttt{vstart}_{active} * \texttt{stride}, (\texttt{evl}_{active} - 1) * \texttt{stride} + \texttt{nf} * \texttt{eew}) & \texttt{stride} \geq 0 \\ [(\texttt{evl}_{active} - 1) * \texttt{stride}, \texttt{vstart}_{active} * \texttt{stride} + \texttt{nf} * \texttt{eew}) & \texttt{stride} < 0 \end{cases} \tag{3.2.4}$$

This is formed of three components:

- $\texttt{vstart}_{active} * \texttt{stride}$, the start of the first segment. This can be simplified to 0, just like for unit accesses, to avoid an arbitrary multiplication.
- $(\texttt{evl}_{active} - 1) * \texttt{stride}$, the start of the final segment. This requires an arbitrary multiplication, unless strided accesses are all Unchecked.
- $\texttt{nf} * \texttt{eew}$, the length of a segment, which can be implemented with a shift.

3.2.5 Indexed accesses

This is the most complicated access of the bunch, because the addresses cannot be computed without reading the index register.

$$[\mathit{base} + \min(\texttt{offsets}[\texttt{vstart}_{\mathit{active}}..\texttt{evl}_{\mathit{active}}]), \tag{3.2.5}$$

$$\mathit{base} + \max(\texttt{offsets}[\texttt{vstart}_{\mathit{active}}..\texttt{evl}_{\mathit{active}}]) + \texttt{nf} * \texttt{eew}) \tag{3.2.6}$$

The most expensive components here are of course min, max of the offsets. These could be calculated in hardware through parallel reductions, making it slightly more efficient than looping over each element. A low-hanging optimization could be to remove the $\texttt{vstart}_{\mathit{active}}..\texttt{evl}_{\mathit{active}}$ range condition, performing the reduction over the whole register group, which would make failures Likely-Failure where $\texttt{vstart}_{\mathit{active}} != 0 \,||\, \texttt{evl}_{\mathit{active}} != \texttt{VLMAX}$. This calculation could also be restricted to certain register configurations to reduce the amount of required hardware. Indeed, the amount of hardware could be reduced to zero by simply classifying all indexed accesses as Unchecked.

3.3 Going beyond the emulator

The emulator is a single example of a conformant CHERI-RVV implementation, and does not exercise every part of the specification. Four properties stand out:

- The emulator assumes all element accesses are naturally aligned, but the spec allows misaligned accesses.
- The emulator doesn't consider multiple hardware threads, essentially assuming all accesses are atomic.
- Segments/elements are always accessed in order, despite the spec not enforcing ordering
- Imprecise traps are used for all exceptions - precise trap behaviour is not explored.

This section notes how relaxed access ordering and precise exceptions may affect the hardware in ways not previously explored.

3.3.1 Misaligned accesses

Implementations are allowed to handle vector accesses that are not aligned to the size of the element. This support is independent of misaligned scalar access support, so if e.g. misaligned 64-bit scalar accesses are allowed, misaligned vector accesses of 64-bit elements do *not* have to be allowed.

Changing the emulator to allow misaligned accesses of integer data would not have any impact on CHERI correctness. Capability loads/stores must be aligned to `CLEN`[6, Section 3.5.2], and an implementation cannot change this. Writing misaligned integer values across a `CLEN` boundary would need to make sure to zero the tag bit on both regions, but this applies to scalar implementations as much as vector ones. Alignment only impacts CHERI-RVV to the extent that it impacts capabilities-in-vectors (**Section ??1**).

3.3.2 Atomicity of accesses/General memory model

Vector memory instructions are specified to follow the RISC-V Weak Memory Ordering model[4][14], although this model hasn't been fully explained in terms of vectors yet. RVWMO defines a global order of "memory operations": atomic operations that are either loads, stores, or both[5, Chapter 14]. The RVWMO spec assumes all memory instructions create exactly one memory operation but calls out that once the vector model is formalized, vector accesses may be defined to create multiple operations.

The RVV spec states "vector misaligned memory accesses follow the same rules for atomicity as scalar misaligned memory accesses", i.e. that misaligned accesses may be decomposed into multiple memory operations of any granularity[15]. This is the only mention of atomicity in that document.

Again, atomicity of integer data doesn't really impact the fusion of CHERI and RVV, as long as tag bits are correctly zeroed on all integer writes. However, it does impact capabilities-in-vectors (**Section ??2**).

3.3.3 Relaxed access ordering and precise traps

Ordering is only enforced insofar as it is observable. The only instructions that are forced to perform their accesses in order are indexed-ordered accesses, which can be used to write to e.g. I/O regions where order matters, and instructions that trigger precise traps. Precise traps require `vstart` to be set to a value such that all elements before `vstart` have completed their accesses, and all accesses on/after `vstart` have not completed or are idempotent.

If a vector memory access instruction is 1. not indexed-ordered and 2. guaranteed not to trigger a precise trap[16] then it may execute out of order. This does not affect CHERI-RVV in any way.

[14] Behaviour under the Total Store Ordering extension hasn't been defined.

[15] e.g. each byte could be written in a separate access.

[16] Even instructions that *would* trigger precise traps but are guaranteed not to throw an exception or respond to asynchronous interrupt may execute out of order.

3.4 Testing hypotheses

Hypothesis H-1 - Feasibility

It is possible to use CHERI capabilities as memory references in all vector instructions.

This is true. All vector memory access instructions index the scalar general-purpose register file to read the base address, and CHERI-RVV implementations can simply use this index for the scalar capability register file instead. This can be considered through the lens of adding CHERI to any RISC-V processor, and in particular adding Capability mode to adjust the behaviour of legacy instructions. RVV instructions can have their behaviour adjusted in exactly the same way as the scalar memory access instructions.

That approach then scales to other base architectures that have CHERI variants. For example, on Morello scalar Arm instructions were modified to use CHERI capabilities as memory references**[TC8[17]]**, so one may simply try to apply those modifications to e.g. Arm SVE instructions. This only works where Arm SVE accesses memory references in the same way as scalar Arm instructions did i.e. through a scalar register file. Arm SVE has other addressing modes like `u64base`, which uses a vector as a set of 64-bit addresses**[TC9[18]]**, which require more specific attention.

Hypothesis H-2 - Fast-path checks

The cost of capability bounds checks can be amortised over multiple per-vector-element accesses.

[TODO28 In a theoretical sense, definitely - n individual checks can be replaced by 1 check, amortizing whatever the cost is] [TODO29 Evaluate cost in a hardware sense: compare a theoretical implementation which does each per-element check, potentially m checks in parallel to issue m requests in parallel, all pipelined with the actual accesses, vs. hardware that has a single capability check at the start which blocks it from initiating any accesses.]

"Cost" can be defined in multiple ways. As with all concepts in hardware, implementing fast-path capability checks requires a trade-off between competing interests, which are each considered here. As mentioned above, it is also assumed that successful accesses (i.e. those without any capability violations) are the common case. Overall, it seems the key benefit of fast-path checks is power consumption.

[17] **There's definitely an Arm CHERI v8 document somewhere**

[18] **https://developer.arm.com/documentation/100891/0612/coding-considerations/using-sve-intrinsics-directly-in-your-c-code**

Power - Better

If the fast-path check succeeds, then no power needs to be wasted on capability checks for the remaining cycles of the access. If the vector unit has its own dedicated capability check logic, it could even be clock gated to completely eliminate dynamic power. This shows a clear benefit as long as the extra logic for bounds calculations uses less power than $n-1$ capability checks. Implementations which use large vectors or make careful use of the simplifications laid out in Section 3.2 should fulfil this condition easily.

$$\begin{aligned} slow-path = & \quad n\ checks \\ fast-path = & \quad 1\ check \\ & +bounds\ logic \end{aligned} \tag{3.4.1}$$

Area - Worse or negligible change

No matter how you slice it, slow-path circuitry is *always* necessary for a fully conforming implementation. If the slow-path is always required, and always takes up area, then adding any circuitry for fast-path must require more area. Elements from the slow-path, e.g. capability decoding units, may be shared with the fast-path, and any spare space in a slow-path unit could also be shared with the fast-path, so it may have a *negligible* impact. Crucially, adding a fast-path will never *decrease* the area of a conformant design.

Throughput - No change

Assuming the bounds calculations can be pipelined to the same clock speed as a capability check, putting them before a set of pipelined accesses should not affect the throughput of those accesses. If the fast-path check is subdivided between different registers in a group, the fast-path check for one register should be performed in parallel with the accesses for other registers for maximum throughput. There is currently no reason to believe the bounds calculations have to significantly decrease clock speed or throughput.

Latency - Worse

If the fast-path check could be done in parallel with other memory accesses, then it would not affect latency. Unfortunately, performing a memory access without checking if it's allowed first completely undermines the security model! Under very particular circumstances, it could be tolerable: if the access is a read with no side-effects, and the read data would be thrown away on a capability violation, and side-channel attacks were impossible (i.e. no caches were present) or ignored, then an unauthorized read out-of-bounds *technically* has no impact on security; but it's implausible that any architect on a CHERI design would accept this.

Unfortunately, the fast-path check must always block the memory access that depends on it, so the fast-path will always increase latency (unless a separate memory access is performed in parallel).

[TODO30 where to note that capability checks count as a synchronous exception? centralized point for ”here are the differences between RVV and CHERI-RVV?] **[TODO31 I wrote a type-safe wrapper for capabilities]**

CHAPTER 4

The CHERI-RVV software stack

While building hardware that can execute vector instructions is important, generating those vector instructions in software quickly and easily is equally important. This chapter explores the current state of the so-called "CHERI-RVV software stack": mainstream compiler support for vanilla RVV (Section 4.1), and the modifications required to bring support to CHERI-Clang (Section 4.2). The software hypotheses are tested with this knowledge (Section 4.3), and we recommend a set of changes to bring CHERI-Clang support to par with other compilers (Section 4.4).

4.1 Compiling vector code

Modern compilers provide many ways to generate vectorized code. While this support is very advanced for well established vector models, like x86-64 AVX, newer vector models like RVV don't have as many options. It can even be difficult to get the compiler to generate any vector instructions at all. This section examines support across the Clang and GCC compilers for various vectorization methods on RVV.

4.1.1 Required command-line options

Before you can compile vector code, the compiler must be told to use a vector ISA. This is fortunately quite easy, only requiring an addition to the architecture feature string **[TODO32 is it called that?]** in most cases. On Clang 13 and other LLVM-13-based compilers, version 0.1 of the vector specification is supported as an experimental extension, so an extra command. Clang/LLVM 14 and up support RVV v1.0. GCC is an interesting case - there is a branch of `riscv-gcc-toolchain` that supports RVV v0.1, based on RISC-V GCC 10.1, but it hasn't been touched for more than a year. See Appendix A for more information on finding and building this version.

Compiler	Required Arguments	Notes

Clang-13	`-march=rv64gv0p10`
	`-menable-experimental-extensions`
		Supports intrinsics, inline assembly for RVV v0.1

Clang-14+	`-march=rv64gv`	Supports intrinsics, inline assembly for RVV v1.0
GCC 10.1	`-march=rv64g_v`	Requires special toolchain (see Appendix A) and has incomplete support (see **Section ??3**)

Table 4.1: Command-line arguments for compiling RVV code on various compilers (assuming the base ISA is `rv64g`)

4.1.2 Automatic vectorization

[TODO33 Figure page showing generated ASM for increment loop - see godbolt links MD] Compilers with auto-vectorization can automatically create vectorized code from a scalar program. For example, a scalar loop over an array that increments each element could be converted to a vectorized loop that increments multiple elements at once. Although this is simple in some cases, auto-vectorization can take significant effort and time to implement (for example, GCC started implementing it for x86 in 2003 and only turned on basic support in 2007 **[TC10[1]]**). Currently there is no support for RVV auto-vectorization in Clang or GCC. Both compilers have support for Arm SVE auto-vectorization, explored further in Section 4.1.5.

4.1.3 Vector intrinsics

“Intrinsics” are functions built in to a compiler that can invoke low-level functionality and instructions directly for a specific architecture. When automatic vectorization is not available, intrinsics are the next best thing - they present a familiar high-level interface (function calls), that gives the programmer fine-grained control over which instructions to execute, typically providing an intrinsic for each vector instruction. The compiler then handles low-level decisions like register allocation under the hood, and sometimes may provide extra functionality for ease of use.

RVV has a comprehensive set of vector intrinsics[22], implemented in the aforementioned special version of GCC and Clang 13+. With these, the general strip-mining loop is easy to construct: **[TODO34 example based on** `https://github.com/riscv-non-isa/rvv-intrinsic-doc/blob/master/examples/rvv_memcpy.c`**]**

1. Use a `vsetvl` intrinsic to get the vector length for this iteration.
2. Allocate vector registers by declaring variables with vector types (e.g. `vuint32m8_t` represents 8 registers worth of 32-bit unsigned integers).

[1] **https://gcc.gnu.org/projects/tree-ssa/vectorization.html**

3. Pass the vector length to the computation/memory intrinsics, which operate on the vector variables.

[TODO35 Hammer home that intrinsics aren't reusable across instruction sets? e.g. AVX intrinsics don't work with RISC-V]

4.1.4 Inline assembly

If a compiler doesn't supply complete intrinsics, or if the programmer desires extremely fine-grained control, inline assembly may be used. The programmer gives a string of handwritten assembly code to the compiler, which is parsed and directly inserted into the output code at that point. The compiler still has to interpret the instruction and understand it correctly, but as long as it knows the instruction this method does not depend on any intrinsics being present (or functional[2]).

Inline assembly can interact with C code and variables through a template syntax. The programmer inserts a placeholder in the assembly code with a corresponding expression, noting how the expression is stored using a "constraint". For our purposes, constraints enforce that a value is either in a register or in memory (see Table 4.2). As an example, writing to a memory address stored in a variable could use a constraint `"m"(*addr)` - i.e. "the value pointed to by `addr` is stored in memory". **[TODO36 example] [TODO37 the previous sentence kinda sucks at getting the point across. Using a constraint forces the compiler to move the value into a register/memory]**

Using the constraint, the compiler determines how the expression's value is stored, and inserts a reference to it in the assembly string. Because this is done before the assembly string is parsed, and isn't immediately type-checked against the assembly instruction, it can lead to some difficult errors.

Clang and GCC support inline assembly for RVV quite well, and even allows the intrinsic vector types to be referenced by assembly templates (thus making the compiler do register allocation instead of the programmer). The only caveat is that *memory* constraints are not supported by RVV memory accesses. None of the vector memory access instructions support address offsets, unlike their scalar counterparts. Clang always treats the *memory* constraint as an offset access, even when that offset is zero, so it adds an offset to the assembly string **[TODO38 example]**, making it invalid. To get around this, one must use the pointer itself with a *register* constraint - effectively saying "find the register this pointer is in, and use that as the base address for the memory access" **[TODO39 example]**. On CHERI platforms, because pointers must be stored in capability registers, the *capability register* constraint must be used instead (see **[TODO40 example of CHERI-agnostic inline assembly]**).

[2] For example, CHERI-Clang could not compile code with vector intrinsics, so had to use inline assembly for all vector instructions.

”=”	Output - the old value is overwritten. Can be combined with other constraints.
”r”	Store in a register
”vr”	Store in a vector register (RVV only)
”C”	Store in a capability register (CHERI-Clang only)
”m”	Store in memory

Table 4.2: Inline assembly constraints **[TC12[3]]**
[TODO41 Beautify this table]

Broadly speaking, inline assembly supports more RVV instructions than intrinsics do. It is used extensively in the testbench code for the evaluation (Chapter 6) alongside intrinsics where possible.

4.1.5 vs. Arm SVE

Summarizes [23]

Arm SVE uses a similar model to RVV, where the vector length may scale between 128 and 2048[4] and the instructions are designed to be totally agnostic across different platforms[3]. Arm have released a C language extension to support SVE development ([24]), supported by the Arm Compiler for Embedded**[TC13[5]]**, Clang, and GCC. It supports all of the previously examined vectorization types.

Auto-vectorization is supported, and the main focus of the user guide is helping the compiler decide whether to auto-vectorize [23]. Intrinsics are also supported, and seem to cover all of the SVE instructions, but take a slightly different approach to RVV. Arm SVE intrinsics do not directly map to available instructions, but aim to “provide a regular interface and leave the compiler to pick the best mapping to SVE instructions”, while RVV intrinsics (at least for memory) tend to map 1:1 to existing instructions. Arm’s approach gives more flexibility for future extensions, as the same intrinsics could be compiled to new instructions with newer compilers.

Arm SVE also supports inline assembly, but the experience is notably worse than for RVV. The two standout issues are a lack of register allocation and the use of condition code flags for branching. Unlike RVV, the intrinsic types for vector values cannot be referenced in inline assembly[3], so all vector registers must be allocated and tracked by the programmer. Arm SVE’s equivalent of `vsetvl`, the `while` family[24], do not return the number of updated elements, and instead set the condition flags based on how many elements are updated. Because there is no way to branch based on the condition flags in C, the programmer must manually insert a label for the top of the loop, and a branch to that label (see `https://godbolt.org/z/zoWh9jq3o`), which is more error prone than the RVV method. Examples of Arm SVE code with auto-vectorization, intrinsics, and inline ASM can be found in **[TODO42 appendix based on**

[4] RVV slightly differs here, as it allows VLEN smaller than 128.

[5] `https://developer.arm.com/Tools%20and%20Software/Arm%20Compiler%20for%20Embedded`

https://godbolt.org/z/zoWh9jq3o].

Compiler	Required Arguments	Notes

CHERI-Clang
(LLVM-13)
	`-march=rv64gv0p10xcheri`
	`-menable-experimental-extensions`
	`-mabi=l64pc128`
	`-mno-relax`
		Supports intrinsics, inline assembly for RVV v0.1.
		LLVM 13 RVV support is experimental.
		ABI string required to set capability width.
		Linker relaxations must be disabled.

Table 4.3: Command-line arguments for compiling CHERI-RVV code
(assuming the base ISA is `rv64g`)

4.2 Compiling vector code with CHERI-Clang

Current CHERI compiler work is done on CHERI-Clang, a fork of Clang and other LLVM tools that supports capabilities. It's based on LLVM 13, so it includes support for vanilla RVV v0.1, but none of the vector-extension related code had been updated to work with capabilities. This section outlines the changes one has to make to CHERI-Clang and vector code to compile programs for CHERI-RVV.

4.2.1 Required command-line options

By default CHERI-Clang doesn't actually compile capability-enabled code. The documentation on enabling capabilities is unfortunately sparse and outdated. In particular, the CHERI-Clang help menu states that `--cheri` will "Enable CHERI support with the default capability size", but this has no effect (at least on RISC-V). To find up-to-date answers, we consulted the source code for the CHERIbuild build tool[6].

CHERIbuild's code[7] revealed three requirements:

- The architecture string must contain `xcheri`
- The capability length must be set using the ABI string
  - In pure-capability mode, pointers and capabilities are `CLEN` long
    - Example string: `l64pc128`
    - Integer width (`long`, or `l`) = XLEN = 64-bits
    - Pointer width (p) = Capability width (p) = CLEN = 128-bits
  - For hybrid mode, pointers remain `XLEN` long and capability length is not specified
    - Example string: `lp32`
    - Integer width (`l`) = XLEN = Pointer width (p) = 32-bits

[6] `CSTRD-CHERI/cheribuild` on Github

[7] `config/compilation_targets.py:176` in `CSTRD-CHERI/cheribuild` on GitHub

- "Linker relaxations", where function calls are converted to short jumps[25], must be disabled.
  - This is likely because CHERI requires function calls to go through capabilities
  - However the code that adds this option wasn't documented, so there may be more to it

Once the above options are set, plain CHERI-RISC-V code compiles without a hitch. Changes to CHERI-Clang itself are required to compile vectors.

4.2.2 Adapting vector assembly instructions to CHERI

LLVM uses a domain-specific language to describe the instructions it can emit for a given target. The RISC-V target describes multiple register sets that RISC-V instructions can use. Vanilla RVV vector memory accesses use the General Purpose Registers (GPR) to store the base address of each access. CHERI-Clang added a GPCR set, i.e. the General Purpose Capability Registers. As noted in Section 3.1.2.4 CHERI-RVV requires the vector memory accesses to support integer *and* capability mode, therefore two versions of the vector accesses must be created: versions which take a capability base address, only available in CHERI/Capability mode, and versions which take integer base addresses for Integer mode.

[TODO43 Appendix on how this was done?] With the above changes, inline assembly could be used to insert capability-enabled vector instructions**[TODO44 Example]**. However, as this requires using a capability register constraint for the base address, inline assembly code written for CHERI-RVV is not inherently compatible with vanilla RVV. For un-annotated pointers (e.g. `int*`), which are only capabilities in pure-capability code and integers in legacy or hybrid code, a conditional macro can be used to insert the correct constraint: **[TODO45 example]**. However, this falls apart in hybrid code for manually annotated pointers (e.g. `int* __capability`) because the macro cannot detect the annotation.

4.2.3 Adapting vector intrinsics to CHERI

Vector intrinsics are another story entirely. When compiling for pure-capability libraries, all attempts to use vector intrinsics crash CHERI-Clang **[TODO46 example of error message?]**. This is due to a similar issue to inline assembly: the intrinsics (both the Clang intrinsic functions and the underlying LLVM IR intrinsics) were designed to take regular pointers and cannot handle it when capabilities are used instead. **[TODO47 Appendix which covers what I know so far about this problem?]** Unfortunately the code for generating the intrinsics on both levels is spread across many files, and there's no simple way to change the associated pointer type (much less changing it for pure capability vs. hybrid mode).

It seems that significant compiler development work is required to bring vector intrinsics up to scratch on CHERI-Clang. We did experiment with creating replacement wrapper functions, where each function tried to mimic an intrinsic using inline asssembly. These were rejected for two reasons: the increased overhead of a function call on every vector instruction[8], and the lack of support for passing vector types as arguments or return values. The RISC-V ABI treats all vector registers as temporary and explicitly states that "vector registers are not used for passing arguments or return values"[26], and CHERI has its own issues with saving vector registers on the stack.

[TODO48 segue into saving registers on stack doesn't make sense - doesn't explain why someone would want to]

4.2.4 Storing scalable vectors on the stack

If a program uses more data than can fit in registers, or calls a function which may overwrite important register values, the compiler will save those register values to memory on the stack. Because vector registers are temporary, and thus may be overwritten by called functions, they must also be saved/restored from the stack**[TODO49 Example https://godbolt.org/z/KPTW7rcvY]**. This also applies to multiprocessing systems where a process can be paused, have the state saved, and resume later. RVV provides the whole-register memory access instructions explicitly to make this process easy.

CHERI-Clang contains an LLVM IR pass[9] which enforces strict bounds on so-called "stack capabilities" (capabilities pointing to stack-allocated data), which by definition requires knowing the size of the data ahead of time. This pass assumes all stack-allocated data has a static size, and crashes when dynamically-sized types e.g. scalable vectors are allocated. It is therefore impossible (for now) to save vectors on the stack in CHERI-Clang, although it's clear that it's theoretically possible. For example, the length of the required vector allocations could be calculated based on `VLEN` before each stack allocation is performed, or if performance is a concern stack bounds for those allocations could potentially be ignored altogether. These possibilities are investigated further in the next section.

[TODO50 Note: Arm Language C extensions https://developer.arm.com/documentation/100987/0000/ defines the concept of a sizeless type, which may be stored on the stack. Would be a good base for RVV?]

[TC14[10]]

[8] This could have been eliminated by using preprocessor macros instead of real functions, but they are difficult to program and do not easily support returning values like intrinsics do.

[9] llvm/lib/CodeGen/CheriBoundAllocas.cpp

[10] **LLVM IR pass investigated in TR-949 $3.8.2, couldn't find earlier reference to it**

4.3 Testing hypotheses

Hypothesis H-3 - Compiling legacy code for integer mode

> *Vector code can be compiled in legacy forms (with integer addressing) and still function correctly on CHERI with no source code changes.*

This is true in theory and (partially) true in practice, with two assumptions:

- The scalar elements of the code have this property
- All usages of vector memory instructions access memory that the program has access to
  - i.e. all addresses touched by vector memory instructions are within the bounds of the DDC

The CHERI-RVV versions of the vector instructions used in Chapter 3 have identical encodings to legacy RVV instructions and function correctly in Integer mode. Not only is it theoretically possible for an RVV program to function correctly with no changes on CHERI-RVV, it should be possible to take a vanilla RVV binary and run it without modification on CHERI-RVV. In practice, this depends on compiler support for the three vectorization types in Hybrid compilation mode.

Inline assembly - True

Integer mode CHERI-RVV instructions use the same general-purpose registers as memory references as in vanilla RVV. CHERI-Clang handles this case correctly.

Intrinsics - False until CHERI-Clang is fixed

Intrinsics do not currently function on CHERI-Clang in pure-capability *or* hybrid mode.

Auto-vectorization - False

CHERI-Clang, as well as vanilla Clang, does not have auto-vectorization for RVV.

Hybridized legacy programs

Hybrid-mode compliation allows pointers to be annotated with `__capability`, which means they will be represented internally with capabilities instead of integer addresses. It's possible for scalar code to use mix and match capability instructions and integer-mode memory accesses, because CHERI-RISC-V adds instructions for dereferencing capabilities that work in both Integer and Capability encoding modes. Because the behaviour of CHERI-RVV instructions only changes with the encoding mode, it is impossible to use both capability-aware and integer-addressed RVV instructions in the same program.

Hypothesis H-4 - Converting legacy code to pure-capability code

> *Vector code can be compiled into a pure-capability form from a legacy form with no source code changes.*

This is true in theory but not yet true in practice, with two assumptions:

- The scalar elements of the code have this property
- All usages of vector memory instructions access memory through references with correct provenance
  - i.e. each vector memory instruction only accesses addresses within the intended provenance of the base pointer
- All vector memory instructions use valid pointers as their base addresses
  - e.g. there are no integer-to-pointer casts that would produce invalid capabilities in pure-capability mode

All vanilla RVV vector instructions have CHERI-RVV counterparts which produce identical results in Capability mode. When recompiling for pure-capability mode, the only differences in the output binary from the original should be

- The encoding mode used
- The types of registers used for base addresses

If the abstractions used for vector programming are high-level enough to be independent of register types, a compiler may change the register types under the hood with no changes from the user. CHERI-Clang's support for these high-level abstractions is lacking.

Inline assembly - False

Inline assembly allocates registers based on a user-provided register constraint. The 'r' general-purpose register constraint used for legacy code does not match the 'C' capability register constraint, which is required for capability-mode instructions. Thus, inline vector assembly would have to be updated to compile correctly for pure-capability systems.

Intrinsics - False until CHERI-Clang is fixed

Vector intrinsics, which take pointer types as arguments, are high-level enough for the compiler to change the register types. However, intrinsics do not currently function on CHERI-Clang in pure-capability *or* hybrid mode.

Auto-vectorization - False

A compiler that auto-vectorizes scalar code, as long as that code is valid under pure-capability CHERI, could easily choose to auto-vectorize it with CHERI instructions instead. Unfortunately CHERI-Clang, as well as vanilla Clang, does not have auto-vectorization for RVV.

Hypothesis H-5 - Saving vectors on the stack

> *Vector code that saves/restores variable-length vectors to/from the stack can be compiled on CHERI-RVV with no source code changes.*

This is true in theory, but not yet supported by CHERI-Clang in practice. Placing variable-length structures on the stack is possible as long as the length can be known at runtime (and as long as the stack has space, of course). This isn't exclusive to CHERI - to push and pop values on the stack, the stack pointer must be incremented or decremented by the size of the value. Because the length already has to be measured, and CHERI-RISC-V supports setting capability bounds from runtime-computed values, it's entirely possible to correctly set tight bounds for capabilities pointing to variable-length vectors on the stack.

Hypothesis H-6 - Running CHERI-RVV code in a multiprocessing system

> *CHERI-vector code can run correctly in multiprocessing systems, where execution may be paused and resumed on interrupts or context switches.*

This requires two conditions: an OS must be able to save and restore vector state, and the vector hardware must support resuming from an interrupted state. The first condition is easy to fulfil by extending the previous hypothesis. If it is possible to save variable-length vectors on the stack, given their length is known at runtime, it must also be possible to save their data on the heap. Some OSs might need to make changes to their "current process state" structure to support variable-length data, and they would also need to allocate space for the `vtype` value, but it is certainly possible.

The second condition can be upheld in two ways. First, if the OS only context switches and services interrupts while the vector hardware is in a complete state (i.e. not partially executing an instruction), then context switches and interrupts are completely transparent to the vector hardware and no changes need to be made. Secondly, if context switches and interrupts can actually interrupt vector instructions partway through, then they can only be cleanly resumed if the vector hardware supports precise traps for the exact instruction being executed.

4.4 Recommended changes for CHERI-Clang

- Build on current work to make all RVV memory access instructions and pseudoinstructions CHERI-compatible.
- Make RVV memory access intrinsics take capabilities as arguments when compiled in pure-capability mode.
- Consider options on handling stack bounds for scalable vectors, and implement in the CheriBoundAllocas IR pass.

CHAPTER 5

Capabilities-in-vectors

5.1 Emulator changes

5.2 Hardware implications

5.3 Possible software improvements

5.4 Testing hypotheses

Hypothesis H-7 - Holding capabilities in vectors

> *It is possible for vector registers to hold capabilities to enable copying without violating CHERI's security principles.*

Hypothesis H-8 - Sending capabilities between vectors and memory

> *It is possible for vector memory accesses to load and store capabilities from vector registers without violating CHERI's security principles.*

Hypothesis H-9 - Manipulating capabilities in vectors

> *It is possible for vector instructions to manipulate capabilities in vector registers without violating CHERI's security principles.*

CHAPTER 6

Evaluation

6.1 Vectorized memcpy test and results

6.2 Capabilities-in-vector test and results

6.3 Conclusion

[TODO51 Summarize hypotheses outcomes] [TODO52 Summarize results] [TODO53 List artifacts - github repositories, where to get the rust-cheri-compressed-cap crate etc]

CHAPTER 7

References

[1] Robert N M Watson et al. *An Introduction to CHERI*. UCAM-CL-TR-941. September 2019, p. 43.

[2] Robert N.M. Watson et al. “CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization”. In: *2015 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2015, pp. 20–37. ISBN: 978-1-4673-6949-7. DOI: 10/gfpgzz.

[3] Nigel Stephens et al. “The ARM Scalable Vector Extension”. In: *IEEE Micro* 37.2 (March 2017), pp. 26–39. ISSN: 0272-1732. DOI: 10.1109/MM.2017.35.

[4] *RISC-V ”V” Vector Extension*. 20th September 2021. URL: https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf.

[5] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. 13th December 2019. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[6] Robert N. M. Watson et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. UCAM-CL-TR-951. University of Cambridge, Computer Laboratory, 2020. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.html (visited on 06/10/2021).

[7] Alexander Richardson. *Complete Spatial Safety for C and C++ Using CHERI Capabilities*. UCAM-CL-TR-949. University of Cambridge, Computer Laboratory, June 2020, p. 189. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-949.pdf.

[8] Andrew Waterman, Krste Asanovic and John Hauser, eds. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. 4th December 2021. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf.

[9] Matthew Johns and Tom J. Kazmierski. "A Minimal RISC-V Vector Processor for Embedded Systems". In: *2020 Forum for Specification and Design Languages (FDL)*. September 2020, pp. 1–4. DOI: 10/gnrfdb.

[10] Stefano Di Mascio et al. "On-Board Decision Making in Space with Deep Neural Networks and RISC-V Vector Processors". In: *Journal of Aerospace Information Systems* 18.8 (1st August 2021), pp. 553–570. DOI: 10/gnrfch.

[11] *AndesCore NX27V Processor*. Andes Technology. URL: https://www.andestech.com/en/products-solutions/andescore-processors/riscv-nx27v/ (visited on 11/12/2021).

[12] *SiFive Intelligence X280 - SiFive*. sifive.com. URL: https://www.sifive.com/cores/intelligence-x280 (visited on 15/05/2022).

[13] Chen Chen et al. "Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-Bit High Performance RISC-V Processor with Vector Extension : Industrial Product". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. May 2020, pp. 52–64. DOI: 10.1109/ISCA45697.2020.00016.

[14] Matheus Cavalcante et al. "Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-Nm FD-SOI". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.2 (February 2020), pp. 530–543. ISSN: 1557-9999. DOI: 10/gnrd7v.

[15] Imad Al Assir et al. "Arrow: A RISC-V Vector Accelerator for Machine Learning Inference". 15th July 2021. arXiv: 2107.07169 [cs]. URL: http://arxiv.org/abs/2107.07169 (visited on 11/12/2021).

[16] Kariofyllis Patsidis et al. "RISC-V2: A Scalable RISC-V Vector Processor". In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. October 2020, pp. 1–5. DOI: 10/gnfrn3.

[17] Michael Platzer and Peter Puschner. "Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation". In: *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Ed. by Björn B. Brandenburg. Vol. 196. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 1:1–1:18. ISBN: 978-3-95977-192-4. DOI: 10/gnfrn2.

[18] Francesco Minervini and Oscar Palomar Perez. “Vitruvius: An Area-Efficient RISC-V Decoupled Vector Accelerator for High Performance Computing” (RISC-V Week - Paris). 4th May 2022. URL: https://www.youtube.com/watch?v=tlC5kMhrh-k (visited on 13/05/2022).

[19] Gopinath Mahale et al. “A RISC-V VPU for Very Long and Sparse Vectors” (RISC-V Week - Paris). March 2021. URL: https://open-src-soc.org/2022-05/media/posters/4th-RISC-V-Meeting-2022-05-03-Gopinath-Mahale-poster.pdf (visited on 13/05/2022).

[20] Jonathan Woodruff et al. “CHERI Concentrate: Practical Compressed Capabilities”. In: (2019), p. 15. DOI: 10/gm9ngf.

[21] Alexandre Joannou et al. “Efficient Tagged Memory”. In: *2017 IEEE International Conference on Computer Design (ICCD)*. November 2017, pp. 641–648. DOI: 10/ghnj26.

[22] *RISC-V Vector Extension Intrinsics (v1.0)*. RISC-V Non-ISA Specifications, 16th November 2021. URL: https://github.com/riscv-non-isa/rvv-intrinsic-doc/blob/00882f19a84ab354dc8cf6a10c100b8daa2654e4/rvv-intrinsic-api.md (visited on 16/11/2021).

[23] Arm Ltd. *Arm Compiler Scalable Vector Extension User Guide Version 6.12*. 0612-00. 27th February 2019. URL: https://developer.arm.com/documentation/100891/latest/ (visited on 13/05/2022).

[24] Arm Ltd. *ARM C Language Extensions for SVE 0.0bet6*. 00bet6. 2020. URL: https://developer.arm.com/documentation/100987/0000/ (visited on 13/05/2022).

[25] Shiva Chen and Hsiangkai Wang. “Compiler Support For Linker Relaxation in RISC-V” (RISC-V Workshop Taiwan). 13th March 2019. URL: https://riscv.org/wp-content/uploads/2019/03/11.15-Shiva-Chen-Compiler-Support-For-Linker-Relaxation-in-RISC-V-2019-03-13.pdf (visited on 04/05/2022).

[26] *RISC-V ABIs Specification v1.0rc2*. 6th April 2022. URL: https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/download/v1.0-rc2/riscv-abi.pdf.

Appendix A

Building the riscv-gnu-toolchain with vector support

As of May 2022, the RISC-V GNU toolchain (hosted as riscv-collab/riscv-gnu-toolchain on Github) does not support the vector extension or it's intrinsics. **[TODO54 verify - do a fresh clone and see]** The rvv-intrinsic branch of this repository claims to support vector intrinsics, but is slightly outdated. It references a repository for glibc that no longer exists as a submodule, which makes compilation impossible.

To build the full toolchain with intrinsic support, perform the following steps (derived by the author independently, then amended based on macOS instructions from [TC15[1]]):

1. Clone the repository as usual, *without* cloning any submodules.

2. Change the .gitmodules file to point the riscv-glibc submodule at the upstream glibc repository https://sourceware.org/git/?p=glibc.git, which hosts the RISC-V version as of 19 March 2021[2].

   [TODO55 Example]

   - Alternatively, this can be pointed at https://github.com/riscvarchive/riscv-glibc.git, the archived version of the old repository[TC16[3]].

3. **[TODO56 clone submodules (macOS guy turns off SSL for this?)]**

4. **[TODO57 Configure with as many risc-v exts as possible]**

5. Build

[TODO58 Does RISC-V GCC 10.2 support vector intrinsics if you compile with v?]

[1] https://github.com/riscv-collab/riscv-gcc/issues/323

[2] riscvarchive/riscv-glibc (branch archive-notify) on Github

[3] maxOS GCC instructions