WAMR Porting to CHERI Morello

Project Report

Version 2.0

Table of Contents

[**1**](#_heading=h.gjdgxs) **Introduction 9**

[1.1](#_heading=h.30j0zll) The Web Assembly Micro Runtime 9

[*1.1.1*](#_heading=h.1fob9te) *WASI 10*

[1.2](#_heading=h.3znysh7) CHERI Morello 10

[*1.2.1*](#_heading=h.2et92p0) *Morello Hardware 10*

[1.3](#_heading=h.tyjcwt) WAMR Porting Objectives 10

[**2**](#_heading=h.3dy6vkm) **Executive Summary 12**

[2.1](#_heading=h.1t3h5sf) WAMR Codebase 12

[2.2](#_heading=h.4d34og8) Development Environment 12

[2.3](#_heading=h.2s8eyo1) CHERI Porting Issues 13

[2.4](#_heading=h.17dp8vu) WAMR AOT Compiler Issues 14

[2.5](#_heading=h.3rdcrjn) Conclusions 14

[**3**](#_heading=h.26in1rg) **Development, Test & Debug Environment 16**

[3.1](#_heading=h.lnxbz9) Setting up the Morello SDP 16

[*3.1.1*](#_heading=h.35nkun2) *Configuring the Host Machine 16*

[*3.1.2*](#_heading=h.44sinio) *Building the SW Stack 16*

[*3.1.3*](#_heading=h.3j2qqm3) *Setting up the Morello Board 18*

[*3.1.4*](#_heading=h.qsh70q) *Installing Debian Pure-Cap on the Morello SSD 20*

[*3.1.5*](#_heading=h.3as4poj) *SSH access to the Morello Board 21*

[3.2](#_heading=h.147n2zr) Fork of the WAMR Repository 22

[3.3](#_heading=h.3o7alnk) Cross-Compilation for Morello 22

[*3.3.1*](#_heading=h.23ckvvd) *GNU Toolchain Setup 23*

[*3.3.2*](#_heading=h.ihv636) *LLVM Toolchain 24*

[3.4](#_heading=h.32hioqz) Remote Debugging 25

[*3.4.1*](#_heading=h.1hmsyys) *Using Gdb 25*

[*3.4.2*](#_heading=h.41mghml) *Using lldb 26*

[3.5](#_heading=h.2grqrue) Visual Studio IDE 27

[*3.5.1*](#_heading=h.vx1227) *Visual Studio Installation and Setup 28*

[*3.5.2*](#_heading=h.4f1mdlm) *Verifoxx CHERI WAMR Updated Project Layout 28*

[*3.5.3*](#_heading=h.28h4qwu) *Remote Debugging on Morello 33*

[*3.5.4*](#_heading=h.46r0co2) *Troubleshooting Visual Studio Build & Debug Issues 37*

[3.6](#_heading=h.4k668n3) Testing Environment 38

[*3.6.1*](#_heading=h.2zbgiuw) *Generating WASM from C Code 38*

[*3.6.2*](#_heading=h.1egqt2p) *Generating WASM from WAT 39*

[**4**](#_heading=h.3ygebqi) **Generic Porting Steps 40**

[4.1](#_heading=h.2dlolyb) Building the new WAMR Platform 40

[*4.1.1*](#_heading=h.sqyw64) *CMakeLists.txt 40*

[*4.1.2*](#_heading=h.3cqmetx) *Platform API Folder 41*

[*4.1.3*](#_heading=h.1rvwp1q) *Program Entry Point 41*

[*4.1.4*](#_heading=h.4bvk7pj) *Build Warnings 42*

[4.2](#_heading=h.2r0uhxc) WAMR Memory Management 42

[*4.2.1*](#_heading=h.1664s55) *Understanding WAMR Memory Structures 42*

[*4.2.2*](#_heading=h.25b2l0r) *Structure Alignments 44*

[*4.2.3*](#_heading=h.kgcv8k) *Pointer Alignments 45*

[*4.2.4*](#_heading=h.34g0dwd) *Operand Stack Changes 48*

[*4.2.5*](#_heading=h.1jlao46) *Stack Size 52*

[*4.2.6*](#_heading=h.43ky6rz) *Frame Stack Allocation 52*

[*4.2.7*](#_heading=h.2iq8gzs) *Memory Mapping 52*

[*4.2.8*](#_heading=h.xvir7l) *Native Heap Changes 53*

[*4.2.9*](#_heading=h.3hv69ve) *WASM Linear Memory and Heap Allocation 57*

[4.3](#_heading=h.1x0gk37) WASM to Native Interface with Morello Assembler 60

[*4.3.1*](#_heading=h.4h042r0) *Native Pointer Invocation 61*

[*4.3.2*](#_heading=h.2w5ecyt) *Passing large number of Arguments 61*

[*4.3.3*](#_heading=h.1baon6m) *Need for CHERI Changes 61*

[*4.3.4*](#_heading=h.3vac5uf) *CHERI Changes List 62*

[4.4](#_heading=h.39kk8xu) Hybrid Support 67

[4.5](#_heading=h.1opuj5n) WAMR Documentation Updates 68

[**5**](#_heading=h.48pi1tg) **WASM Interpreter Modes 69**

[5.1](#_heading=h.2nusc19) Classic Interpreter 69

[*5.1.1*](#_heading=h.1302m92) *Frame Stack Changes 69*

[*5.1.2*](#_heading=h.2250f4o) *WASM calling Native Function Changes 72*

[*5.1.3*](#_heading=h.haapch) *Developing Native Code called from WASM 73*

[5.2](#_heading=h.319y80a) WAMR Fast Interpreter 74

[*5.2.1*](#_heading=h.1gf8i83) *Operation of Fast Interpreter 75*

[*5.2.2*](#_heading=h.40ew0vw) *CHERI Modifications 80*

[*5.2.3*](#_heading=h.3ep43zb) *Code Size Issue 86*

[5.3](#_heading=h.1tuee74) Interpreter mode Testing & Validation 86

[**6**](#_heading=h.4du1wux) **WASI Support 92**

[6.1](#_heading=h.2szc72q) WAMR and WASI Components 92

[*6.1.1*](#_heading=h.184mhaj) *WASI Components 93*

[*6.1.2*](#_heading=h.279ka65) *Support within WAMR 93*

[*6.1.3*](#_heading=h.meukdy) *WASI Capabilities 94*

[*6.1.4*](#_heading=h.36ei31r) *Impact of CHERI 95*

[6.2](#_heading=h.1ljsd9k) OpenSSL Support in WASM 95

[*6.2.1*](#_heading=h.45jfvxd) *WASI-WAMR OpenSSL Version 95*

[*6.2.2*](#_heading=h.2koq656) *Using the openSSL WASM Libraries 96*

[*6.2.3*](#_heading=h.zu0gcz) *Example TLS Client and Server 97*

[*6.2.4*](#_heading=h.3jtnz0s) *CHERI Morello SSL / TLS Issues 100*

[6.3](#_heading=h.1yyy98l) WASI Output Examples 100

[6.4](#_heading=h.4iylrwe) WASI Test Suite 101

[*6.4.1*](#_heading=h.2y3w247) *WASI Test Suite Operation 102*

[*6.4.2*](#_heading=h.3x8tuzt) *Modifications to support CHERI Morello 106*

[**7**](#_heading=h.2ce457m) **WAMR AOT Porting 109**

[7.1](#_heading=h.rjefff) AOT Mode Operation 109

[*7.1.1*](#_heading=h.3bj1y38) *AOT Compiler 110*

[*7.1.2*](#_heading=h.14ykbeg) *AOT File Format 120*

[*7.1.3*](#_heading=h.3oy7u29) *WAMR AOT Support 122*

[7.2](#_heading=h.j8sehv) AOT Compiler Changes for CHERI 134

[*7.2.1*](#_heading=h.1idq7dh) *Building for Cross-Compilation 135*

[*7.2.2*](#_heading=h.42ddq1a) *Creating Morello Targets 137*

[*7.2.3*](#_heading=h.2hio093) *Morello Address Space Support 138*

[*7.2.4*](#_heading=h.3gnlt4p) *Enforced Reduced Optimisation Pass 141*

[7.3](#_heading=h.1vsw3ci) WAMR AOT Runtime Changes for CHERI 142

[*7.3.1*](#_heading=h.4fsjm0b) *Target Info support 142*

[*7.3.2*](#_heading=h.2uxtw84) *Morello Pure-cap Procedure Linkage Table 142*

[*7.3.3*](#_heading=h.1a346fx) *Morello Symbol Relocations 144*

[*7.3.4*](#_heading=h.3u2rp3q) *Setting up the Code Region 146*

[*7.3.5*](#_heading=h.2981zbj) *Aligning Module Structure Buffers 146*

[7.4](#_heading=h.odc9jc) AOT System Wide Changes for Morello 146

[*7.4.1*](#_heading=h.38czs75) *WAMR Pure-cap PC Relative Addressing to Data Regions 146*

[*7.4.2*](#_heading=h.1nia2ey) *Structure Layout Equivalence of WAMR and wamrc on Morello 159*

[*7.4.3*](#_heading=h.47hxl2r) *AOT File Binary Version 166*

[7.5](#_heading=h.2mn7vak) WAMR AOT Bug Workaround 166

[7.6](#_heading=h.11si5id) AOT Mode Testing & Validation 169

[*7.6.1*](#_heading=h.3ls5o66) *AOT Mode Automated Testing 170*

[*7.6.2*](#_heading=h.20xfydz) *Example Outputs 171*

[**8**](#_heading=h.4kx3h1s) **WAMR External References 176**

[8.1](#_heading=h.302dr9l) Overview 176

[*8.1.1*](#_heading=h.1f7o1he) *Null ExternRefs 177*

[*8.1.2*](#_heading=h.3z7bk57) *Command Line Handling 178*

[*8.1.3*](#_heading=h.2eclud0) *Additional WASM Operations 178*

[*8.1.4*](#_heading=h.thw4kt) *WAMR Internal Operation Details 178*

[8.2](#_heading=h.1smtxgf) Modifications for CHERI 179

[*8.2.1*](#_heading=h.4cmhg48) *Technological Solutions 180*

[*8.2.2*](#_heading=h.2rrrqc1) *Passing Externref from Native Code to WASM 181*

[*8.2.3*](#_heading=h.16x20ju) *Passing Externref from WASM to Native Function 182*

[*8.2.4*](#_heading=h.3qwpj7n) *Command Line Handling Changes 183*

[8.3](#_heading=h.261ztfg) ExternRef Testing & Validation 184

[*8.3.1*](#_heading=h.l7a3n9) *Extended Native Test Library 185*

[*8.3.2*](#_heading=h.356xmb2) *Command Line Externref Testing 187*

[*8.3.3*](#_heading=h.1kc7wiv) *Externref Standalone Native Test Application 190*

[**9**](#_heading=h.44bvf6o) **Testing 200**

[9.1](#_heading=h.2jh5peh) WAMR Front End Application 200

[*9.1.1*](#_heading=h.ymfzma) *Code Structure 201*

[9.2](#_heading=h.3im3ia3) Automated Test System with WASI Test Suite 201

[*9.2.1*](#_heading=h.1xrdshw) *Test Adapters 201*

[*9.2.2*](#_heading=h.4hr1b5p) *Interpreter Test Adapter 206*

[*9.2.3*](#_heading=h.2wwbldi) *AOT Test Adapter 208*

[9.3](#_heading=h.1c1lvlb) Automated Test Running 210

[*9.3.1*](#_heading=h.3w19e94) *Additional Test Cases 211*

[*9.3.2*](#_heading=h.2b6jogx) *Automated Testing Results 213*

[9.4](#_heading=h.qbtyoq) Potential Future Testing Improvements 220

[**10**](#_heading=h.3abhhcj) **Benchmarking 222**

[10.1](#_heading=h.1pgrrkc) WAMR Benchmarks 222

[10.2](#_heading=h.49gfa85) WAMR Resource Usage 222

[10.3](#_heading=h.2olpkfy) CHERI-WAMR Benchmarking Setup 223

[*10.3.1*](#_heading=h.13qzunr) *WAMR Applications 223*

[*10.3.2*](#_heading=h.3nqndbk) *Benchmarking under an Operating System 224*

[*10.3.3*](#_heading=h.22vxnjd) *WAMR Internal Profiling 224*

[*10.3.4*](#_heading=h.i17xr6) *Use of Benchmark ABI 224*

[*10.3.5*](#_heading=h.320vgez) *Toolchains 224*

[10.4](#_heading=h.1h65qms) CHERI-WAMR Benchmark Execution 225

[*10.4.1*](#_heading=h.415t9al) *Building the Executables under Test 225*

[*10.4.2*](#_heading=h.2gb3jie) *CMake and Visual Studio Support for Native Benchmark Applications 227*

[*10.4.3*](#_heading=h.vgdtq7) *Test Framework Details 229*

[10.5](#_heading=h.3fg1ce0) CHERI-WAMR Benchmark Results 233

[*10.5.1*](#_heading=h.1ulbmlt) *Hybrid Best vs Purecap Best (non-Benchmark ABI) 234*

[*10.5.2*](#_heading=h.4ekz59m) *Testing the Benchmark ABI 235*

[*10.5.3*](#_heading=h.2tq9fhf) *Toolchain Comparison: GCC vs Clang 238*

[*10.5.4*](#_heading=h.18vjpp8) *Toolchain Comparison: LLVM Like-for-Like 239*

[10.6](#_heading=h.3sv78d1) Benchmarking Conclusions 242

[*10.6.1*](#_heading=h.280hiku) *Relative Performance of WAMR 242*

[*10.6.2*](#_heading=h.n5rssn) *Explanation for Outlying Poor Performance Results 242*

[*10.6.3*](#_heading=h.375fbgg) *Development Recommendations for Optimal Purecap Performance 245*

[**11**](#_heading=h.1maplo9) **Summary and Conclusions 247**

[11.1](#_heading=h.46ad4c2) CHERI Porting Specific Issues 247

[*11.1.1*](#_heading=h.2lfnejv) *Structure Alignment 247*

[*11.1.2*](#_heading=h.10kxoro) *Capability Pointer Alignment and Size 247*

[*11.1.3*](#_heading=h.3kkl7fh) *Loss of Pointer Provenance 248*

[*11.1.4*](#_heading=h.1zpvhna) *Mmap() Issues 249*

[*11.1.5*](#_heading=h.4jpj0b3) *Out-of-Bounds Access Violation 249*

[*11.1.6*](#_heading=h.2yutaiw) *Morello ISA and ABI Extensions 249*

[*11.1.7*](#_heading=h.1e03kqp) *AOT Compiler 250*

[11.2](#_heading=h.3xzr3ei) WAMR Complexity and Legacy SW Issues 250

[*11.2.1*](#_heading=h.2d51dmb) *“Real-world” Software 250*

[*11.2.2*](#_heading=h.sabnu4) *Embedded Software 251*

[*11.2.3*](#_heading=h.3c9z6hx) *Use of Structured Language 251*

[*11.2.4*](#_heading=h.1rf9gpq) *Build-time support for Multiple Platforms 251*

[*11.2.5*](#_heading=h.4bewzdj) *Lack of Documentation 252*

[*11.2.6*](#_heading=h.2qk79lc) *Minimal Support for Development 252*

[11.3](#_heading=h.15phjt5) CHERI and Morello Issues 252

[*11.3.1*](#_heading=h.3pp52gy) *Documentation and Examples 253*

[11.4](#_heading=h.24ufcor) Key points for Future CHERI Porting 253

[**12**](#_heading=h.jzpmwk) **WAMR Compartmentalisation and Capability Manager 256**

[12.1](#_heading=h.33zd5kd) Concept 256

[*12.1.1*](#_heading=h.1j4nfs6) *WAMR Compartment Capabilities and Data Structures 258*

[*12.1.2*](#_heading=h.434ayfz) *Same or Multiple Compartments 259*

[*12.1.3*](#_heading=h.2i9l8ns) *Calling into a Compartment 259*

[*12.1.4*](#_heading=h.xevivl) *Capability Manager Callback Services 261*

[12.2](#_heading=h.3hej1je) Realisation – Phased Approach 263

[12.3](#_heading=h.1wjtbr7) Compartmentalising WAMR Code 264

[*12.3.1*](#_heading=h.4gjguf0) *Initialized Capability Problem 265*

[*12.3.2*](#_heading=h.2vor4mt) *Relocation Symbol Patching 266*

[12.4](#_heading=h.1au1eum) Calling WAMR Operation in Compartment 275

[*12.4.1*](#_heading=h.3utoxif) *Compartment Setup 275*

[*12.4.2*](#_heading=h.29yz7q8) *Compartment Call WAMR Function 276*

[*12.4.3*](#_heading=h.p49hy1) *Crossing the Compartment Boundary 279*

[*12.4.4*](#_heading=h.393x0lu) *WAMR Proxy Functions 282*

[12.5](#_heading=h.1o97atn) Compartmentalisation with iwasm Front End 286

[*12.5.1*](#_heading=h.488uthg) *New WAMR Product 286*

[*12.5.2*](#_heading=h.2ne53p9) *Modifications to iwasm Executable 287*

[*12.5.3*](#_heading=h.12jfdx2) *Changes to Common CHERI-WAMR Code 288*

[*12.5.4*](#_heading=h.3mj2wkv) *Capability Manager Service Callback Support 289*

[12.6](#_heading=h.21od6so) Compartment Callback to Capability Manager Services 289

[*12.6.1*](#_heading=h.gtnh0h) *Service Callback Flow 290*

[*12.6.2*](#_heading=h.30tazoa) *Service Callback Processing 292*

[*12.6.3*](#_heading=h.1fyl9w3) *Service Callback Return 294*

[*12.6.4*](#_heading=h.3zy8sjw) *Additional Compartment Entry Capabilities 297*

[*12.6.5*](#_heading=h.2f3j2rp) *Invoking a Service Callback Function 300*

[*12.6.6*](#_heading=h.u8tczi) *Implemented Service Callback Functions 304*

[12.7](#_heading=h.3e8gvnb) Testing and Results 304

[*12.7.1*](#_heading=h.1tdr5v4) *WASI Test Suite Updates 304*

[*12.7.2*](#_heading=h.4ddeoix) *Untestable Features 305*

[*12.7.3*](#_heading=h.2sioyqq) *Results 305*

[12.8](#_heading=h.17nz8yj) Compartmentalisation Conclusions and Future Work 306

[*12.8.1*](#_heading=h.3rnmrmc) *Future Work Proposals 306*

[*12.8.2*](#_heading=h.26sx1u5) *System Deficiencies 308*

[**13**](#_heading=h.ly7c1y) **Experimental GNU C Library Port to Support Compartments 310**

[13.1](#_heading=h.35xuupr) Overview of Loader Operation 310

[*13.1.1*](#_heading=h.1l354xk) *Modifications for Compartment Loading 311*

[13.2](#_heading=h.452snld) Implementation 312

[*13.2.1*](#_heading=h.2k82xt6) *Creating the Mirror: Obtaining and Building GLibC 313*

[*13.2.2*](#_heading=h.zdd80z) *API Modification 313*

[*13.2.3*](#_heading=h.3jd0qos) *Detecting the Loader 313*

[*13.2.4*](#_heading=h.1yib0wl) *Relocation Symbol Changes 314*

[*13.2.5*](#_heading=h.4ihyjke) *Future Work 314*

[13.3](#_heading=h.2xn8ts7) Conclusions 315

[**14**](#_heading=h.1csj400) **Issues Reference 316**

[**15**](#_heading=h.3ws6mnt) **Resources Reference 319**

# Introduction

This document describes the work involved in porting the Web Assembly Micro-runtime (WAMR) application to the CHERI Morello Linux Pure-cap platform. At the time of writing the following WAMR features have been ported and are fully functional running on Morello pure-cap and Morello hybrid-cap:

* WASM Interpreter (classic and fast modes)
* WASI LibC functionality (includes networking via Berkley sockets and a WASM port of openSSL) and use of WASI-SDK
* AOT (generation with WAMR AOT Compiler (WAMRc) and execution with WAMR)
* External References (a WASM Prototype Feature)

Within this document a description of the changes made to WAMR has been provided along with an explanation of the challenges faced making WAMR work under a CHERI architecture, and the resulting solutions.

As this work has been carried out on a research platform architecture, which is still very much in development, it is unsurprising that a number of technical issues and bugs have been found. These are also detailed in this document, along with a couple of (non-CHERI specific) issues found within WAMR itself.

A “lessons learnt” conclusions section is presented in the hope to aid in the development of both CHERI / Morello and future architectures.

Finally, an experimental part of the project has commenced which is to CHERI-compartmentalise WAMR. Progress to date is fully documented at the end of this document, along with conclusions that are specific to this experimental aspect of the workload.

## The Web Assembly Micro Runtime

Web Assembly was originally developed to allow web-browsers to execute generic code natively on the host machine. WAMR takes this runtime technology and applies it as a standalone environment, running on the host, either as an executable program or via libraries which are called from native code such as C code. Although other standalone runtimes are available (e.g *wasmtime*), WAMR is designed to be lightweight on system resources and high performance, and as such is more suited to embedded environments. Like any virtual runtime, WAMR takes many steps to ensure the contained WASM (or compiled AOT) runs in a sandboxed environment.

WAMR supports a number of architectures and operating systems including Linux, Windows and crucially for this project AArch64 embedded Linux as well as Linux x86\_64. WAMR is written largely in C, with some C++ and platform specific assembly code for low-level native code interaction. The WAMR application is designed to build on multiple platforms and to support this makes use of *CMake* as the generic build system.

For AOT support, WAMR is supplied with *wamrc* – WAMR AOT Compiler – which compiles WASM to native object code that can be executed with WAMR at a much higher speed than running WASM through in the interpreter or JIT modes. WAMRc makes use of LLVM backend libraries, accessed via the C++ API.

WAMR is open-source SW developed by the *ByteCodeAlliance,* and is available from <https://github.com/bytecodealliance/wasm-micro-runtime>.

### WASI

The Web Assembly System Interface (WASI) (<https://wasi.dev/>) provides native library support for WASM applications, and along with the *WASI-SDK* allows host applications to be compiled to WASM. WAMR supports the WASI *libC* and associated libraries making it possible to generate WASM from fully contained C applications. WASI works by providing an “upper” and “lower” part of the library, with the “upper” part being a standard C library and the lower part calling defined WASI API functions.

WAMR then provides an implementation of the WASI API as native functions, which are then called from WASM in the same way that WAMR supports calling any native function from WASM. WASI includes the ability to require permissions to access network resources, this is controlled within the WAMR WASI handlers.

## CHERI Morello

Morello is an extension of the Armv8-A architecture which implements the CHERI model. Morello supports A64 “hybrid-capability” and C64 “pure-capability” modes; the former allows applications to run with limited, or no, use of the protections that capabilities provide.

For this project we have used the Linux SW stack and have focussed on Morello pure-cap, although hybrid-cap is also supported.

### Morello Hardware

We have made use of the *Morello System Development Platform* provided by Arm Limited. This is the official DSbD prototype and comprises a Morello SoC along with associated peripherals to provide a fully functional, networked computer that can host a Debian Linux distribution.

***Note:*** *Arm also provide an FVP (Fixed Virtual Platform) for Morello, however this was not used as it was found to be extremely slow, and the SDP was already available.*

## WAMR Porting Objectives

The main objective has been to add CHERI Morello Linux pure-cap as an *additional* architecture supported by WAMR, and to keep support for all existing architectures. To this end, and given the lack of an object-oriented language to allow target specialisation, CHERI specific changes have been added as *#if / #else / #endif* preprocessor conditional macros within the codebase.

Note that in some cases the WAMR codebase has been improved for all platforms in order to better support architectures like CHERI where pointers are not just large integers, but these are minimal and in general the changes have been platform-specific code.

The intention is that at some point the changes made to support CHERI can be pulled back into the main *ByteCodeAlliance* WAMR project. To support this, periodically the private forked repository is updated from the parent WAMR repository – the last update was during May 2023.

To ensure there are no breaking changes made to support CHERI, WAMR is regularly tested on both Morello hybrid-cap and Linux x86\_64 and in fact automated test cases execute on all three platforms.

# Executive Summary

## WAMR Codebase

WAMR is a standalone runtime for WASM; its core includes a WASM Interpreter, ability to run target-specific AOT compiled WASM and also supports JIT mode (not ported). WAMR includes an implementation of the backend WASI C standard library and features the ability to call WASM from native code and vice versa.

The porting project was complicated by the fact that WAMR contains almost no documentation either within the code or externally. WAMR, written almost entirely in C code with platform specific assembler, makes heavy use of macros and *goto* instructions which makes debugging the code more difficult. The application could benefit from being written in an object-oriented language; modern C++ compilers are likely to give negligible worse performance than “plain old C” and this would allow WAMR to make use of data encapsulation, platform-specific implementation overrides and a better structure than it currently maintains.

WAMR features a number of low-level algorithms, including implementing its own heap management and wrapping system calls such as *malloc()* and *mmap()* to support multiple platforms and provide sandboxed services to WASM application. It also lazily treats pointers as 64-bit integers throughout the code and performs unsafe pointer arithmetic. All of these aspects increased the complexity of porting to the CHERI architecture.

## Development Environment

To use the Morello SDP, Arm documentation specifies an Ubuntu Linux host machine and a wired ethernet connection for the Morello board. As neither of these were available Windows 10 running an Ubuntu distribution under WSL2 was used, with a VirtualBox VM for initial disk formatting operations, and this was found to work admirably. For the Morello board, a wired ethernet to WiFi adapter was used which has worked well. The Debian pure-cap SW stack was installed on the Morello hard disk so that from boot it can start Linux, and the installation of an openSSH server enabled SSH access from the host machine for transferring files, remote debug and terminals via the local network.

To permit workable development, an IDE is needed. Visual Studio running on Windows 10 was used (partly due to its maturity and partly due to the complexities of running reasonable GUIs from WSL2) this is setup to remotely build the *CMake* WAMR project on WSL2. For the Morello builds, cross-compilation was used from Linux x86\_64 -> Linux CHERI Morello pure-cap or hybrid-cap. Initially the GNU toolchain port from Linaro was used for cross-compilation because at the start of the project the Arm LLVM toolchain port was not mature enough (too complex to use and was unable to build C++ on pure-cap). Later the LLVM toolchain was added (a *CMake toolchain file* has been added for it), but required some codebase modifications because LLVM is “stricter” in some CHERI aspects and also some minor issues were found in GCC build (which didn’t affect runtime operation).

Debugging is remotely from within Visual Studio using *gdb*. The *gdb* client runs locally on Windows and the server runs on Morello. Initially the LLVM debugger *lldb* was tried but this would not halt at breakpoints so was abandoned (a ticket was raised with Arm – unclear on the status of this).

## CHERI Porting Issues

To port WAMR a number of aspects have needed to be addressed within WAMR, the main issues are:

1. Pointer Alignment
   1. WAMR frequently allocates a byte buffer and then uses as a memory block, often casting to structures which contain pointers. Example with the WASM frame stack.
   2. This leads to a misalignment because WAMR assumes pointers and integers are the same size, but on CHERI a pointer is twice as large and must be aligned accordingly
   3. Results in *bus error*. This issue has been the largest problem.
2. Loss of Pointer Provenance
   1. WAMR freely converts between a pointer type and an integer type throughout the code.
   2. This leads to the loss of provenance and a *segmentation violation* when a capability pointer with a cleared Tag is attempted to be used
3. Size of Pointer
   1. In several places WAMR reads pointers from a byte stream or skips over the size of pointers in the stream. An example is in fast interpreter mode whereby offsets in the WASM bytestream have been converted to pointer addresses.
   2. WAMR assumes the pointer will be the same size as an integer; and further that the pointer will be aligned as an integer. Neither of these is true on CHERI.
   3. This has led to the need to ensure enough space is available for a pointer and, in the case that there may be a misalignment, enough space is available for a 2 \* sizeof(pointer) to allow the pointer to appear aligned anywhere within the space
4. Out of Bounds Access
   1. WAMR has no concept of deriving pointer capability permissions and bounds from a previous system-assigned pointer; this can lead to problems when attempting to transfer code execution to a new frame of reference as the new frame may be out of bounds
   2. Solving this problem involves allocating a single block, with a capability pointer to access the whole block, and using this for each separate buffer that is required
5. *Mmap()* Protection
   1. As is documented in the *CHERI Programmer’s Guide*, it is necessary to *mmap()* a region with full access and restrict as needed via *mprotect()*. WAMR typically reserves a memory block with *PROT\_NONE* therefore its *mmap()* wrapper had to change
6. Low-level Native Code Interface
   1. WAMR marshals calls to native code by setting up a restricted stack and heap environment; the wrapping for this is implemented in assembler specific for each target
   2. For Morello pure-cap a new assembler file was required with C64 instructions to manage the stack and transfer execution.
   3. This was further complicated by the fact that WAMR would attempt to pass pointers to native code as if they were integers, which on CHERI they are not, and therefore the entire native calling system had to be re-worked when running under pure-cap.

## WAMR AOT Compiler Issues

WAMR AOT compiler uses LLVM backend libraries to generate LLVM IR from parsed WASM, and then compile LLVM IR to target specific object code. The AOT file itself is, on Linux, made up of some ELF sections and headers and WASM formatted data sections.

The build setup was complicated by the fact that for Morello *wamrc* runs on Linux x86\_64 but cross-compiles for Morello. This brought with it a number of specific problems:

1. New Target Options
   1. It was undocumented how the LLVM Morello target worked, which made it more difficult to correctly create the target-triple and add the correct feature set using the LLVM C++ API
2. Use of non-default Address Space
   1. Pure-cap target requires use of a different LLVM Address Space than zero (200 is used)
   2. WAMRc code wrongly assumes the AS is always 0, and so this had to be changed
3. Structure and Data Alignment consistency with WAMR
   1. WAMR native code has access to the *execution environment* at runtime, which provides module information in a *WASMModule* structure.
   2. When generating object code, the fields in this structure must align at AOT compile time vs WAMR runtime and therefore the WAMRc structures had to change to ensure that pointers would be aligned and sized correctly to map to a pure-cap execution environment (which is not the same as a 64-bit pointer in the compile environment)

AOT mode was complicated to develop and debug, because the compiled object code bears little resemblance to the original WASM (as WASM Text format). Additional difficulties on the WAMR side were implementing the new symbol resolution types which are Morello specific, and ensuring the ELF header and other sections were correctly formatted. Further, it was important to ensure *wamrc* – which, unlike WAMR, is not built for a specific platform – would still be able to generate AOT for other targets, which added to the difficulty of the problem.

## Conclusions

It appears to have been assumed in the literature that adapting an application to work with CHERI will be relatively trouble-free however we can see from WAMR that real-world, bloated codebases which make use of low-level operations show this is not the case. Sadly, in the real-world, and in particular with embedded systems, many developers treat pointers to be the same as an integer and freely convert between the two and utilise architecture-specific tricks to make things work in an optimal manner. This vastly increases the complexity of porting such a codebase to a CHERI architecture.

It has been found that to tackle such a problem a good understanding of the codebase and the CHERI architecture is essential, and it is necessary to debug problems and “think outside the box” when issues such as bus errors or segmentation violations are encountered. Bus errors would seem to be the greatest issue due to pointer misalignment, but finding the root cause can be far from trivial.

Such projects are even more difficult with an early stage research platform, however through this work it has been found that the SoC has worked well with no issues caused by the HW. Instead, it has been the toolchains which have proved problematic at times (both generation of object code and debugging). Frequently when a problem was encountered it was first assumed that the fault was in the silicon, but this was never found to be the case.

In all aspects though, better and more up-to-date documentation, both for the codebase being ported and the CHERI architecture itself will greatly improve the issues described. More examples of possible problems and their solutions would be beneficial for CHERI, and specifically for Morello which brings its own complexities. As the Morello architecture is rolled out further then it is likely more and more complex projects will be ported to use it. It is hoped that some of the material, and the setup of the development environment, utilised in this WAMR porting project will be useful for others.

# Development, Test & Debug Environment

This section covers the development environment used, explains challenges in setting up the Morello Hardware and provides an overview of testing which has been performed to validate the porting has been successful.

## Setting up the Morello SDP

The Arm Morello release 1.5 was used for this, as this was the latest official release at the time the project commenced. Reference:

<https://git.morello-project.org/morello/docs/-/blob/morello/release-1.5/user-guide.rst>

### Configuring the Host Machine

Although Arm informed that a PC running a native Ubuntu OS was required, due to availability of such HW the project work was carried out using *Windows Subsystem for Linux* (WSL2) hosted under Windows 10. An Ubuntu 20.04 LTS distribution was installed from <https://releases.ubuntu.com/focal/> on WSL2.

Although almost everything can be achieved with WSL2, there are a couple of issues which required a Linux VM:

* WSL2 does not support preparing a USB Flash drive image or formatting a SATA Hard Disk (at least, this is far from trivial on Windows 10 – may be possible on Windows 11), which is needed to configure the Morello board
* when building the Linux Pure-Cap image there are issues configuring the disk mounts to the WSL2 environment. This either requires modification to the dockerfile supplied in the build, or building on a Linux VM for Pure-Cap. For the project work the dockerfile modification were performed, but if regular re-builds are needed with new versions then it may be better to avoid modifying the Arm release files

For the project work an Oracle VirtualBox VM was used, note that it is important to additionally install *VirtualBox Guest Ad-ins* in order to be able to access PC USB drives from the VM.

The remaining setup was as described in the Arm Morello release notes.

#### WSL2 GUI

In order to support a limited GUI from a WSL2 installation an X-server is needed. For this project, *VcXsrv* was used which is available from <https://sourceforge.net/projects/vcxsrv/>. Setup is as described in VcXsrv documentation, however one note is that the following should be added to the user’s *.bashrc* file to get a terminal:

**export DISPLAY=$(ip route|awk '/^default/{print $3}'):0.0**

### Building the SW Stack

The SW Stack for Linux Pure-cap was built largely as described in the Arm Morello release documentation, and also here: <https://git.morello-project.org/morello/morello-aarch64>.

There were a few issues as described below:

#### Unusable recommended Debian Version

The version which is downloaded from <http://deb.debian.org/debian> is “*stable”*, however this contains an obsolete *glibc* version which is not suitable for e.g running *gdb-server*. It is better to use the Debian “*testing”* branch.

This can be achieved either by modifying the version specified in the *qemu-debootstrap* build command in the Dockerfile:

**RUN qemu-debootstrap --arch arm64 testing "$debian\_dir" "http://deb.debian.org/debian/"**

or, by updating the version once the Morello board has been setup:

* Updated existing packages with *apt update && apt upgrade -y*
* Edit */etc/apt/sources.list* and modify **stable** to **testing** in Debian occurrences
* Run *apt update* to update the repository index, when complete reboot the board

#### Docker Volume Mount Issues: WSL2 Only

The Linux Pure-cap build is done in a Docker container. This is configured to set up a Docker Volume so that the container’s output folder is accessible from the host Ubuntu distribution.

The problem is that in WSL2 on Windows 10 (and possible Windows 11 – but this is unknown), it is not possible to use a docker volume on a folder which is not mounted to the native Windows file system.

The build scripts bind mount two docker volumes when the container is created:

* The *<morello-workspace>/morello-aarch64* folder which is used read-only
* The *<morello-workspace>/output/<soc|fvp>/intermediates* folder which is used to map the built image back to the host machine (i.e outside the docker container)

The approach taken was to modify the dockerfile to copy the read-only file to the image and map a volume on a windows shared drive for the output volume.

**NOTE:** *It must be stressed this process is changing released files – if a native Linux PC is not available then a better approach is arguably to build using a VM. Failing that, building on a folder shared between WSL2 and Windows (e.g /mnt/c/<folder>) would eliminate the problem although the build would be up to an order of magnitude slower.*

Modifying the dockerfile to make the build work was done as follows:

1. Edit the called build script, which is *<morello-workspace>/build-scripts/build-debian.sh*
2. Observe there are two lines related to docker; these first of all build the docker image:  
   **docker build -t “$docker\_container\_name” “$docker\_env\_dir”**

and then create & launch a new container based on the built image:

**docker run –-rm -v $docker\_mnt\_pt:/morello-aarch64:ro \**

**-v “$PLATFORM\_OUT\_DIR/intermediates”:/out “$docker\_container\_name”**

1. The *docker run* command is modified as follows to make use of a windows 10 mapped volume only, let’s call it temp/docker, so modify the *docker run* line as follows (this assumes the folder to be used is *c:\temp\docker*):

**docker run –-rm -v “/mnt/c/temp/docker”:/out “$docker\_container\_name”**

1. Then the file *<morello-workspace>/build-scripts/config/Debian/Dockerfile* is edited; at the bottom of the file, *before* the ENTRYPOINT, add the following line then save & close the file:

**COPY <morello-workspace>/morello-aarch64/ /morello-arch64/**

1. Finally, update the *<morello-workspace>/build-scripts/build-debian.sh* to create the temporary folder *c:\temp\docker* and copy the generated image back when it is done:
   1. Add the following line ahead of the “Building the docker container” echo:

**mkdir -p /mnt/c/temp/docker**

* 1. Add the following line after the “docker run” command:

**cp /mnt/c/temp/docker/Debian.root.img $PLATFORM\_OUT\_DIR/intermediates**

* 1. And add the following to the list of folders to clean under the *do\_clean()* function:

**“/mnt/c/temp/docker”**

1. You can now run the build. Note that you MUST at least clean the build-debian.sh part, i.e first run:

**./build-scripts/build-debian.sh -f Debian -p fvp|soc clean**

And then run the build-all script.

### Setting up the Morello Board

In the main the Arm instructions were followed verbatim for the setup, however there were some differences as for the project a Windows PC was being used and the Morello board was not on a wired LAN. This section describes additional steps which were carried out to make the system work with a Windows Laptop host.

#### Wireless Morello Internet

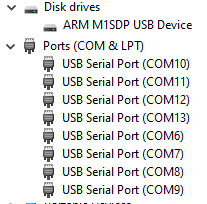
Two approaches were considered:

1. Connect the Morello to the Windows PC via the Host PC ethernet port and use the PC as a *network bridge* (there are many documents on how to do this in e.g Windows 10)
2. Obtain an ethernet-to-wifi HW bridge. These can be configured to provide the board with a static IP; configure the device from the laptop and then simply connect it to the Morello board and it can be used with a Wireless router

The second approach was chosen as it does not require that the host (a laptop) have to be physically alongside the Morello board at all times.

#### Host Communications

When connecting the Morello board to the PC via USB then all eight COM ports and the virtual disk will simply appear in device manager:



Any suitable terminal emulator can then be used to access the COM ports, for the project PuTTy was used (<https://www.putty.org>) and also the PuTTy session manager (<https://puttysm.sourceforge.io>) was used to be able to easily launch ports of interest.

For each of the COM ports the settings are:

* 115200 Baud
* 8 Data, 1 Stop, No parity
* No flow-control
* Default line endings (no implicit CR/LF required)

***Note****: If the first port, shown in the above picture as COM6, is configured to always be on the same COM port number then the same sequence appears each time the Morello is connected (i.e the port numbers persist).*

#### Access to the Morello SSD

The Arm guide suggests connecting the SATA cable directly into the host machine – not possible with the majority of modern laptops.

Instead, the following was done:

1. Remove the SSD:
   1. Remove the Morello PC side panel which is not the glass one
   2. The SSD is simply stuck on to the side and can easily be disconnected and removed
2. Use a SATA to USB adaptor suitable for a 2.5” SSD to connect to your PC
   1. This will result in the SSD coming up as a virtual disk on the PC

#### Setting up the Morello SSD using MS Windows

As described in the Arm documentation, the Morello SSD must be formatted as *ext4*. However as mentioned earlier in this document, this had to be done using the VirtualBox VM as it was not possible from WSL2.

This is achieved by using the SATA to USB adapter as described above, then closing any windows program using the disk (e.g if explorer is auto-launched) and then make the virtual disk appear to the VM. Then run *lsblk* to identify the disk on the VM and *fdisk* to be able to create an initial partition and format it.

#### Updating Morello Internal SD

This appears in Windows Device Manager as a virtual USB disk, but it is necessary to use the VM to be able to access the disk. As above, mount the USB drive for the VM to access ensuring windows is not accessing the disk. The built firmware images can then be copied to the SD card from the VM.

***Note:*** *If the build is done on WSL2 then it will be necessary to first copy the images to the VM using a shared drive that both windows and the VM can access.*

#### Creating a Bootable Drive

As per the Arm Morello guide, it is necessary to use create a bootable image on a USB Flash Drive. Under Windows 10 this step would also need to be carried out on the VM as follows:

1. Configure the USB drive to be visible as a USB device on the VM and use *lsblk* to find it
2. It is then likely the USB drive is formatted for DOS (probably *FAT32*) so use *fdisk* to:
   1. Delete the existing partition
   2. Remove the partition table and create a new one
   3. Create a new *ext4* partition
3. Proceed using *dd* to write the image onto the USB flash drive.

### Installing Debian Pure-Cap on the Morello SSD

It was swiftly found to be not practical to have to boot Morello from a USB Flash Drive each time; instead an install of the OS was needed on the hard disk so Morello could boot straight into it and it could be modified as required.

Arm documentation recommends the use of *wgets* to download a Debian image, but this was found to be far too slow when running of the USB flash stick. Instead the following process was carried out – recommended for other users who want to have a similar install:

1. Obtain a second USB flash drive, with at least 4GB in size. Access it on your Linux VM using method described previously.
2. Copy the *debian.img* file onto this flash stick, e.g:

**cp <morello-workspace>/output/soc/debian.img /media/usb**

1. Remove the flash drive from your host PC, then boot the Morello board from your *other* flash drive (i.e the one created previously) so that you can now access the shell from the AP terminal window in e.g PuTTy.
2. Insert the flash drive which you have just copied *debian.img* onto. Use *lsblk* to identify this flash drive and then mount it so it is available on the morello board, e.g:

**mkdir /mnt**

**mount /dev/sdx1 /mnt**

1. Use *lsblk* again to identify the Morello SSD, and then use *dd* command in order to burn the d*ebian.img* onto the SSD, e.g:

**dd if=/mnt/sd<flash>/Debian.img of=/dev/sd<ssd> conv=fsync bs=1M**

**sync**

1. When this has completed, unmount the flash drive which contained the *debian.img* and power off the Morello board then remove the other flash drive as well. Then reboot the Morello board; this time the bootloader should (after a pause) boot directly from the SSD into the Debian pure-cap image
   1. Note that you can use the MCC “shutdown” command and then “reboot”, you don’t need to physically remove power to the whole Morello PC

### SSH access to the Morello Board

It was found to be inconvenient to only be able to access the Morello via an AP terminal on the host. Instead, SSH access was preferred using the local network so that an SSH client (e.g PuTTy again) could be used on the host, and file transfer and remote jobs could be run on the Morello board across an SSH connection.

This setup is described in this section.

#### Setting up an SSH Server

1. Log in to the Morello board via the AP terminal, then check to see if SSH server is installed, and see if it is running and start it if not:

**which sshd**

**systemctl status ssh**

**systemctl start ssh # Start server if installed but not running**

1. If an install is required, then install and configure as follows:

**apt install openssh-server**

**systemctl enable ssh –now**

**systemctl start ssh**

#### Obtain Morello’s IP Address

The IP address may already be known if e.g a bridge was used; if not use *ifconfig* (may need installing) on Morello to obtain the address – look for *inet* address in the *link/ether* connection.

On a wireless network the address is likely to be in the format *192.168.0.X.*

***NOTE:*** *Configure your WiFi-Ethenet Bridge / Router, as applicable, to give the Morello a static IP address on the local network. Assuming using DHCP, this may be as simple as reserving the IP for the MAC address of the bridge. You may also need to configure the Morello Linux install to set a static IP address in ethernet network settings.*

#### Connect via SSH

From the host use e.g PuTTy to open an SSH connection to the Morello’s IP address.

The login is *root* and the password is *morello*.

You may also wish to login via a key – in this case use e.g *ssh-keygen* on the host to generate a key pair, transfer the public part to Morello and store in */root/.ssh* and configure the SSH server to accept connections using a key.

Files can be transferred to Morello using e.g *scp* from the host (WSL2 or VM).

## Fork of the WAMR Repository

A private “fork” of the *ByteCodeAlliance* WAMR repository has been made (as github does not allow private forks this is actually a private repository with a mirror-push of a clone of the WAMR public repo). This is so that a new platform, *linux-cheri-purecap,* can be developed to support CHERI Morello Linux pure-cap (and hybrid-cap).

This can be found here: <https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/>

This repository is public; anybody has read access.

Within this repo, the *main* branch contains the latest fully verified changes and *develop* is the latest “bleeding-edge” working branch.

## Cross-Compilation for Morello

The Morello board does not present a suitable SW development environment, not all necessary tools to easily support development and therefore cross-compilation on the host machine is used with remote debugging / executable file pushes onto the embedded Morello target.

For this project the host machine is a WSL2 installation running Ubuntu Linux on an x86\_64 architecture.

Two toolchains are available for cross-compilation:

* LLVM/Clang : This is the official Arm release
  + Release binaries for x86 are available from <https://git.morello-project.org/morello/llvm-project-releases/-/tree/morello/linux-release-1.5> (or later versions)
  + Source code to build locally available from <https://git.morello-project.org/morello/llvm-project>, it is necessary to set the correct options to compile for x86 binaries
  + MUSL-libC is also required, source can be downloaded from <https://git.morello-project.org/morello/musl-libc/-/tree/morello-release-1.5.0>
* GNU Toolchain : Binaries and source are available from <https://developer.arm.com/downloads/-/arm-gnu-toolchain-for-morello-downloads> (for both x86 and AArch64)

For this project, the GNU Toolchain was preferred because:

* At project commencement LLVM did not support pure-cap compilation of C++ and the options to use it where more complicated (as MUSL had to be specified as a sysroot)
* There was an issue with *lldb* whereby execution would not halt at breakpoints for pure-cap images

LLVM Toolchain is now supported for building the Verifoxx CHERI WAMR, although GNU is still the default toolchain used – however note that for benchmarking, LLVM is needed because the GNU Toolchain (which is only an evaluation alpha release) is not sufficiently optimising the codebase for performance.

***NOTE:*** *When building for Morello pure-cap it is necessary to statically link all libraries as the pure-cap ABI is not finalised and so only A64 dynamic libraries are available on the Morello Linux installation. In the case that a library is not statically available, a RUNPATH / RPATH must be explicitly specified in the ELF and then C64 libraries placed into this folder (which should be elsewhere than the default LD\_LIBRARY search path)*.

### GNU Toolchain Setup

Setting up the GNU Toolchain was largely as documented in the binary source repository, however there were some issues that should be noted:

1. It is recommended to add the install folder to the *PATH* defined in .bashrc. For headless shells, e.g as would be used when remote building on WSL2 using Visual Studio, these paths must still be setup. Example to achieve this is:

**# Prefer GNU cross compile first – My GNU Toolchain installed in /cross-morello-gnu**

**export PATH="/cross-morello-gnu/bin:/cross-morello-gnu/aarch64-none-linux-gnu/bin:$PATH"**

**alias gcc='aarch64-none-linux-gnu-gcc'**

**alias g++='aarch64-none-linux-gnu-g++'**

**alias gdb='aarch64-none-linux-gnu-gdb'**

**[[ $- != \*i\* ]] && return # Exit on no terminal**

1. Remote debugging is used which requires *gdb* on the client (i.e the host) and *gdbserver* installed and running on the Morello board. Note that *gdbserver* has a dependency on *libc.so.6* but the version supplied with a Debian Stable installation is too old – previously in this document it was described how the Debian Testing release was used instead.

### LLVM Toolchain

For LLVM it is first necessary to build the MUSL-lib as per the *Readme.md* document from the MUSL-lib release. Note that for pure-cap builds then *lib-shim* is disabled; suitable commands to build MUSL-lib would be as follows:

**mkdir musl-bin # Make a build folder**

**cd musl-libc**

**# Configure without libshim (assume clang located as shown)**

**CC=../toolchain/llvm-project-releases-morello-linux-release-1.5/bin/clang ./configure --disable-shared --enable-morello --disable-libshim --target=arch64-linux-gnu --prefix=../musl-bin**

**make**

**make install**

Since the project commenced, using LLVM for cross-compilation has become somewhat simpler. To compile with LLVM the following command is now sufficient to, for example, build a simple C++ program for pure-cap:

**clang++ -target aarch64-unknown-linux-musl\_purecap \**

**-march=morello+c64 \**

**--sysroot /path/to/musl-bin \**

**main.cpp -o main \**

**-lunwind -lc++abi -static**

As was the case for the GNU Toolchain, LLVM needs to be on the system path even when a “headless shell” is used. It is also convenient to define environment variables which can be used to find the *sysroot* in e.g a *CMake* build. Typical *.bashrc* changes may look like the below:

**# Assume LLVM installed in /cross-morello-llvm and musl built to /musl-bin**

**export PATH="/cross-morello-llvm/llvm-project-releases-morello-linux-release-1.6/bin:$PATH"**

**export CHERI\_LLVM\_TOOLCHAIN\_DIR=/cross-morello-llvm/llvm-project-releases-morello-linux-release-1.6**

**export CHERI\_MUSL\_TOOLCHAIN\_DIR=/musl-bin**

**[[ $- != \*i\* ]] && return # Exit on no terminal**

**IMPORTANT NOTE:** A *WASI-SDK* supplied LLVM version (with a WASM target backend) is used to build WASM from C code. *WASI-SDK* uses *clang* and therefore if using the LLVM toolchain version shown above be aware this will break any *WASI-SDK* compilation as the wrong version of clang will be referenced.

## Remote Debugging

For this project debugging was performed remotely, making use of the fact that the Morello board is available on a static IP on the local network and running an SSH server (as described during *Morello Board Setup* steps).

Remote debugging can either use *gdb* (GNU) or *lldb* (LLVM). For this project *gdb* was primarily as:

1. *lldb* would not halt at breakpoints when debugging pure-cap programs (bug reported to Arm)
2. *gdb* is better supported by the IDE chosen, Microsoft Visual Studio

### Using Gdb

To run *gdbserver* on Morello, a one-time setup step is needed. The following command is run on Morello target (assuming logged in as *root*, which is the default):

**sysctl cheri.ptrace\_forge\_cap=1**

Assuming the program is cross-compiled, it must be built with debugging symbols (*-g* option) and then transferred to the Morello board with e.g *scp*.

The server is then launched on Morello with the following command:

**gdbserver localhost:<port> <binary>**

Where:

* <port> is the debug TCP port to use, e.g 2000
* <binary> is the executable to debug

On the host, *gdb* is run as follows:

**gdb --eval-command="target remote <ip\_addr\_of\_target\_from\_host>:<port> <binary>**

Where:

* <ip\_addr\_of\_target\_from\_host> is the IP address of the Morello Board
* <port> is the TCP port the gdbserver is listening on
* <binary> is the same binary file you are debugging on the gdbserver side

Debugging on the host then proceeds as normal. Once debugging ends on the host, *gdbserver* should exit on the target. It can anyway be closed at any time from the host via:

**(gdb) monitor exit**

command within gdb.

### Using lldb

On the Morello board, run:

**lldb-server platform -–listen <port> --server**

Where:

* <port> is the port to listen on, e.g 4096

This starts the LLDB server side.

On the host, run:

**lldb**

To launch the interactive lldb session. Then run the following commands within lldb:

**(lldb) platform select remote-linux**

**(lldb) platform connect connect://<ip\_addr\_of\_target\_from\_host>:<port>**

This will cause lldb to connect to the remote lldb-server, and you should receive confirmation messages that the connection is open. You can verify with:

**(lldb) platform status**

You can then choose a file to debug, which will cause it to be sent to the Morello board, and you can run it:

**(lldb) file <binary>**

**(lldb) run**

You can then carry out any other debugger commands, and when finished you can disconnect from the server as follows:

**(lldb) platform disconnect**

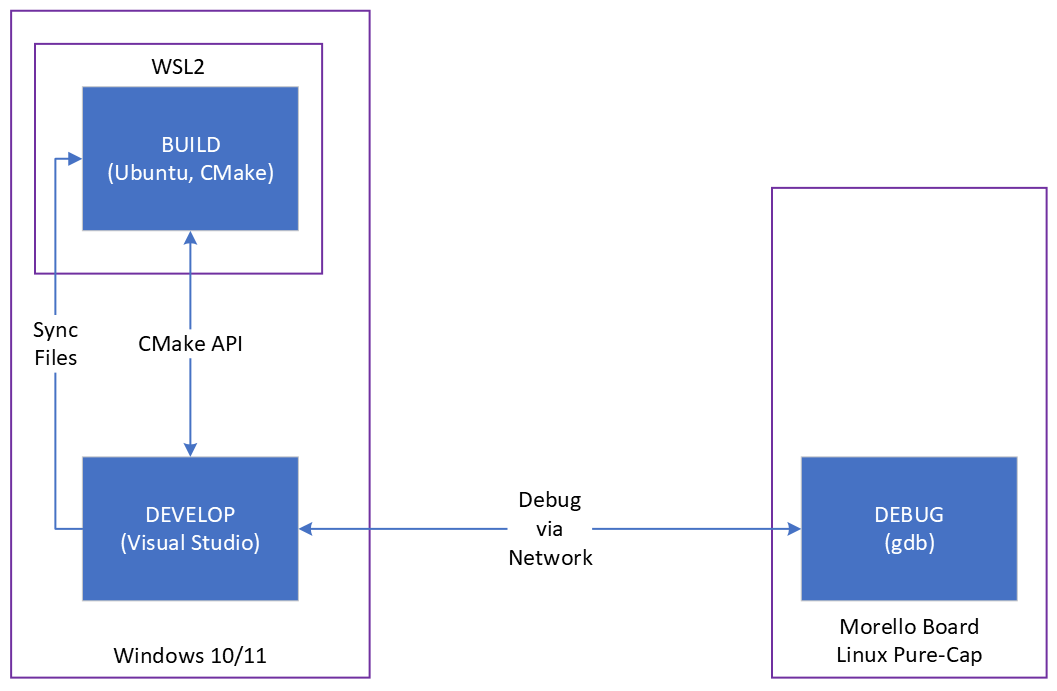
## Visual Studio IDE

WAMR Software changes were performed on Windows using the Visual Studio IDE, with WAMR configured as a *CMake* project with remote builds on a WSL2 Ubuntu Linux distribution using the x86 cross-compilation version of the Morello toolchain.

***Note:*** *As WAMR already uses CMake for its multi-platform build mechanism then it was well suited to this development method.*

Debugging is performed within Visual Studio using *gdb* remotely on the Morello board. Visual Studio was configured to remotely SSH to the Morello machine, transfer the executable and debug.

The development environment flow is shown graphically below:



The whole setup was found to work extremely well. The only downside was that, despite informing it to use a remote toolchain, VS *IntelliSense* still uses a local version of *gcc* for some syntax.

Setting up Visual Studio correctly to perform in the required manner is complex. This section describes key steps that are not part of a standard remote build & debug setup for VS.

### Visual Studio Installation and Setup

Visual Studio 2019 or later is needed, the version used for the project is VS2022. When installing VS it is necessary to include the *“C++ for Linux / CMake Component”*.

*CMake* needs to be installed on the WSL2 Ubuntu distribution, version 3.19 or later (v3.24 was used during the project).

As previously discussed, *gdb* must be setup and configured on the Morello board. Note that, unlike the standalone remote debugging, visual studio can control *gdb* across the network so *gdbserver* is not actually used.

#### Configuring Visual Studio

The *Verifoxx CHERI WAMR* project uses *CMakePresets* instead of the deprecated *CMakeSettings* method and therefore VS should be setup to prefer CMakePresets:

* In Visual Studio go to Tools -> Options and under “CMake” select to use CMakePresets

#### Useful Reference Resources

The following resources proved useful in developing the make system within Visual Studio:

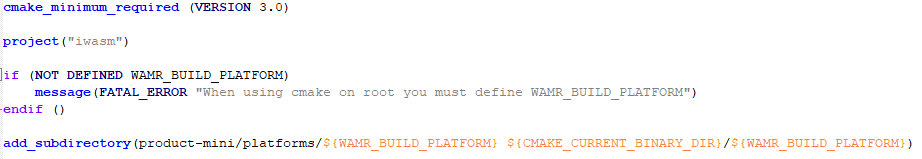
* CMake (“cross-platform make”) is required; this can be thought of as a makefile for makefiles. An understanding of CMake is useful, refer to <https://cmake.org/>
* The visual studio integration with CMake is to transfer files onto the WSL2 target and then use the CMake API to build them. An basic understanding of this process may be useful, see: <https://cmake.org/cmake/help/latest/manual/cmake-file-api.7.html>
* The CMake configuration is done via *CMakePresets.json* and a CMake generator is used for remote build creation, refer specifically to <https://cmake.org/cmake/help/latest/manual/cmake-presets.7.html> and <https://cmake.org/cmake/help/latest/manual/cmake-generators.7.html> for further details.
* Finally, Microsoft provide a reasonable description of creating and using a Linux CMake project: refer to <https://learn.microsoft.com/en-us/cpp/linux/cmake-linux-project?view=msvc-170> and associated pages for more information.

### Verifoxx CHERI WAMR Updated Project Layout

The *ByteCodeAlliance* WAMR project has a *CMakeLists.txt* file for each platform under a sub-folder; for example, for Linux (non-CHERI) it can be found as:

<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/product-mini/platforms/linux/CMakeLists.txt>

However Visual Studio requires a *CMakeLists.txt* at the root level. To support this, *verifoxx-cheri-wamr* adds a *CMakeLists.txt* that includes the platform specific one based on whatever the platform is defined as in the settings (supplied from *CMakePresets.json*). Here is an example snippet:



#### CMakePresets

*CMakePresets* is a *CMake* mechanism to set up configuration settings that are used when parsing *CMakeLists.txt* and other includes *cmake* files.

*CMakePresets* can define multiple configurations, all of which are specified in a file named *CMakePresets.json* which must be placed at the top level of the *CMake* project. When running *CMake* from the command line then a particular configuration can be chosen from the file using *--preset*.

Visual Studio makes use of the configurations in this file by allowing them to be listed in a drop-down box, it is therefore possible to easily choose the build configuration to build on within Visual Studio.

The *verifoxx-cheri-wamr* project includes a *CMakePresets.json* with the following available configurations:

***DEBUG:***

| Configuration | Displayed Name | Use |
| --- | --- | --- |
| ARMc64-hybrid-debug | Debug armC64 Hybrid | Cross-compile for Morello hybrid-cap |
| ARMc64-PureCap-debug | Debug armC64+ PureCap | Cross-compile for Morello pure-cap |
| Linux\_x86\_64\_Debug | Linux Debug x86\_64 | Host build for Linux x86\_64 |
| WAMRc\_Linux\_x86\_64\_Debug | WAMRc Linux Debug x86\_64 | Host build for wamrc AOT compiler |

***RELEASE:***

| Configuration | Displayed Name | Use |
| --- | --- | --- |
| ARMc64-hybrid-release | Release armC64 Hybrid | Cross-compile for Morello hybrid-cap |
| ARMc64-PureCap-release | Release armC64+ PureCap | Cross-compile for Morello pure-cap |
| Linux\_x86\_64\_Release | Linux Release x86\_64 | Host build for Linux x86\_64 |
| WAMRc\_Linux\_x86\_64\_Release | WAMRc Linux Release x86\_64 | Host build for wamrc AOT compiler on Linux x86\_64 |

WAMR uses two *CMake* variables to specify the build *platform* and the build *architecture,* WAMR\_BUILD\_PLATFORM and WAMR\_BUILD\_TARGET (optional and dependent on the platform). The above configurations will set these flags as appropriate to choose the correct platform *CMakeLists.txt* and, where necessary, specify the correct architecture.

#### CMake Build Flags

WAMR defines a number of build flags, to specify what features are included. These are listed in the *CMakePresets.json* file and set appropriately when the default is not adequate.

There are several flags which must be set appropriately on CHERI Morello, as follows:

| Flag | Value | Reason |
| --- | --- | --- |
| WAMR\_DISABLE\_HW\_BOUND\_CHECK | 1 (defaults to 0) | CHERI provides HW checking of stacks and heaps; Morello would not support the SW that this feature would provide if it were included |
| WAMR\_BUILD\_SIMD | 0 | SIMD not supported on Morello |
| WAMR\_BUILD\_LIBC\_WASI | 1 | WASI should always be supported as the Morello test cases are generated from WASI compiled source code |
| WAMR\_BUILD\_DEBUG\_INTERP | 0 | This feature is not yet supported by the CHERI WAMR port. |
| WAMR\_BUILD\_JIT | 0 | JIT is not yet supported by the CHERI WAMR port (note: this will actually default to 0). |

In addition to the *CMake* configuration flags defined in WAMR, some specific flags have been added for *verifoxx-cheri-wamr* which are listed below.

***NOTE:*** *If the flag is not defined, then it takes on the default value in the build.*

| Flag | Default | Meaning |
| --- | --- | --- |
| CHERI\_PURECAP | 0 | Builds for pure-cap if 1, else builds for hybrid-cap (default).   * Ignored for x86\_64 builds |
| CHERI\_STATIC\_BUILD | 1 | If set, build static libraries where possible for Morello (should always be set currently) |
| CHERI\_COMPILER\_FLAGS | *Empty String* | Specific *linux-cheri-purecap* platform build flags if any are required for development |
| WAMR\_APP | 0 | If set, builds the development WAMR front-end instead of *iwasm* official WAMR front-end (**deprecated**). |
| WAMR\_BUILD\_NATIVE\_TEST\_LIB | 0 | If set, additionally builds a dynamic library on the selected platform to be used for WASM-to-native testing of WAMR |
| MORELLO\_PURECAP\_LIBS\_FOLDER | “/purecap-lib” | Only needed for Morello pure-cap builds of the native test library, sets the *RPATH* of the library so that dependencies can be loaded from a dedicated path to avoid loading A64 variants. |
| WAMR\_BUILD\_AOT\_CHERI\_PTR | 0 | If defined and non-zero then pads structures as needed to the supplied byte size to ensure on a pure-cap platform that structures will match having a pointer of this size.  Typically this would be set to 16 when building the AOT WAMR Compiler (*wamrc)* and any WAMR platform *apart from* Morello pure-cap. On Morello purecap it should then be omitted or set to 0. |
| WAMR\_BUILD\_AOT\_EXCEPTION\_WORKAROUND | 0 | If set, then work around a bug found in *ByteCodeAlliance* WAMR AOT handling code |
| WAMRC\_TOOL | 0 | If set then build *wamrc* else build *WAMR*. |
| WAMR\_BUILD\_WITH\_CUSTOM\_LLVM | 0 | Applies for *wamrc* build only.  If set then the LLVM libraries will be sourced from a folder supplied in *LLVM\_DIR* environment variable instead of looking on the path. |
| LLVM\_DIR | “” | Applies only when *WAMR\_BUILD\_WITH\_CUSTOM\_LLVM != 0.*  Specifies a specific folder which should contain *LLVMConfig.cmake;* use when a custom LLVM build is to be used that is not on the search path. |

#### Toolchain Files

The *CMakePresets.json* in *verifoxx-cheri-wamr* references a toolchain file. Three toolchain files have been added to the project:

* *toolchain.cmake* [Default] : Configures the GNU Toolchain for cross-compile to Morello
* *llvm\_toolchain.cmake* : Configures the LLVM Toolchain for cross-compile to Morello
* *llvm\_toolchain\_x86\_64.cmake* : Configures LLVM Toolchain for build reference copy for x86\_64
* *toolchain\_x86\_64.cmake* :Uses the standard Ubuntu Linux GNU toolchain that shipped with the distribution; for building Linux images to run on the host

The toolchain files provided require a number of environment variables to be setup, which would be done from the user’s *.bashrc* file. These are as follows:

***For GNU Morello Cross-Compile (toolchain.cmake)***:

* Set *CHERI\_GNU\_TOOLCHAIN\_DIR* to the folder containing the gcc sysroot. For example in the *.bashrc*:

**export CHERI\_GNU\_TOOLCHAIN\_DIR=/cross-morello-gnu/aarch64-none-linux-gnu**

***For LLVM Toolchain (llvm\_toolchain.cmake)***

* Set *CHERI\_LLVM\_TOOLCHAIN\_DIR* to the folder which is the root of the LLVM installation and set *CHERI\_MUSL\_TOOLCHAIN\_DIR* to the folder which contains the MUSL libC built for Morello pure-cap cross-compilation
* (Optional) Set *CHERI\_MUSL\_TOOLCHAIN\_DIR\_HYBRIDCAP* to the folder containing the MUSL libC built for Morello hybrid-cap compilation, i.e *Aarch64* standard Linux **(only applicable if hybrid builds using LLVM are to be supported)**

For example in .bashrc:

**export CHERI\_LLVM\_TOOLCHAIN\_DIR=/cross-morello-llvm/llvm-project-releases-morello-linux-release-1.6**

**export CHERI\_MUSL\_TOOLCHAIN\_DIR=/musl-bin-morello**

**export CHERI\_MUSL\_TOOLCHAIN\_DIR\_HYBRIDCAP=/musl-bin-aarch64**

***For LLVM Toolchain Linux x86\_64 reference builds (llvm\_toolchain\_x86\_64.cmake)***

* Set *CHERI\_LLVM\_TOOLCHAIN\_DIR* to the folder containing LLVM binaries and *CHERI\_MUSL\_TOOLCHAIN\_DIR* to the folder containing MUSL-lib binaries. For example in *.bashrc*:

**export CHERI\_LLVM\_TOOLCHAIN\_DIR=/cross-morello-llvm/llvm-project-releases-morello-linux-release-1.6**

**export CHERI\_MUSL\_TOOLCHAIN\_DIR=/musl-bin**

Note there are no required settings for the *toolchain\_x86\_64.cmake* file because this will just use the standard Linux install of GCC.

### Remote Debugging on Morello

Visual Studio supports remote debugging on Morello using *gdb,* for hybrid and pure-cap. This requires some setup within VS and a configuration file to provide information to use for the debug session.

To use VS remote debugging there are some pre-requisites:

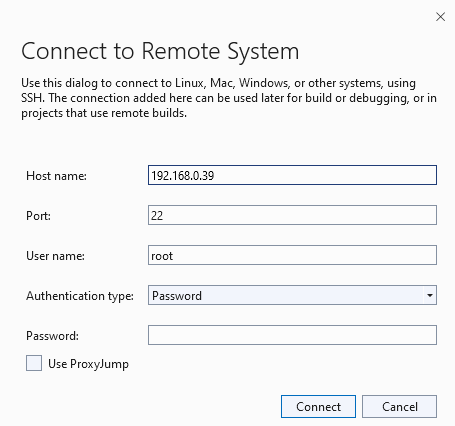
1. Morello board must be visible on the internal network with a known (and ideally static) IP address
2. The Morello board should be running an SSH server and accept remote connections because SSH is used for the connection

Note that you may need to allow access via your firewall on the Windows machine.

#### Adding a Remote Connection

As debugging is not happening on WSL2, you must configure a remote connection in Visual Studio. Note that this is tool, not project, specific. To do this go *Tools -> Options* then in the left hand side choose *Cross Platform -> Connection Manager* and then on the right hand side “Add” to create a new connection.

You will then need to fill in the details to look something like the below:



The Host Name should be the IP address of the Morello board on your internal network. The username is “root” and password “morello”, and by default the standard SSH port (22) is used.

When you choose “connect” VS will make sure it can actually connect to the Morello board, so ensure it is powered on and able to receive a connection. Then the connection will be added to your VS install’s list.

#### Creating a Debugger Launch Session

Visual Studio needs to know how to connect to gdb and run the program to be debugged. This is done by first building the executable and then going to *Debug -> Debug and Launch Settings* which will open a file called *launch.vs.json*.

***NOTE:*** *This file is stored as <project\_folder>/.vs/launch.vs.json*, *should it ever need to be manually edited.*

An example configuration is shown below:



The above example defines two configurations:

1. *Morello Target* : For remote debugging on the Morello board
2. *iwasm* : For debugging on WSL2 Linux

Key parameters for Morello remote debugging are:

* *args* are any arguments you wish to pass to the executable – paths must reference those on the target (also applies for WSL2 debugging)
* *cwd* is the working folder for the debug session (shown as the home location, above)
* *gdbPath* is the path to the GDB program on the target – this must be wherever you installed aarch64 gdb on the Morello board
* *name* is the target name to appear in the VS dropdown
* *projectTarget* will be set to the name of the project
* *remoteMachineName* must match the Host Name set for your Remote Connection (Morello static IP address)
* *type* must be set to *cppgdb* for gdb
* *targetArchitecture* must be *arm64* which is VS’s closest match for Morello
* *externalConsole* is set so the program’s terminal appears in a Visual Studio window
* *deployDirectory* is a folder which the executable will be copied to on the target
* *disableDeploy* should be false to force the executable to be written and persist on the target

#### Running or Debugging the Program

Once a debug session is defined, in the *Select Startup Item* drop-down choose the name of your configuration (e.g “Morello Target”):

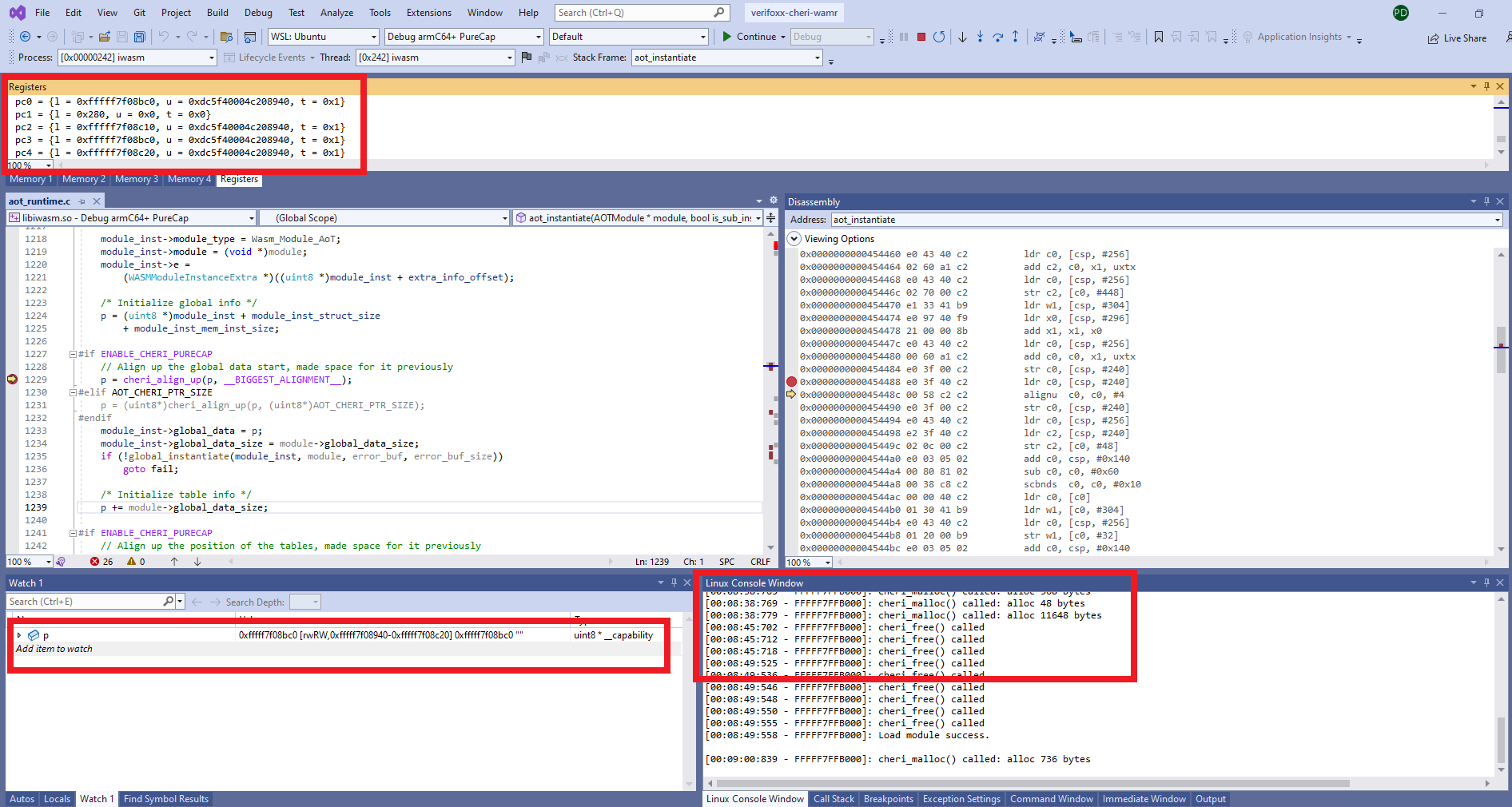


***NOTE:*** *For Debug configurations, a “Debug” box will display alongside this, this won’t be the case for the Release configurations.*

To run a release build, or launch a debug session, click on the green arrow.

Visual Studio will then switch into the release / debug mode, usual VS debug windows are available and you can view program output via the *Linux Console* window.

The screenshot below shows an example of VS debugging of *iwasm* running on Morello as a pure-cap executable. At the point shown, we are about to align a memory block to ensure it will start on a capability pointer size boundary. As well as the source code and disassembly, note (highlighted with red boxes) the display of Morello C64 registers, console output from the Morello target and display of pointer variables with capability information parsed.



### Troubleshooting Visual Studio Build & Debug Issues

During the project work, various small issues were encountered with the Visual Studio setup as described in this document. This section is included as an aid to future projects which may wish to develop Morello SW on Windows with the VS IDE and execute remotely on Morello.

#### Use of Virtual Substitute Windows Drives

Windows allows the use of virtual drives via the DOS *subst* command. However if your *CMake* project is located on a substitute drive then VS will not be able to successfully communicate with the WSL2 Linux distribution and transfer files to it, which will cause the remote WSL2 build to hang or fail.

There are a number of ways to solve this, but by far the easiest is to create a symbolic link on the WSL2 Linux distribution. For example, assuming your windows substituted drive is *N:*, which maps to *C:\Verifoxx*, then proceed as follows on Ubuntu:

**sudo ln -s /mnt/c/Verifoxx /mnt/n**

This should resolve the problem, but you will need to delete and rebuild the *CMake* cache.

#### Problems Launching WSL2 Processes

VS needs to be able to launch the build on the WSL2 machine, but if your *.bashrc* includes terminal output when first logging in then this can stop the build working.

As per the example snippets from *.bashrc* in this document, the solution is to support a “headless” mode when there is no terminal. Add the following line near the top of the *.bashrc* file after first setting any paths which are needed for the VS build:

**[[ $- != \*i\* ]] && return**

#### CMake Generation Problems

If VS *Build* menu options are unavailable, or you have strange CMake errors, try the following things in order to see if they will resolve the problems:

1. Rebuild the CMake cache (*Project -> Delete cache and Reconfigure)*
2. On your WSL2 Linux distribution, delete the entire visual studio folder for your project (or for all projects), this would normally be *$HOME/.vs/<project>*
   1. You will then need to rebuild the CMake cache
3. Restart WSL:
   1. Close Visual Studio
   2. In a windows command shell run *wsl –shutdown*
   3. Relaunch Visual Studio (there may be a delay in opening your project while WSL2 restarts Linux)

#### Remote Debugging Problems

In the case of issues with remote debugging on the Morello target, some points to check:

1. Ensure you have selected the correct target in the *select startup item* drop-down (e.g choose “Morello Target”)
2. Make sure you can still SSH onto the Morello board (and if not, this is a non-VS problem that needs resolving, e.g the IP address has changed)
3. In case *launch.vs.json* has become corrupted, close Visual Studio and check *<project>/.vs* on your windows machine for the file, and ensure it is present and contents are as expected.

## Testing Environment

Through the project a number of *test vectors* have been created. These comprise WASM files which are generated from source code, either C code or WAT (WASM Text format). Both the source and the generated WASM have been included to the *verifoxx-cheri-wamr* github WAMR fork, and can be found in the following folder:

<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/tests/wamr-linux-cheri-purecap-tests>

The WASM files are used for manual testing of WAMR and in addition a number of test vectors are made available for automated testing. Automated testing is done by using a modified version of the WASI Test Suite (<https://github.com/WebAssembly/wasi-testsuite>) and is discussed in more detail in the dedicated WASI Test Suite section, Section 6.2.

Manual testing is performed on Morello by transferring the WASM files to the Morello target with *scp* and then running each test using the *iwasm* program with appropriate command-line arguments. For backwards compatibility tests, *iwasm* is generated for Linux x86\_64 and the test repeated on this platform.

In the case of AOT files (compiled WASM), the WASM test vector is compiled to an AOT file with *wamrc* and then used as an input to the *iwasm* program.

### Generating WASM from C Code

The *WASI-SDK* (<https://github.com/WebAssembly/wasi-sdk>) provides a *clang* compiler which can compile C code to WASM and link in the C runtime and standard library provided by WASI-libc. These will then make use of the corresponding WASI API functions in WAMR’s WASI library implementation.

Each supplied C code test vector in the *verifoxx-cheri-wamr* includes instructions in C code comments on how to compile it to WASM. In general there are three types of compilation used:

1. Compile with full C startup and runtime code:

**clang --target=wasm32-unknown-wasi -o foo.wasm foo.c**

1. Compile without the C standard library or startup code (C code contains no *main()* function and must call WASI functions directly) – optionally specify point with *-Wl,--entry=<function\_name>*:

**clang -Os -nostdlib --target=wasm32 -Wl,--allow-undefined -o foo.wasm foo.c**

1. Compile without linking, this will generate a WASM with no exported function and will require a native library which implements any externs:

**clang --target=wasm32-unknown-wasi -nostdlib -c -o foo.wasm foo.c**

### Generating WASM from WAT

Some of the test vectors are hand-crafted WAT (*WASM Text)* format files.

These are converted to WASM using the *wat2wasm* tool which is supplied with the *Webassembly Binary Toolkit* (<https://github.com/WebAssembly/wabt>):

**wat2wasm -o bar.wasm bar.wat**

*WABT* contains a number of useful tools for converting WASM <-> WAT and also a simple WASM interpreter and has been invaluable in working on this WAMR porting project.

# Generic Porting Steps

In this section we describe the initial porting steps and changes made which were needed to make WAMR start to work on Linux CHERI pure-cap. Although WAMR’s classic interpreter mode was the first milestone for porting, the work described here is not specific to any mode of execution.

Following an initial analysis of the WAMR codebase, and attempts to test & debug issues with the classic interpreter, the following issues were identified:

1. Linux on CHERI Morello is so different from Linux on any existing platform, that a new “product type” would be needed. This would also avoid overly complicating the existing Linux product.
2. WAMR freely uses “blobs” of allocated memory and these contain pointers. To avoid a bus error, these must then be aligned to a capability pointer boundary (16 bytes on Morello), whereas for WAMR they would only be aligned on a machine native word boundary (8 bytes for AArch64).
3. WAMR treats pointers as integers in numerous places; for CHERI this would cause an invalidation of the capability tag if casting a pointer to an integer (a [*u]intprt\_t* is needed instead)
4. Control of WAMR allocated memory regions is required to ensure these can be given a full capability pointer, and also to make it easier to support future work of managing all WAMR memory areas.

Note the last point refers to potential future work – this is the capability modelling involved in compartmentalisation of each WASM module sandbox. Some steps were put in place as a driver towards these work, when there would be **no detriment** to WAMR in doing it and where it would bring benefits (i.e reducing risk) of potential future work.

## Building the new WAMR Platform

Following *ByteCodeAlliance* instructions from <https://github.com/bytecodealliance/wasm-micro-runtime/blob/main/doc/port_wamr.md>, a *linux-cheri-purecap* product was created (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/product-mini/platforms/linux-cheri-purecap>).

### CMakeLists.txt

A new *CMakeLists.txt* file is needed for the new platform. The *linux* platform one was used as a base for this.

The *WAMR\_BUILD\_PLATFORM* is then set to *linux-cheri-purecap* to utilise this product, and the *WAMR\_BUILD\_TARGET* is set to *AARCH64* as this is an AArch64 base architecture.

The *CMakeLists.txt* was setup to build for hybrid-cap or pure-cap, and is designed to utilise a *CMake* toolchain file. With reference to Section 3.5.2.2, the following CMake variables are applicable:

* CHERI\_PURECAP : Select pure-cap mode when enabled, causes the C preprocessor macro *ENABLE\_CHERI\_PURECAP* to be defined and the compiler flags to be set accordingly:
  1. *–march=morello+c64, --mabi=purecap* for purecap, or
  2. *–march=morello* for hybrid
* CHERI\_STATIC\_BUILD : add the *–static* option to the build flags to build static libraries (which is needed for pure-cap)

### Platform API Folder

As per WAMR porting instructions, <https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/core/shared/platform/linux-cheri-purecap> is added.

Although similar to the *linux* platform variant, there are key differences:

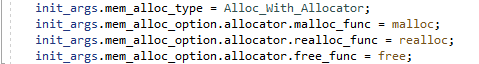
1. This is where *ENABLE\_CHERI\_PURECAP* compiler preprocessor macro is added if *CHERI\_PURECAP* CMake flag was defined
2. Due to the need for bespoke memory management on CHERI, it is not possible to add the *common\posix* platform source code; therefore these are added individually (all but the *posix\_malloc.c* are added)
3. Platform specific files are added to support CHERI memory management; these are described further below

Note that *platform\_api\_vmcore.h* uses the *shared/* version.

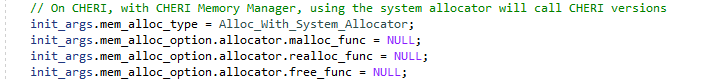
### Program Entry Point

The WAMR standalone runtime, *iwasm*, implements *main()* in a platform specific *main.c* however POSIX platforms such as *linux* make use of a common flavour.

In the *linux-cheri-purecap* case we implement <https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/product-mini/platforms/linux-cheri-purecap/main.c> because of the need to use the CHERI bespoke memory management functions. In the POSIX version, we can see that by default using the system allocator will fallback to use built-in C standard library memory allocation functions:



Whereas for *linux-cheri-purecap* these function pointers must not be defined because then the built-in CHERI flavours will be used:



This slight change means that, unfortunately, the entire POSIX *main.c* file needs to be replaced with the bespoke version.

### Build Warnings

Building WAMR on CHERI pure-cap will emit around 400 *cheri-provenance* warnings:



These can safely be ignored. In almost all cases, they are rather unhelpful as given an assignment is of the form *lhs = rhs* then the behaviour as specified in the warning would be correct.

The project work has not attempted to remove these warnings as first of all there are far too many to be practical, secondly they can simply be ignored and third it is (even with C casting) in some cases not possible to remove them. A future option could be to suppress the warnings in compilation options but this has not been done at the time of writing.

## WAMR Memory Management

Significant changes had to be made to the WAMR’s memory management to support the necessary alignments and CHERI permissions needed to support the capability model. The way WAMR uses memory is first explained, as a reference, and then the changes made are discussed here.

### Understanding WAMR Memory Structures

#### Operand Stack and Stack Frames

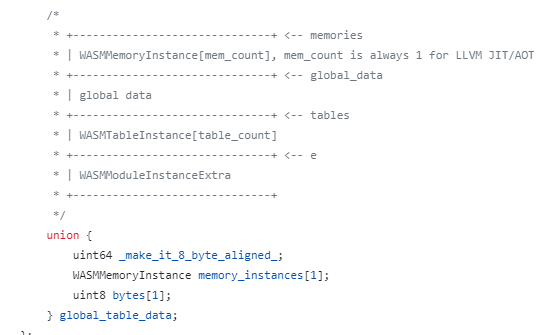
For non-AOT modes, WAMR maintains an operand stack which is allocated onto the bottom of the *WASMExecEnv* structure. Within this stack, WASM maintains a *stack frame* for each call level. The frame structure is *WASMInterpFrame*. Following the convention defined in the [*WASM runtime spec*](https://webassembly.github.io/JS-BigInt-integration/core/exec/runtime.html) (<https://webassembly.github.io/JS-BigInt-integration/core/exec/runtime.html>) this comprises the call frame (parameters), the operand stack and the label stack. This all follows one after the other with no alignment, which needs to be addressed for CHERI.

**NOTE:** The total stack size is known at WAMR runtime startup, therefore the entire stack for the module can be constructed before WASM parsing.

#### Memory, Global, Table & Extra Structures

When WAMR parses the WASM module or AOT file it can encounter 0, 1 or more entries for globals, table entries, memory entries (0 or 1 only as per WASM spec v1.0) and extra data. For each of these it generates a structure type so there may be 0, 1 or more of each of these structure types (0 or 1 only for memory - which may or may not be an import).

These structures are added to the bottom of the allocated space for *WASMModuleInstance* but note that there is no guaranteed alignment between each of them. This is shown below – it can be seen that the structure contains only enough space for a *WASMMemoryInstance* and other structures (shown in the comments) simply appear in the “blob” that is allocated on the (native) heap once the size of other data is known.



***NOTE:*** *For AOT the same structure is used, albeit with an aliased name.*

#### Linear Memory & Application Heap

WAMR needs to allocate space for the Linear Memory (if applicable), the size of which is defined in the WASM module (as number of 64k pages). For this, WAMR will either use the global *malloc*() or allocate a region of *mmap*()ed memory, depending on build config options. The appropriate memory structure is then updated to point to this allocated area.

WAMR also maintains an application heap which can be used by native functions which need to exchange data buffers with WASM module(s). By allocating from this heap, native functions can ensure that the buffer is accessible from the WASM “sandbox” maintained by WAMR (for this reason the native heap cannot be used).

WAMR deals with this heap by extending linear memory by the heap size and reserving an area in linear memory for the heap. If no linear memory is requested, but a heap is requested, WAMR will allocate a single page for the heap which is the size of the heap. In other cases, exactly how and where in the memory WAMR allocates the heap depends on the requested number of pages.

**NOTE:** The heap size is known at WAMR startup time, but the number of pages (if any) is of course only known when parsing the WASM module. Note also that WAMR does not implicitly make any alignment guarantee as to where the heap would start and end, although in practice due to the page size it would be aligned because the page size is aligned.

The system is further complicated by the fact that WAMR implements its own Heap Management Unit (HMU) for this heap. This also needs addressing on CHERI because memory allocated from this internal heap need to be in blocks that are aligned to a capability pointer.

#### Imports, Exports, Globals and Functions

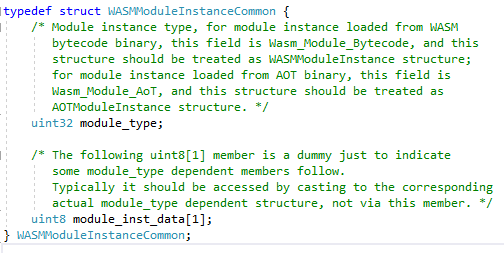
These are allocated by WAMR as part of the WASM parsing into the different areas of the *WASMModule* structure. There are no issues here, but these would need to be addressed in potential future work if control of WAMR capabilities is fully implement.

### Structure Alignments

The first step to bespoke memory management for CHERI is to provide structure alignments to support pure-cap when fields would not otherwise be aligned.

This happens frequently with WAMR because it uses “blobs” of memory within a structure for uses that may contain pointers, or otherwise casts a structure pointer to another which contains pointers. Clearly the compiler cannot know about all these cases at compile time.

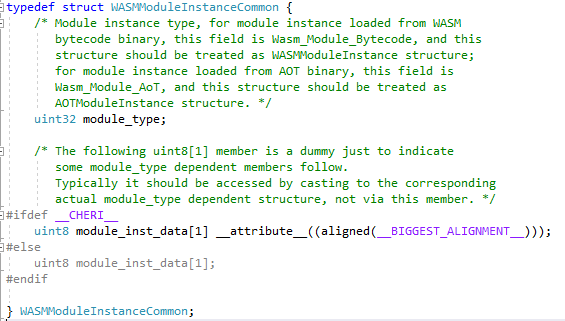
An example is the *WASMModuleInstanceCommon* structure. This “wraps” the actual module structure an includes a type ID for it (an attempt to create a self-describing type):



In this case, if *module\_inst\_data* is used as an offset for the actual module structure then the actual module structure will start on a 32-bit alignment.

In the case of CHERI we can solve this by forcing the compile to align the element to a capability pointer boundary. Since the capability is the largest raw data type, we can use the compiler built-in *\_\_BIGGEST\_ALIGNMENT\_\_* which would then work on any CHERI platform.

The modified code is shown below, making use of the *ENABLE\_CHERI\_PURECAP* flag:



The same technique is used throughout the codebase where structure alignments are needed. Note, though, that this is only where the compiler must be informed that the natural alignment would be incorrect, so if the structure contains pointers then these will automatically be aligned to a capability boundary.

### Pointer Alignments

As explained previously, WAMR will utilise a region of memory to dynamically contain different structures which include pointers. The pointers can then become mis-aligned, instead of being aligned to a capability pointer boundary, which will cause a *bus error* at runtime. Here, we explain this problem in detail and then present the solution which is used throughout the WAMR code when CHERI is enabled.

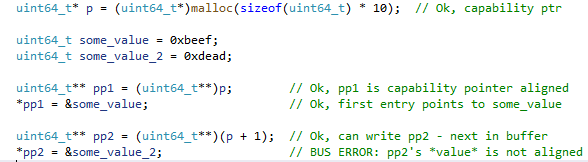
#### The CHERI Pointer Alignment Problem

A pointer references an area of memory. This area does not need to be aligned, but the pointers *address* must be aligned. If not, then when the pointer is *used* then a bus error will result.

On CHERI, the pointer is not the same size as a machine word (64-bit machine word on Morello) and instead the pointer’s location in memory must be aligned to twice the machine word (128-bit on Morello). Any pointer allocated via a direct heap allocation, or placed on the stack, will be aligned because the kernel (*malloc*()) and the compiler can enforce this, and the same applies for pointers within a structure. The problem comes when at runtime an area of memory is used for a different purpose – typically with a cast – to contain a pointer. Everything works fine until the pointer is used, and then a bus error occurs.

Some examples of this are useful. These are – greatly simplified – code snippets that illustrate what happens in WAMR code.

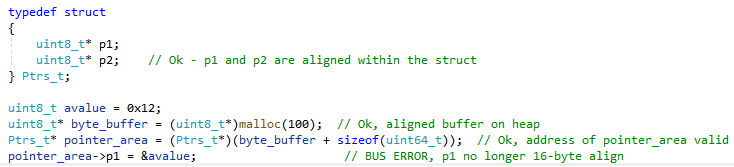
*Case 1: Using buffer of (e.g) Integers to hold Pointers*



In the above example the pointer *pp2* is being set to *&p[1]* and this is then cast to a pointer-to-pointer type. WAMR would expect this to work fine on AArch64 because a sizeof(pointer) == sizeof(uint64). However on CHERI Morello, *&p[1]* is *&p[0]* plus 8 bytes which is not aligned to a 16-byte boundary.

The operation to assign *pp2* will work fine, but when it is *used* (in the next instruction) then the bus error occurs. In a complex system, this can make it tricky to track down these errors.

*Case 2: Structure Misalignment within Buffer*



This case is of the same cause as the previous one, but demonstrates better how structures can become misaligned which is not a problem until they (or sub-structures, of which WAMR has many) contain pointers. Again, until the pointer is used there is no problem which can make it hard to find the root cause at runtime.

*Ptrs\_t* is a structure containing pointers – the compiler will align everything ok on pure-cap so long as the structure begins at a capability boundary.

But in WAMR, typically a byte buffer is allocated and then the structure offset set to some start value within that buffer. Although WAMR will align on AArch64 to a 64-bit boundary that is not sufficient when the structure contains pointers.

In the above we can see that *pointer\_area* is set to an offset into the allocated memory which is not aligned to a capability boundary. Everything would work fine, and in fact if *Ptrs\_t* contained structure members which were not pointers then this would also work. The problem comes when we wish to use a contained pointer as now the *address* of the pointer is not aligned and a bus error results. This is to say that, in the above case, *&(pointer\_area->p1) % 16 != 0*.

In the above example the offset from the start of the allocated byte buffer is static but in WAMR sometimes it is dynamic (depending on the WASM data) and therefore sometimes the pointer addresses may align but other times they may not.

#### Pointer Alignment Solutions

The general solutions applied throughout the WAMR code are as follows:

1. Update sizes, where possible, to ensure the size is aligned up to a capability boundary (at the expense of padding)
2. Allocate enough space to deal with alignment, and align up to a capability boundary before each new region is needed

Note that the system calls to allocate memory always guarantee a capability aligned address, but the problems occur when WAMR uses the memory as a working buffer.

In order to manage the alignments, the compiler built-ins *cheri\_align\_up()* and *cheri\_is\_aligned()* are used. These we presume will always give the most optimal way of performing an alignment.

*Example: Aligning WASMModule Dynamic Region*

We recall the dynamic buffer which is allocated to the bottom of the *WASMModule* structure:

A screenshot of a computer code

Description automatically generated

The start of this buffer will be aligned, but within it the size of *global\_data*, *tables* and *e* may not be.

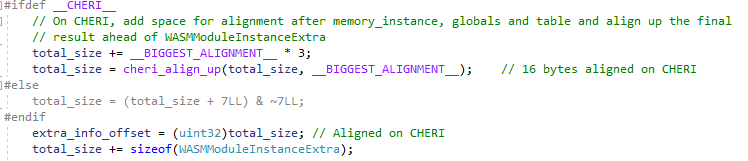
The solution is that when allocating the memory buffer a simple algorithm is applied to ensure there is room for alignment padding:

**alloc\_size = required\_size + 3 \* \_\_BIGGEST\_ALIGNMENT\_\_;**

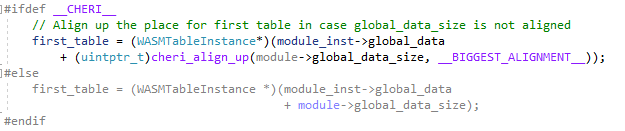
and then when determining the data layout, for example:

**tables\_ptr = cheri\_align\_up(global\_data\_ptr + global\_table\_size, \_\_BIGGEST\_ALIGNMENT\_\_);**

This can be seen in the code by referencing *wasm\_instantiate()* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/core/iwasm/interpreter/wasm_runtime.c>). Key parts of the code:

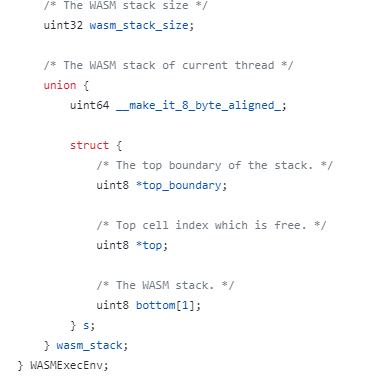


and:



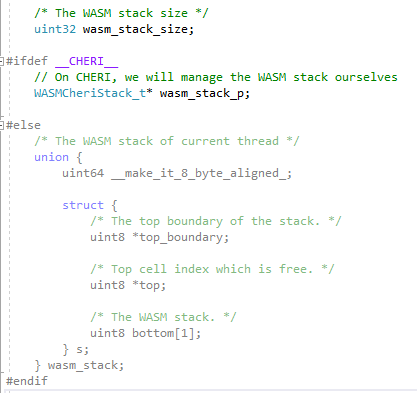
### Operand Stack Changes

The *WASMExecEnv* structure in WAMR contains pointers to a block of memory which is allocated for the operand stack at the bottom of the structure:

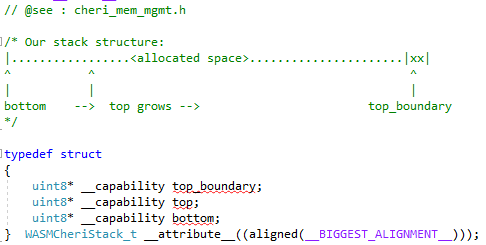


The start of the stack is the “bottom” and then the “top” points to the current location.

For CHERI, this was modified to use an external allocated memory area, so that we can maintain a capability outside of the *WASMExecEnv* to be used for the stack. Here is the revised structure after the changes:



The *WASMCheriStack\_t* structure is similar to the non-CHERI version:



The snippet above also shows how the stack’s memory region is allocated to start at *bottom* and the size == *top\_boundary – bottom*. Initially, *top == bottom* and then as stack is used, *top* grows closer to *top\_boundary*.

The stack itself is then managed, along with the heap functions, in a new C++ class *CheriMemMgr*. This is declared in the platform specific file *core/shared/platform/linux-cheri-purecap/cheri\_mem\_mgmt.h* and implemented in *core/shared/platform/linux-cheri-purecap/cheri\_mem\_mgmt.cpp* (files relative to the project root for the CHERI-WAMR repository).

The class lazily creates the actual memory space for the stack, and the *WASMCheriStack\_t* structure, during the WASM initialization process. The call to *CheriMemMgr::create\_stack\_struct()* will allocate the memory if it has not yet been allocated, using *CheriMemMgr::setup\_wasm\_stack()*.

To support the C API a singleton instance of the *CheriMemMgr* is created via a call to *create\_cheri\_mem\_mgr()*.

The full flow of creating the stack structures is shown below:

A screenshot of a computer program

Description automatically generated

Key points:

1. The singleton instance is managed by a C API wrapper to the C++, this is accessed from existing WAMR code and created as part of the custom *bh\_platform\_init()* called early in the WAMR startup.
2. The process is to set a stack size, once, and ignore repeated calls to set it – this allows for more flexibility in the WAMR code.
3. The *cheri\_wasm\_update\_stack\_size()* actually **updates** the requested size so the stack size will be aligned. This is done *before* allocation because WAMR internally needs to store the *actual* stack size that will be used (this is discussed further in the next sub-section of this document).
4. The actual creation of the stack memory (via *new[]* in C++) is handled after the stack size has previously been set – this is done lazily, the first time the stack is created, but subsequent calls to *cheri\_wasm\_create\_stack\_struct()* will return the *WASMCheriStack\_t* pointer
5. The *CheriMemMgr* class internally handles setting the correct CHERI permissions for the stack pointers, and always clears the contents of allocated memory to zero because WAMR assumes it will be so.
6. The *bounds* of the stack capability pointer are adjusted so the *top* pointer can never go out of bounds even if *top==top\_boundary*. Therefore, in actual fact, the amount of memory allocated is *requested stack size + sizeof(capability)*.

Note that, as described above, the stack (and actually the heap) sizes are handled via the use of *Optional Values*. They are checked to ensure they have been set before being used, and once set cannot be updated.

To support this, a very simple C++ template class *OptionalValue* is implemented. This is a cut-down version of *OptionalValue* that is available from STL in C++17. WAMR, though, builds with C++14 and so we have to roll our own version (see *core/shared/platform/linux-cheri-purecap/optional\_value.h* – path relative to the CHERI-WAMR project root).

The stack is destroyed by freeing the memory allocated for the stack buffer and structure. This occurs automatically when the *CheriMemMgr* singleton instance is destroyed by the C API layer, which itself happens when WAMR calls *bh\_platform\_destroy().* The flow for teardown is shown below:

A diagram of a computer

Description automatically generated

### Stack Size

WAMR passes the stack size from the *main()* function – it is supplied by the user, or default. To support a case where WAMR *vmlib* is being called from native then as discussed the set size function can be called multiple times without error but only the first call will set the actual size used.

When the stack size is set, it is actually updated and then returned to the WAMR calling code. This is because the size is checked *before* any allocation is actually performed, and then the size is stored internally in the *WASMModule* structure.

The process then is:

1. Align-up the provided stack size
2. Store internally in *CheriMemMgr* ahead of the allocation
3. Return the aligned-up size for use by WAMR

Note that the stack size is not automatically increased to deal with large pointer sizes or the need to align – the user must ensure sufficient stack is made available (WAMR provides a build-time option to enable stack usage profiling – CHERI changes do not affect this).

### Frame Stack Allocation

WASM (in non-AOT mode) uses a stack machine. Each time WAMR program execution changes (e.g calling a new function) a new stack *frame* is allocated from the stack memory which was allocated as described above.

In the case of CHERI, this frame needs to be aligned on allocation because during processing the frame will end up containing pointers. To avoid a *bus error* alignment is needed to a capability pointer boundary.

Further, WAMR mandates that frame size requests are aligned to a 64-bit boundary. In the CHERI case, this is extended to double that.

When allocating space on the stack, the compiler built-in *cheri\_align\_up()* is used to skip as many bytes as needed to hit a capability boundary. The function *wasm\_exec\_env\_alloc\_wasm\_frame()* is modified for the CHERI case to handle the process – this function is called from *ALLOC\_FRAME()* used by the runtime.

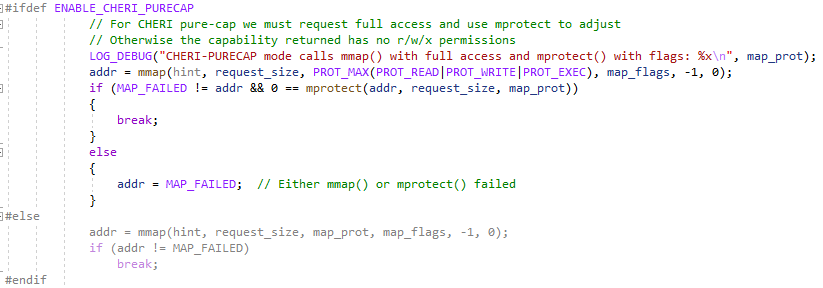
### Memory Mapping

WAMR makes use of *mmap()* via the platform wrapper *os\_mmap()*. Although handling the linear memory in the CHERI port has removed some of these allocations, *mmap()* must still be used on AOT in order to reserve a data area to be used for code in the future.

The usual approach to reserve a memory area with *mmap()* – and that taken by WAMR – is to map with *PROT\_NONE* and then set the protection level needed later. Unfortunately, as documented in the *CHERI Programmer’s Manual*, this cannot work with CHERI due to the inability to “gain” additional permissions not granted by the system.

The known solution is to instead allocate with full permissions and reduce permissions later with *mprotect()*. Additionally, CHERI Morello introduces new permissions and a macro to use them.

To this end, WAMR code is modified so that the POSIX version of *os\_mmap()* is updated as follows when Linux CHERI pure-cap is enabled:



The *PROT\_MAX()* macro is used to apply full permissions and so the returned capability pointer, *addr*, will have sufficient permissions for any usage. On *mmap()* success then *mprotect()* is used to reduce the available usage permissions of the mapped memory without modifying the capability pointer’s permissions.

***NOTE:*** *WAMR code already raises request\_size to be a multiple of the AArch64 page size. AArch64 page size == Morello page size == 4Kbytes therefore there are no additional CHERI changes needed to deal with this.*

### Native Heap Changes

WAMR makes extensive use of the heap in order to allocate memory (i.e, C’s *malloc(), realloc()* and *free()*). For *verifoxx-cheri-wamr* we implement our own native heap handling functions as a wrapper around the standard library functions; this is done to be able to control capability creation (to support potential future work) and to correct manage capability permissions and bounds.

Additionally, to aid profiling, the new heap management maintains a count of all currently allocated bytes. A final check can then be done on WAMR exit to ensure that all allocated memory has been free’d. This option is enabled using WAMR’s built in memory tracing *CMake* flag, *WAMR\_BUILD\_MEMORY\_TRACING*.

***NOTE:*** *WAMR mmap()s the Linear Memory region, however on CHERI we currently assign this directly with malloc(). Linear memory allocation will in future need to be correctly managed with a capability but this is not considered at this time as this would be a potential future work.*

#### WAMR Native Heap Options

WAMR can be built with support for a number of different heap options. If running from native code that calls into WAMR *vmlib* then the memory type can then be selected at runtime. The following table summarises the different types:

| Alloc Type | Internal ID | Use in iwasm |
| --- | --- | --- |
| Memory Pool | Alloc\_With\_Pool | Used if *WAMR\_BUILD\_GLOBAL\_HEAP\_POOL* enabled. Will then use a built-in heap pool. |
| User Allocator | Alloc\_With\_Allocator | Used if *WAMR\_BUILD\_GLOBAL\_HEAP\_POOL* not enabled, but then the user functions are set to the system *malloc, realloc* and *free*. |
| System Defined | Alloc\_With\_System\_Allocator | Not used |

For the CHERI changes, the memory pool option is not currently supported and the system allocator will use the new bespoke memory allocation functions. For this reason, attempting to build with *WAMR\_BUILD\_GLOBAL\_HEAP\_POOL* on the *linux-cheri-purecap* platform will lead to a build *#error* and the system defined option will be the default, but this won’t use C standard library functions.

***NOTE:*** When using the WAMR *vmlib* all options can be defined. WAMR stores the allocation functions internally in a structure – if these are NULL then WAMR will fall back to *os\_malloc()*, *os\_realloc()* and *os\_free()* defined for the platform.

#### Implementation

The *CheriMemMgr* class which handled the WASM operand stack (as described previously) is also used to manage the memory allocation functions.

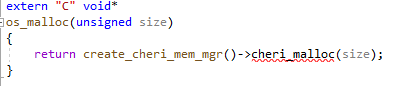
For the *linux-cheri-purecap* platform the common POSIX memory allocation source code is not used. Instead *cheri\_posix\_malloc.cpp* is used. This provides a C API implementation of *os\_malloc()*, *os\_realloc()* and *os\_free()* which call into the *CheriMemMgr* implementation. Therefore in the default case of using the system allocator, we are guaranteed on the CHERI platform to use our bespoke functions.

The bespoke functions are as follows:

* *CheriMemMgr::cheri\_malloc()* (compare *malloc()*)
* *CheriMemMgr::cheri\_realloc()* (compare *realloc()*)
* *CheriMemMgr::cheri\_free()* (compare *free()*)

At this time they simply wrap the system C library functions, but do provide log points and also manage a “live memory allocated” count for profiling (see below for how this works).

Note that, as discussed previously, the *CheriMemMgr* singleton instance used by the C code API is lazily created when needed. Therefore it does not matter if, for example, *os\_malloc()* is called first by WAMR or if the stack is created first. This can be seen from the *os\_malloc()* function for the platform:



*create\_cheri\_mem\_mgr()* returns the *CheriMemMgr* instance, creating it if needed.

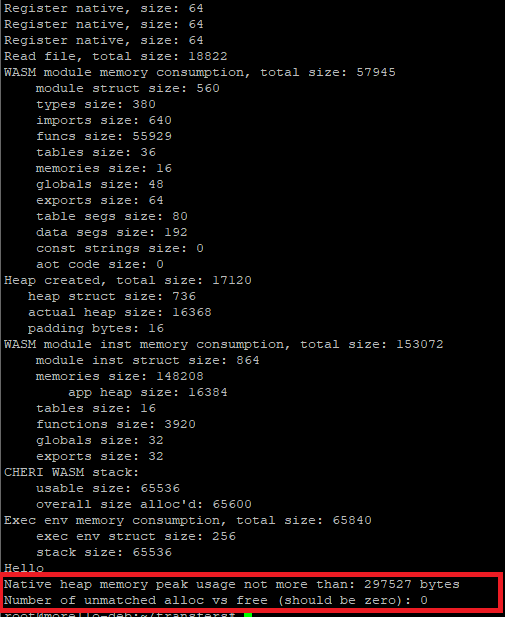
*Maintaining Allocation Count for Profiling*

For profiling purposes a WAMR trace option can be enabled which will cause *CheriMemMgr* to track:

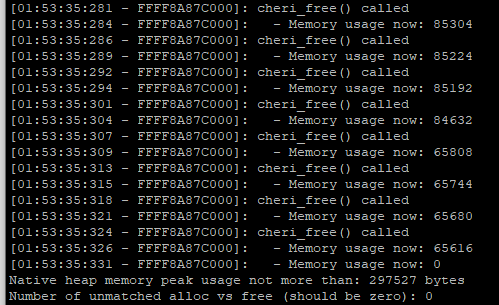
* Total number of allocations vs frees (at program exit this should be 0)
* Total number of bytes allocated vs bytes free’d (at program exit this should be 0)
* Maximum instantaneous size of heap used (at program exit this will be the maximum heap usage, i.e difference between allocations and frees)

The metrics are reported during the cleanup phase of *CheriMemMgr* i.e in its destructor. Therefore when running WAMR’s *iwasm* program, they will be logged just before program exit.

Below is a screenshot of running *iwasm* on the console with the profiling build flag enabled (this simple program echos “Hello” to the terminal). The red box shows the new output for CHERI Morello pure-cap:



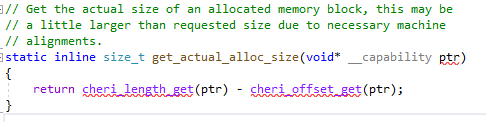
Additionally, the instantaneous heap usage is logged in high verbosity log mode after each allocation and free. This is shown in the snippet from the below screenshot when running the same WASM with *-v==5* argument to *iwasm*:



In order to implement the allocation counters it is necessary to know the requested size for the memory allocation (which is passed in to the function), but also the original requested size for a *free()* and also the original requested size (along with the new size) for a *realloc()*.

This is a problem because we are just wrapping the system functions, therefore there is no way to obtain the original size when *free()* or *realloc()* is called as only the pointer is passed.

Instead, we use the fact that the pointer is a capability and therefore its bounds will yield almost exactly the requested allocation size (due to alignment, this may not be exact). The calculation is performed as shown below:



Due to the possibility of error it is possible that the final reported memory usage may actually be negative (which is clearly impossible) instead of zero.

### WASM Linear Memory and Heap Allocation

WAMR allocates the linear memory area dynamically via memory-mapping. Its size is determined by the number of *WASM memory pages* which are defined in the WASM file that is parsed (each page is 64Kbytes) and the number of memories (although at present WASM spec restricts this to 0 or 1).

The calculation though is further complicated by the fact that WAMR allocates the *application heap* in the same region as the linear memory. The application heap is a sandboxed area for use by native code which performs memory allocation; the native heap cannot be used as it must be a self-contained area within the WAMR VM.

WAMR determines the total size of linear memory needed by using the number of pages requested in the WASM, the command-line argument for user requested application heap and the page size for the architecture (4Kbytes on AArch64). In the case of *linux* platform, which uses memory guard bands for heap protection, then *mmap()* is used but if not (as is the case for *linx-cheri-purecap*) then *malloc()* would be used.

For CHERI platforms there are additional constraints:

1. The offset of the heap in the linear memory allocated region needs to be aligned to a capability pointer boundary
2. The total size of the heap (and linear memory) should be aligned to a capability pointer size

Although in practice it may be the case that other WAMR constraints would align these anyway, this is not relied on in case of future changes in the way the system works.

#### Implementation changes for CHERI

The *CheriMemMgr* handles allocation of the linear memory + application heap block. Although this is handled by a bespoke function, largely to support potential future work when the linear memory capability must be more carefully controlled, currently the linear memory allocation just wraps the lower-level *cheri\_malloc()* function.

Additionally, a function is provided to both check the heap offset within the linear memory block and check the overall required memory size. This memory size is aligned and then passed back to WAMR because it needs to be stored in a WAMR internal structure.

To support the fact that the sizes need to be known ahead of the actual alignment (since if the sizes are not valid / too large, then alignment is skipped) we implement two separate functions:

* *CheriMemMgr::set\_heap\_metrics(<size>, <app\_heap\_offset>)*
* *CheriMemMgr::alloc\_linear\_memory(<size>)*

As mentioned, currently *alloc\_linear\_memory()* simply wraps *cheri\_malloc()* with a log point.

***NOTE:*** *The heap metrics are not used, only checked, at this time. In a potential future work they will be used in the capability management for the linear memory.*

**IMPORTANT NOTE:** The linear memory is also zero’d after allocation, as WAMR assumes all uninitialized memory will contain zero bytes.

There is also a C API to the above functions, the flow is as shown below:

A diagram of a memory

Description automatically generated

Note that the *CheriMemMgr::set\_heap\_metrics()* maintains the stored heap offset and size as *OptionalValue*s (as is done for the stack size). This allows the function to be called numerous times without error before *alloc\_linear\_memory()*, with the values only being set on the first call.

#### Application Heap Management

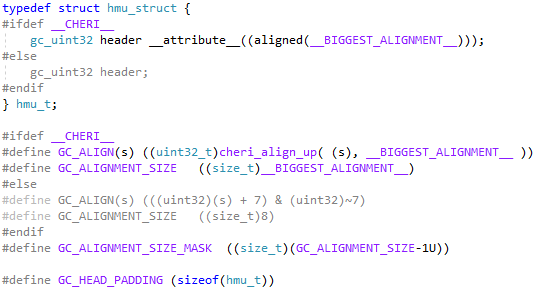
When WASM calls native code within the WAMR sandbox, functions like *malloc()* will be handled by WAMR and will use the application heap instead of the native heap. WAMR implements fairly sophisticated heap management, however to support CHERI changes were needed to this to ensure that heap block allocations are always capability boundary aligned.

Changes needed to achieve these are as follows (reference *ems\_gc.h, ems\_gc\_internal.h* and *ems\_kfc.c* in the *core/shared/mem-alloc/ems* folder, relative to the WAMR project root):

1. The HMU header structure *hmu\_t*, which is before the actual *hmu\_tree\_node\_t* pointers the memory, is a 32-bit integer but this must be padded to be capability aligned on CHERI pure-cap
   1. This is necessary because the *hmu\_tree\_node\_t* structure is *packed* therefore the compiler will not pad it. By aligning the header it forces all of the tree node pointers to be suitably aligned
2. The list arrays within the *gc\_heap\_struct* are also aligned to a capability boundary
3. The heap allocations already ensure a 64-bit alignment, but this is now updated to a capability boundary size (128-bit on Morello) with modifications to *GC\_ALIGN\_SIZE()* and *GC\_ALIGNMENT\_SIZE()* macros

In addition, although not currently being used, if the heap is allocated from a pool then the base address of the pool is aligned up to ensure it is capability boundary aligned (otherwise none of the heap will be aligned). If the heap is simply *malloc()’d* (or *cheri\_malloc()’d*) then it will be correctly aligned by the kernel system call.

Key pre-processor macros and structure changes for when CHERI is enabled are shown below:



## WASM to Native Interface with Morello Assembler

WAMR invokes native functions whenever:

* WASM calls a user native functions
* WASM calls a system native function (e.g a WASI backend function)
* A compiled AOT object code needs to execute

WAMR calls the native function invoker in a similar way to how the C *main()* function is called, whereby the invoker is given an array of arguments (*argv*), the number of arguments *(argc –* actually derived from the function parameter structure) and an area to write the function return value.

The input arguments can be any valid WASM value type, i.e 32-bit integer, 64-bit integer, 32-bit float or 64-bit float. The output is either a 32-bit, 64-bit integer, 32-bit float or 64-bit float – therefore the output is an array of which 0..2 *int32s* are used.

In order to determine the format of the input arguments the invoker is also given a signature string, which can be parsed. Finally, the *WASMExecEnv* structure pointer is also provided as this is made available to native functions as the first argument.

The native function invoker is the function *wasm\_runtime\_invoke\_native()* (see *core/iwasm/common/wasm\_runtime\_common.c,* relative the the project root).

When passing arguments to the native function WAMR needs to reformat the arguments so they can be passed in the correct calling convention for the architecture in question. This is handled by using an assembly code function which takes the buffer of prepared arguments (prepared from the *argv*) and re-works them to match the calling convention of the architecture. Then the actual native function can be called. Note that the underlying native function will expect the *WASMExecEnv\** as the first argument, and then all others to be passed in the normal way (not as a buffer) – for example, here is the function declaration for *wasi\_fd\_fdstat\_get()*:

**wasi\_fd\_fdstat\_get(wasm\_exec\_env\_t exec\_env, wasi\_fd\_t fd,**

**wasi\_fdstat\_t \*fdstat\_app;**

This function:

* Has the first argument to be a pointer to the *WASMExecEnv* structure
* Second argument is a file descriptor (an integer)
* Third argument is returned by reference

This is a normal C function, but the arguments need to be marshalled in the native invoker from a buffer of inputs and a buffer for output.

### Native Pointer Invocation

Native functions expect to receive and return pointers, but WASM only works with integers and floats. Pointers are handled as *offsets* into linear memory – WAMR must convert between them. It does this via the functions *wasm\_runtime\_addr\_app\_to\_native()* and *wasm\_runtime\_addr\_native\_to\_app()*.

### Passing large number of Arguments

WAMR pre-allocates space for arguments as a buffer on the (native) stack, but in the case that the number of arguments at runtime exceeds this space then arguments are passed on a “stack” (actually another buffer). This is mirroring the calling convention of the architecture, and so the number of variables allocated is platform specific. Like most architectures, the AArch64 calling convention is to pass a number of arguments in registers and then the rest on the stack. This is, though, all made more complicated by the fact that the *exec\_env* needs to be passed as an “additional” first argument to the native function.

### Need for CHERI Changes

Significant changes were required for CHERI which necessitated a whole new assembly code part of the invoker, to marshal the C call to the actual native function, and to avoid a “mess” of *#if/#else/#endif* constructs the *wasm\_runtime\_invoke\_native()* function has its own implementation for CHERI pure-cap platforms.

The changes were driven by the following:

1. The Morello stack is, like AArch64, aligned to 64-bits but when pushing a capability pointer to the stack in pure-cap mode then it is 128-bit aligned.
   1. This caused a problem with the non-pure-cap implementation because of the need to handle pointers at the native side that are not the same as 64-bit integers
2. When converting linear memory offsets to native pointers, WAMR assume the pointer is no larger than the offset and so treats the pointers exactly as a 64-bit integer
   1. This loses provenance on CHERI and so pointers must be passed separately to integers
3. The assembly code needs to be rewritten to deal with the fact that X0..X19 are actually capability registers C0..C19

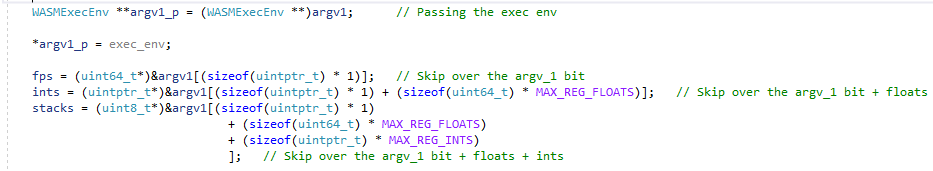
### CHERI Changes List

#### Argument Passing Structure Changes

The structure of the argument buffer that is prepared and passed to the assembly code part of the invoker is now changed. The main change is that the buffer is now a *byte* buffer to deal with the fact that some arguments may be full capability pointers, and others may be integers (before the change, all arguments were 64-bits on size). The new structure for CHERI is as follows:

| Value | Size | Byte Offset | Notes |
| --- | --- | --- | --- |
| *WASMExecEnv \** | 16 bytes (capability) | 0..15 |  |
| Floating Point Values (0..8) | 8 bytes x 8 | 16..79 | If less than 8 FPs then buffer set to 0 |
| “Integer” Values (0..7) (or non-stack Pointer Values) | 16 bytes x 7 | 80..191 | If less than 7 Ints then buffer set to 0  *7 not 8 because WASMExecEnv\* occupies 1 register!*  **Store as 128-bit value** |
| Stack Arguments (0..16) | 16 bytes x 16 | 192..448 | Could be an integer, float or capability pointer.  **Stored as 64-bit value OR 128-bit value *with appropriate alignment.*** |

This can be seen in the following code snippet, where we setup the different regions from within the overall argument buffer:



The resulting values are padded and aligned as required in order to map to the corresponding registers and stack for the calling convention, as follows:

| Value | Calling Convention Map | Notes |
| --- | --- | --- |
| WASMExecEnv \* | C0 Register (capability, 16byte) |  |
| Floating point values (0..8) | D0 -> D7 Registers (float, 8 bytes) |  |
| Integer Values (or non-stack pointers) (0..7) | C1 -> C7 Registers (full capability, 16 bytes) | If the native called function is expecting an integer, it will read the capability as an X or W register |
| Stack Buffer (0..16) | Stack, but with alignment:   * Ints and floats, 64-bit AArch64 stack alignment * Capability pointers, 128-bit alignment | For capability pointers, the 128-bit alignment is achieved by an optional 0 pad of 64-bits.  When preparing the buffer we must align up any pointer values to suit. |

The above is noted to be the Morello C calling convention for pure-cap.

#### Preparing the Arguments

WAMR passes the input argument buffer along with the function signature to determine if the input argument is a 32-bit integer, 64-bit integer, 32-bit float, 64-bit float or an integer representing an offset which needs pointer conversion (this would be either a string or an array).

The arguments are then converted as needed and stored – with necessary alignment – into either:

* The floating point / integer region, if space is available (i.e less than 8 or 7 arguments, respectively)
* The stack, if space is not available

Managing this on CHERI is complex, and therefore additional helper functions are added:

* *update\_args\_as\_int64()* : Write an integer to the *“ints”* buffer, or “stack” area if full
  1. Note: 32-bit integers are extended to 64-bit before calling the function
* *update\_args\_as\_f32()* :Write a 32-bit float to the *“fps”* buffer, or “stack” area if full
* *update\_args\_as\_f64()* :Write a 64-bit float to the *“fps”* buffer, or “stack” area if full
* *update\_args\_as\_pointer():* Write a capability pointer to the “ints” buffer, or “stack” area if full
  1. Note: Recall the “ints” actually map to full 128-bit capability registers

Each of these functions has special handling to ensure alignment is correct. This is summarised below:

| Function Handler | Write to ints/fps buffer case | Write to stack area case | Bytes used in area |
| --- | --- | --- | --- |
| *..as\_int64()* | Cast to 128-bits (0 extends) and write  (will be read as 64-bit X or 32-bit W reg in callee) | Align up the block to a 64-bit boundary and copy value.  Increase stack usage by 64-bits. | 16 |
| *..as\_f32()* | Copy to fps buffer as 32-bit float, will pad to 64-bits | Align up to 64-bits, write value as 32-bit float with pad | 8 |
| *..as\_f64()* | Copy to fps buffer | Align up to 64-bits, write value as 64-bit float | 8 |
| *..as\_pointer()* | Copy to ints buffer (128-bit entries) | Align up to **128-bits,** copy value as 128-bit pointer | 16 |

#### Invoker Assembly Marshaller

The *wasm\_runtime\_invoke\_native()* function calls the *invokeNative* function which is implemented in assembler.

**Note:** To keep the compiler happy, there are multiple external function declarations depending on the expected return type, but the assembler function does not need to change for this.

The assembly function *invokeNative* is implemented once for each architecture in assembly code, which can all be found in <https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/core/iwasm/common/arch> folder. The chosen file is matched in the *CMake* process by the *WAMR\_BUILD\_TARGET*, which in our case is AARCH64 for both hybrid and pure-cap versions.

However, for pure-cap it was decided to implement a new assembly file, *invokeNative\_aarch64\_purecap.s* as there are so many changes needed. To support this, the *CMake* file uses the *CHERI\_PURECAP* build flag to determine if it should include *invokeNative\_aarch64.s* or *invokeNative\_aarch64\_purecap.s* in the build.

The job of the assembly file is to perform the calling convention mapping from the arguments in the buffer to the arguments needed for the native calling convention.

The *invokeNative* function is called as follows (from C):

**invokeNative\_Int32(func\_ptr, argv1, stack\_bytes\_used)**

*Where:*

* *Func\_ptr* is the native function to be called (passed in to the invoker, and unchanged)
* *Argv1* is the arguments buffer in the format outlined previously
  1. Begins with *exec\_env*, then *ints/pointers*, then *fps*, then *stack area*
* *Stack\_bytes\_used* is the size of the stack area actually used

**NOTE:** *Stack\_bytes\_used* is a change specific to the CHERI pure-cap case. WAMR implemented this as the number of arguments passed on the stack, but this cannot work in the pure-cap case because arguments are of variable length with variable alignment (e.g 8 bytes with 8 byte alignment or 16 bytes with 16 byte alignment). Instead, we track the actual number of bytes used – this is the number to copy to the native stack.

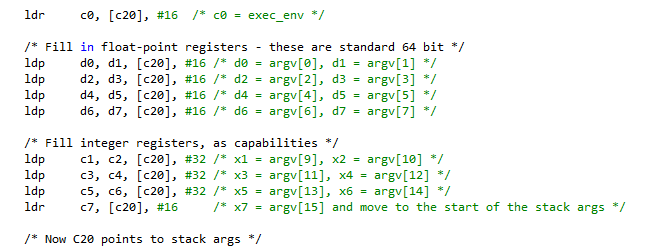
Upon entry to the assembly function, following standard C calling convention, the following are then set:

* C0 register == *func\_ptr*
* C1 register == *argv1* buffer pointer
* X2 register == size of stack area (as 64-bit integer)

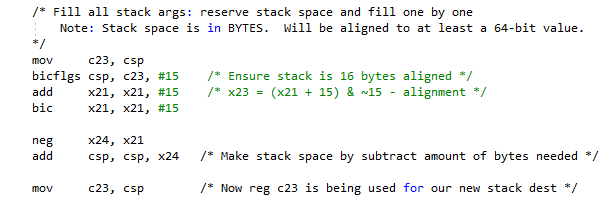
The *invokeNative()* function must now perform the following actions:

1. Save additional stack registers which will be used
2. Copy the first *argv1* argument to C0 register, as this is the *WASMExecEnv\**
3. Copy next 8 arguments to D0..D7 registers, as 64-bit values (floats)
4. Copy next 7 *argv1* arguments to C1..C7 registers, as 128-bit *potential* pointers (or ints)
5. Copy remaining *stack\_bytes\_used* bytes to the native stack
6. Call the function passed as *func\_ptr*
7. (On func return) Restore the LR (i.e C30) register for the function return back to C
8. Restore saved stack registers (ensuring C0 register is not corrupted, as this is the native function return value passed back to C code)

Key snippets from the code are shown below. Here is shown copying values to C0, D0..D7, C1..C7 – NOTE at this point C20 is set to argv1:

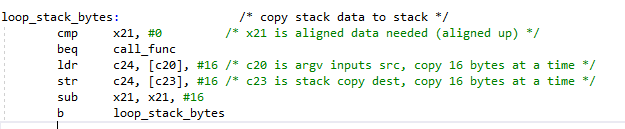


At this point the function can be called if there are no stack args (i.e *stack\_bytes\_used == 0)*. Otherwise, stack arguments need to be copied, but first the stack is 64-bit aligned to match the incoming buffer (Morello stack always at least 64-bit aligned):

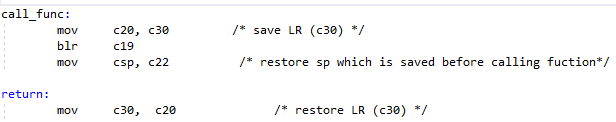


(Note that the stack grows downwards, hence the alignment is achieved by subtracted needed value).

We can then copy the stack, which is looping round copying via an intermediate capability register. The copying is done 16 bytes at a time, because *stack\_bytes\_used* was previously aligned up to a 16-byte boundary (it does not matter if we end up with unused 0 bytes taking up the stack):



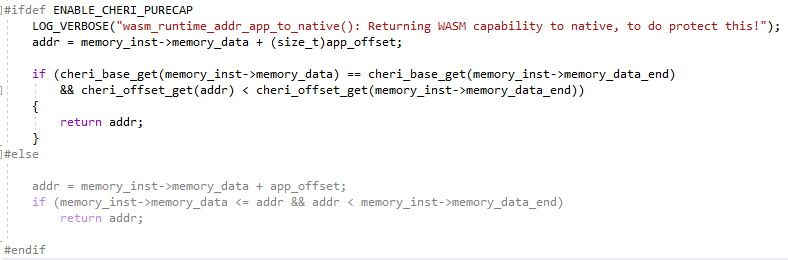
Finally, the native function is called. On return, the crucial step is taken of restoring the stack pointer to what it was when *invokeNative()* was called. This has the effect of deleting all the stack copying which occurred in the function, and ensures the stack is setup correctly for the return to the C function:



#### Converting WASM Application Memory Address to Native Pointers

A WASM linear memory address is an offset from the start of the memory. This is converted to a native address by simply using the offset as an index into the native buffer used for the linear memory.

For CHERI this works ok, because the capabilities for the start and end of the buffer is derived from the capability provided when first *malloc()*ing the buffer, but in order to check the linear memory index is in range WAMR treats the native pointers as integers. This needs a slight tweak to work on CHERI, as shown below:



The above implicitly doesn’t accidentally lose provenance, and also verifies the *memory\_data* and *memory\_end* are derived from the same allocated region capability.

## Hybrid Support

Morello Hybrid is being largely phased out by Arm, although we do continue to support it as a build target with both LLVM and GNU toolchains. It is particularly of interest when evaluating benchmarking as a means to compare the performance of purecap with a non-purecap environment in order to evaluate the impact of CHERI capabilities on performance.

Hybrid allows pure Aarch64 with the ability to explicitly force a pointer to a capability via the *\_\_capability* modifier; e.g *void \* \_\_capability my\_ptr*.

In the CHERI port we explicitly force capabilities on both hybrid and purecap for new code related to CHERI memory management. This is an extension for the platform in question which has negligible impact on any performance testing.

It was found that the LLVM toolchain is more strict about casting capability pointers to non-capability pointers on hybrid – an explicit cast is needed. To this end, macros were added to allow explicit casting between pointers and capabilities on hybrid, as below:

**#if !defined(\_\_CHERI\_PURE\_CAPABILITY\_\_)**

**#define CHERI\_CAP\_TO\_PTR(p) (((void\*)cheri\_address\_get((void \* \_\_capability)(p))))**

**#else**

**#define CHERI\_CAP\_TO\_PTR(p) (p)**

**#endif**

**#if !defined(\_\_CHERI\_PURE\_CAPABILITY\_\_)**

**#define CHERI\_PTR\_TO\_CAP(p) ((void \* \_\_capability)((void \*)(p)))**

**#else**

**#define CHERI\_PTR\_TO\_CAP(p) (p)**

**#endif**

Note that on purecap, these fallback to having no effect.

## WAMR Documentation Updates

The WAMR git repository includes a number of readme files. These have been updated to cover the changes made to support the *verifoxx-cheri-wamr* port. Specifically, and at the time of writing, the updates cover:

1. The top-level *README.md* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/README.md>) which includes the build instructions using *CMakePresets.json* and toolchain files, optionally with Visual Studio
   1. The *build\_wamr.md* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/doc/build_wamr.md>) is also updated to cover the location of the above instructions
2. Changes to *wamrc* for compiling AOT from WASM are covered in *build\_wasm\_app.md* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/doc/build_wasm_app.md>)
3. Use of *wamrc* updates are covered in the WAMR compiler *README.md* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/wamr-compiler/README.md>)

# WASM Interpreter Modes

The initial target for the WAMR port was to be able to run a simple, standalone WASM application that did not rely on any WASI C runtime support with a bespoke front-end. The work then quickly moved on to be able to run complex WASM applications using the WAMR Interpreter Mode with WAMR’s *iwasm* program front-end.

Although “Classic” interpreter mode is not the default, this was implemented first as a baseline before additional work was carried out to support the “Fast” interpreter.

This section describes the changes made and issues which were encountered as part of the process.

## Classic Interpreter

WAMR’s classic interpreter mode is built in lieu of fast interpreter when *WAMR\_BUILD\_FAST\_INTERP* is set to zero. It is not possible to disable build support for all interpreter modes.

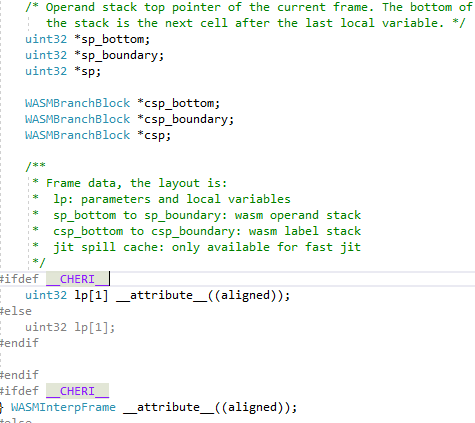
The classic interpreter parses the WASM file as it encounters it and reads opcode information. These are then individually handled by manipulating the *frame stack* (classic interpreter is a stack machine implemented almost exactly as described in the WASM specification) and, as necessary, setting up a new frame for operations that change the thread of execution e.g function calls.

The classic interpreter WASM opcode “handling loop” is cumbersome and difficult to debug, as it uses macros rather than function calls to handle each opcode – what happens is a table of operation IDs 🡪 labels is created, resolved by the preprocessor, so WAMR can goto the correct label to handle the operation. This has clearly been done to avoid having to pass numerous variables to a function, however it is noted that if the code had been implemented in an object-oriented language then these could have been member variables to avoid the use of macros. The code also uses goto instructions to skip over different opcodes and return to different handlers within the “loop”. These are embedded in macros which also makes the code hard to follow and debug. The full handler can be seen in *wasm\_interp\_call\_bytecode()* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/core/iwasm/interpreter/wasm_interp_classic.c>).

### Frame Stack Changes

In the previous section it was explained how the operand stack was set up for CHERI and how the allocation was done to ensure alignment when a new frame was needed. However to complete the changes for classic interpreter we needed to consider the use and alignment *within* the area reserved for a single frame.

WAMR stores information about the frame stack area in the *WASMInterpFrame* structure, however in the same way that we have seen before it lays out a “blob” of data at the bottom of this structure and then sets pointers within the structure to point to areas inside the blob. Shown below is an excerpt from this (modified for CHERI), and the comment describes a little about how WAMR will lay out the blob:



With reference to the WASM specification, the following regions are defined:

1. Function parameters and local variables area
   1. WAMR knows the size because the function declaration information (that was defined in the WASM) informs the parameters and the locals for a given function
   2. WAMR copies input parameters to this area when initializing the new frame
   3. If calling the function from the interpreter (e.g it is the entry point), then this is handled by *wasm\_interp\_call\_wasm()* and WAMR copies input arguments to the frame lp area as they are passed as *argv* / *argc* construct
2. Operands (values) stack
   1. WAMR knows the maximum size for the current function as it was determined by the parsing done at WASM loading time
3. Label stack
   1. WASM determined the maximum number of labels by containing functional blocks at WASM load time

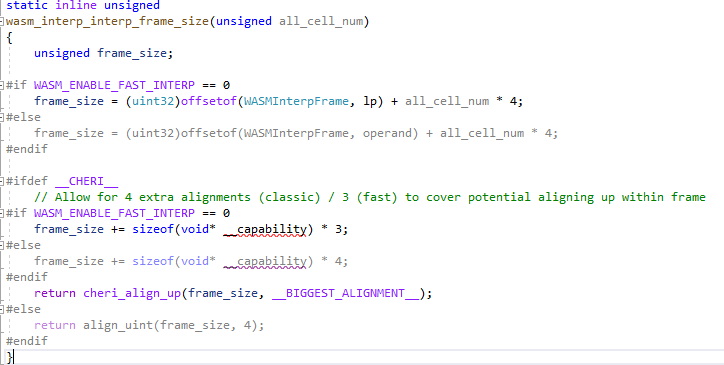
***Note:*** *WASM functions return a value by leaving it on the “stack” – so in the case of the above, that’s the Operands stack*.

#### Changes for CHERI

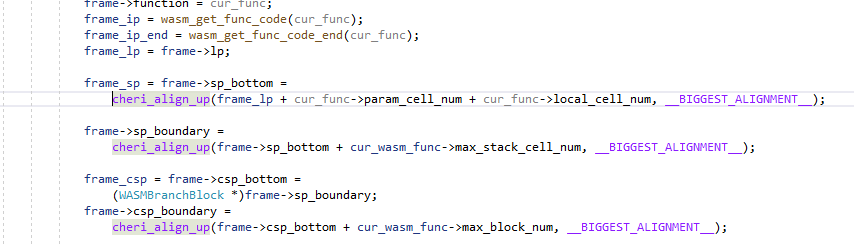
For CHERI, it is necessary to align the three parts of the frame stack up to a capability boundary. Bearing in mind that WAMR operates the stack as containing 32-bit integers, each area could be unaligned.

In order to do this, the overall size of the frame that is “allocated” from the main stack needs to be increased to allow for possible additional alignments.

The size is given by the function *wasm\_interp\_interp\_frame\_size()*. For CHERI we add additional space to cover possible alignments, as shown in the code snippet below. Note the overall size is then aligned so that we always request a whole number of capability pointer sized bytes:



It is then necessary to align up the different areas when the frame is laid out – recalling that the *WASMFrameInterp* structure contains pointers which point to the correct areas in the blob, it is just necessary to set these accordingly (note: during processing WAMR uses temporary variables which are set to this structure’s fields, and writes them back when the frame switches):



With reference to the above code, to explain this in more detail:

1. *frame->lp* is the start of the block after the *WASMInterpFrame* structure. It’s already aligned because:
   1. The allocation of the frame is aligned before we assigned the block
   2. The *WASMInterpFrame* struct field *lp* is aligned because we told the compiler to align it on CHERI (so the structure is padded up to the correct alignment)
2. *frame\_sp* is a proxy for *frame->sp\_bottom*:
   1. It starts after *lp* space, which is space for the function parameters + locals
   2. Note the variables are *uint32\** and so the natural alignment would only be 4-byte aligned
   3. It’s then aligned up to the next capability pointer boundary
3. *frame->sp\_boundary* is the end of the operand stack region, which is then the start of the label stack region, it is also aligned up so the label stack starts on a boundary
4. *frame->csp\_boundary* is aligned up because the end of the frame will be aligned up
   1. It’s used to indicate the end of available space, and whilst technically this does not need to be aligned it is then a true reflection of the actual space available
   2. It’s used to check for stack overruns

The resulting layout is shown graphically below. Note the “alignment” reserve areas can be 0..15 bytes in size and so extra space was allocated to cover them:

A yellow rectangular object with blue and purple lines

Description automatically generated

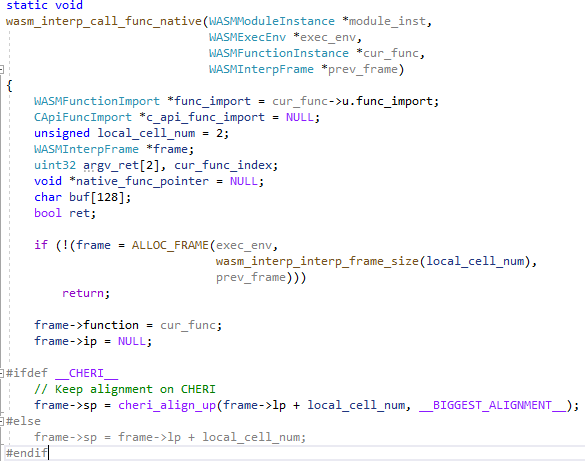
### WASM calling Native Function Changes

When WASM calls a native function WAMR behaves as if it is calling another WASM function with respect to the fact that a stack frame will be laid out as shown above. Given that the return argument from the function is passed back on the stack, in the operand stack (*frame->sp*) then WAMR needs to prepare this and in the case of CHERI that means preparing it to be aligned as explained previously.

There is of course in native code no need for the label stack, so it is just the parameters stack and the operands stack (for function return). To set these up correctly, on CHERI:

1. The frame is allocated, aligned, from the main stack area
   1. This is already covered because all calls to *ALLOC\_FRAME()* will handle this and calls to *wasm\_interp\_interp\_frame\_size()* will increase size as needed
2. The *frame->lp* must be aligned
   1. This is implicitly covered as it is the start of the area straight after the structure
3. The *frame->sp* must be aligned
   1. This must explicitly be done (see below)

Code is added to *wasm\_interp\_call\_func\_native()* to achieve necessary operands stack alignment:



### Developing Native Code called from WASM

It has been described how native functions can be called from WASM. In the case of e.g WASI built-ins, these are part of the WAMR the executable (assuming *--static* was used to build against static WAMR libraries).

User native code, though, is loaded at runtime. This is handled by building a dynamic library (on Linux platforms a *shared object* (.so) file), specifying the pathname to WAMR on the command line argument and then this is loaded via *dlopen().* This works fine for Linux pure-cap, but there is a problem if the library itself refers to other shared objects. This is because the Debian install for Morello contains A64 libraries not C64 libraries that WAMR pure-cap would need.

As it happens, the library almost certainly will need to refer to other shared objects because WAMR’s *iwasm* library is not available statically and this includes code to manipulate the executable environment and derive WAMR module data from it.

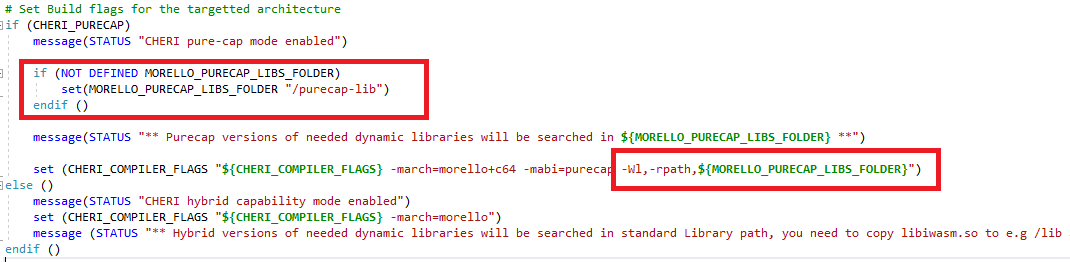
This problem is solved on pure-cap by specifying the library’s *RPATH* to be a predefined location. As described in Section 3.5.2.2 this is done by specifying the build flag *MORELLO\_PURECAP\_LIBS\_FOLDER* to be the path on the Morello board which contains the *libiwasm.so* needed by the library. Other libraries (such as C standard library) can either be linked statically (preferred) or also added to this folder. Any dependent libraries (such as C math library) which are not available statically would also be transferred to this folder. By default this folder is defined on the Morello box as */purecap\_lib*, if the *CMake* build flag is not defined.

***NOTE:*** *Any runtime libraries needed to be transferred must be pure-cap ones – these would be available from the toolchain libs folder for the cross-compiling toolchain, but it must be ensured the correct variants are chosen!*

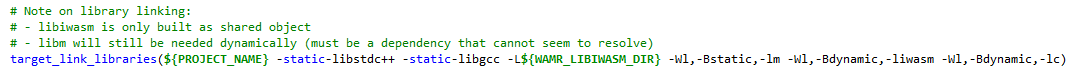
For this project we have included a simple C++ native test library which includes a *CMakeLists.txt* to build it. This can be found in <https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/tests/wamr-linux-cheri-purecap-tests/native_libs>.

This can be built from the top-level by setting the *CMake* build flag *WAMR\_BUILD\_NATIVE\_TEST\_LIB=1*. This will force a build of the library along with WAMR.

From the *CMakeLists.txt* for the native test lib we can see the setting of *RPATH*, and the use of the default option:



Any libraries which are available statically are linked as such. Note the link path to the built WAMR libraries (*-L* argument) for symbols, but the *run* path is set by the linker *-rpath* previously defined:



The actual task of writing the native library code is then no different to as described in WAMR documentation.

## WAMR Fast Interpreter

The WAMR interpreter runs in “fast” mode by default, or if the build flag *WAMR\_BUILD\_FAST\_INTERP=1*. The fast interpreter reuses much of the same code as the classic interpreter, and the opcode handling loop is very similar although an entire new file is created for it (see <https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/core/iwasm/interpreter/wasm_interp_fast.c>).

The rationale behind the fast interpreter vs classic interpreter is that the fast flavour will execute quicker than classic, but the trade-off is higher memory footprint.

*ByteCodeAlliance* claim 2 x faster performance than classic mode, for a 2 x memory footprint vs classic mode.

**NOTE:** The rationale is important, as any significant changes for CHERI should favour increased memory footprint over increased execution time, all else being equal.

### Operation of Fast Interpreter

Fast Interpreter is somewhat different to the classic interpreter, and deviates quite a lot from the approach described in the WASM specification.

Note that the majority of the changes apply to the pre-processing stage.

This makes the fast interpreter operation complicated to understand; during the project this was determined largely by reverse-engineering / studying the code although one of the few WAMR documentation resources was found which explained the fast interpreter in a little detail (<https://www.intel.cn/content/www/cn/zh/developer/articles/technical/webassembly-interpreter-design-wasm-micro-runtime.html>). The operation of Fast Interpreter is therefore described below as an aid to understanding how it differs from classic interpreter and how it did (and didn’t) need to be modified for CHERI Morello pure-cap.

#### Use of Opcode Offsets instead of Opcodes

In WAMR code, instruction handlers are basically a table of C language goto labels (i.e an address in execution space - and note for some compilers there will be an additional step to resolve the label address). WAMR constructs what is analogous to a function table, except for WAMR it is goto labels in one massive function as mentioned previously.

When building the instruction frame for a function, rather than store the opcode the *address offset* of the label to handle the code is stored. This is not an absolute address but offset from the first instruction’s handler label address. Therefore to convert it to an absolute address in WAMR’s implementation, one just needs to add a fixed address to it.

As the amount of code is less than 64Kbytes the address offset can be stored as a 16-bit number. This is convenient because the instruction frame buffer is already read as 16-bit values.

This is clearly much more optimal than having to resolve the address of the bit of software from an opcode for every single opcode. Given that the base address is known at compile time, this offset to absolute address conversion can likely be done in one clock cycle on most architectures (like AArch64) or even an additional zero clock cycles extra depending on how the instruction buffer is being used.

***Note:*** *This change has no real impact on CHERI, i.e no changes are required to this part of the codebase.*

#### Decoding Compressed Integers at Load Time

Integers in WASM are compressed using a variable-length-quantity (VLQ) mechanism whereby the number of bytes they take up is dependent on the value. Specifically, [*LEB128*](https://en.wikipedia.org/wiki/LEB128) is used for webassembly.

When doing the bytecode pre-processing, Fast Interpreter WAMR mode will uncompress numbers into a full integer. This potentially uses a lot more memory – every (maximum of) 64-bit value will now take exactly 8 bytes whereas before compression they could have been as little as 1 byte for small numbers – but it speeds up processing, which is the important point.

Further, WAMR will build a buffer of constants used by the function. This means that if the same *const* value is used more than once then it can be referenced by an index in the WASM instruction frame code and in the data frame the value only needs to appear once.

As the instruction knows what size constant is needed (32-bits or 64-bits), WAMR can store them packed and then store an appropriate offset, for example as shown below (note: in reality WAMR uses a 16-bit value for the offset):

| WASM Opcode (as text) | Offset | *Consts* Byte Byffer (Little Endian) |
| --- | --- | --- |
| i32.const 0x12345678 | 0 | 78 56 34 12 |
| i64.const 0xdead | 4 | 78 56 34 12 00 00 00 00 00 00 ad de |
| i32.const 0x89 | 12 | 78 56 34 12 00 00 00 00 00 00 ad de 00 00 00 00 89 |

***Note:*** *This change has no real impact on CHERI, i.e no changes are required to this part of the codebase.*

#### Reducing Number of Opcodes

During the pre-processing stage, WAMR Fast Interpreter mode is able to eliminate opcodes that are responsible for doing nothing other than pushing or popping to the data value stack frame.

In WASM a number of instructions are just responsible for loading or removing values from the stack. For example consider the following:

* *i32.const 1* -> put value 1 on the stack
* *local.get 0* -> put value of first local on the stack
* *local.set 0* -> set value of first local from the stack

If these instructions can be eliminated, then there is a lot less processing to do and so the WAMR VM will run faster.

We can illustrate the way this is done with an example. Consider this WASM to add two int32s together:

**i32.const 0x12345678**

**i32.const 0x89**

**i32.add**

Given the *const* buffer defined above in the example table, we could instead eliminate the first two instructions by simply providing add with an index into the buffer locations that store the values we need to add:

**i32.add [0] [12]**

WAMR calls these indexes “slots” and as well as constants there is also a buffer space allocated in the data frame for locals. For function parameters (and the return value, which will be 0, 4 or 8 bytes depending on if there is one, and if it is an i32 or i64) there is also a buffer (although it is treated like the locals buffer).

Note that as WASM is stack-based, WAMR needs to keep track of the “slot” indexes for the locals buffer. For example consider the following:

**i32.const 1**

**i32.const 2**

**i32.const 3**

**i32.add**

**drop**

**drop**

WAMR knows the slot index for each of the consts 1, 2 and 3 but during the parsing it needs to know that i32.add applies to the consts 3 and 2 so when it completes the (fast interpreter modified) i32.add instruction it needs to use the appropriate slot offsets in that instruction. To do this, WAMR maintains what it calls a “slot stack” during the pre-processing stage (but this is not needed during the execution stage).

##### Results

Instructions like *i32.add* generate a result which would go onto the frame’s operands (value) stack in the classic interpreter. But that doesn’t exist in Fast mode, so instead we need an additional buffer to deal with the result values. WAMR calls this a “dynamic buffer”.

These dynamic values are short-lived, because they will either end up being written back to locals / globals / function return, or they will be consumed by other instructions.

As above, WAMR maintains the slots for the dynamic buffer on the slot stack so that it knows where to store the next value and what slot index to use in all the instructions.

##### Preserving Values – an additional Buffer

Consider the following WASM instructions:

1. **i32.const 2 ;; 2 on stack**
2. **local.set 0 ;; Set first value = 2 – stack empty**
3. **local.get 0 ;; Get value of first local - 2 on stack**
4. **i32.const 4 ;; 4, 2 on stack**
5. **local.set 0 ;; Set value of first local = 4 – 2 on stack**
6. **i32.const 3 ;; 3, 2 on stack**
7. **i32.add ;; Add 3 + 2 = 5 – 5 on stack**

In the instruction #5, above, the local is updated with the value of 4. But the original value (2) is still on the stack.

When storing slot IDs there is a problem because WAMR cannot use the slot ID for the local, because its value has changed. But WAMR will eliminate all the instructions #1 to #7 because they are just pushing and popping to the stack.

The solution to this problem is what WAMR calls a “preserve” buffer. This is an additional space which is used to store local values that have been updated but their original value needs to be preserved for use in subsequent operations.

WAMR will therefore insert a special “pseudo-opcode” to *copy* the local slot to the “preserve buffer” before updating it in instruction #5. It will then use this preserve slot in instruction #7.

##### Conditional Branches

As WAMR Fast Interpreter is handling values via a slot index instead of having a value frame stack, there can become an issue for conditional branches because different paths through the code can end up with different values in the slots (i.e they will have different meaning) and so WAMR cannot effectively generate the correct slot IDs for instructions occurring after the end of the block.

To deal with this, WAMR will use the internal *copy* pseudo-opcode to copy needed values into the correct slots from one path to ensure at the end of the block the layout is the same no matter which condition was taken.

Note that this in WASM as there is effectively no “else if”, only if/else/end, then this is not as complicated as might have been.

##### Understanding the Frame Data Structure

The frame data for Fast Interpreter mode is quite different from Classic mode. From the above, we can see that there are the following types of buffer needed:

* Constants
* Function locals (comprising func parameters / return values and locals)
* Dynamic Area
* Preserve Values Buffer

WAMR uses an *offset index* for the slot to access one of these variables. The entire buffer space is contiguous, but the index starts after the *consts* area. Therefore:

* A negative offset is used to access a const variable
* Locals ranges from 0… (locals\_size – 1)
* Dynamic ranges from locals\_size… (locals\_size + dynamics\_size – 1)
* Preserve ranges from preserve\_size… (locals\_size + dynamics\_size + preserve\_size -1)

This is illustrated below, this block would be allocated as the frame payload (i.e after the *WASMInterpFrame* structure):



##### WASM Pre-process Multiple Parsing

With reference to the diagram shown above, the dynamic and preserve space cannot be known until the entire function is parsed. In Fast Interpreter mode, WAMR therefore needs to parse the entire WASM function/module twice at WASM load time:

* On the first parse, sizes are collected including the entire instruction code size that will be required for the post-processed instruction buffer
* On the second parse the actual processing is done (i.e filling in the correct instruction values and addresses)

#### Elimination of Label Stack

WAMR’s mechanism for processing instructions on Fast Interpreter mode will be as follows:

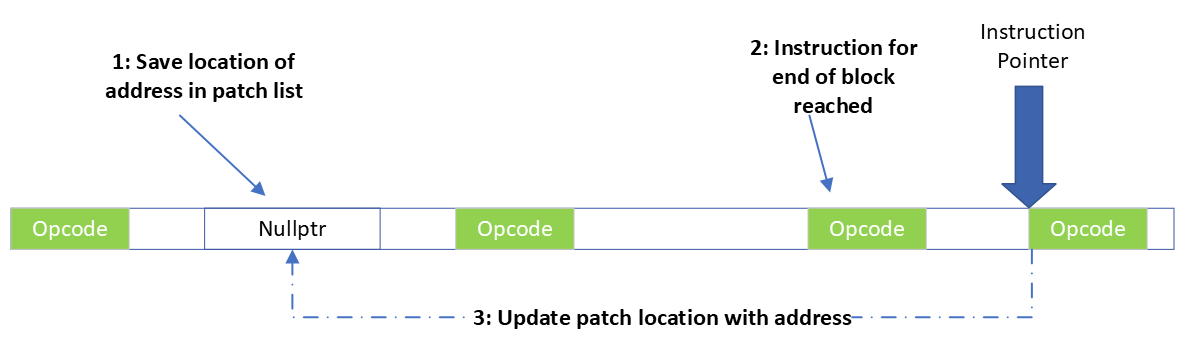
* read the opcode handler address offset and make an absolute address
* run the code at the handler, which will read further instruction operands
* Process the instruction and then advance to the next opcode address in the instruction frame buffer

However, there is a problem with this. Block and loop branches (which may be conditional) along with if / else conditional execution will cause a discontinuity in the flow. It is therefore necessary for WAMR to be able to record in the instruction buffer an *address* of the next location to set the instruction pointer to whenever there is a branch. (The absolute address is the fastest method of doing this, albeit with the cost of memory usage increasing vs using an offset).

By inserting addresses into the instruction flow, WAMR Fast Interpreter ensures that at WASM execution time the flow is fast. Calculation of these addresses is done at pre-processing time, where an effective stack of branch information is built so that WAMR can correctly traverse this to find the correct branch address in the case of nested blocks.

In the case of a backward branch (a loop) there is no problem with this approach because that part of the WASM was already parsed, but in the case of a forward branch (block, if/else) then WAMR does not yet know the address because the instruction buffer will be variable length depending on the instruction and its parameters. So now there is another problem. To deal with *that*, WAMR uses the “patch list”.

The patch list works by inserting a *nullptr* address and putting the location of this in the patch list. Then, when the actual address is discovered, WAMR is able to “apply the patch” and replace the *nullptr* with the correct address. An example is illustrated below:



This solves all the problems, and the WAMR speed-up approach for Fast Interpreter will now work.

### CHERI Modifications

#### Stack Frame Alignments

As is done for Classic Interpreter mode, the stack frame is allocated from the main stack store via *ALLOC\_FRAME()* and this is guaranteed to be aligned.

Recalling the overall frame structure includes *WASMInterpFrame* structure at the start, this is then the frame breakdown for WAMR Fast Interpreter mode:



The *WASMInterpFrame* structure is aligned and therefore the frame data payload will be aligned, hence for WAMR Fast Interpreter the const space starts on a capability aligned boundary.

It is not possible, or necessary, to align the start of the local space – the local and const space are treated as a single entity. This is because WAMR uses negative offsets to refer to slots in the const space (as can be seen, const space is to the left of the zero line hence offset is negative).

It is also not necessary to align the dynamic and preserve slot areas, because these contain only 32-bit or 64-bit values, and slot offsets are used to access them.

Therefore, the only changes needed for CHERI were:

* apply the same alignment when allocating the frame that was done for classic interpreter
* calculate the frame size appropriately (not having to allocate additional space for the alignments)

#### Label Address and Patch Handling

As explained in the discussion of WAMR Fast Interpreter, WAMR will insert addresses into the instruction buffer (perhaps after first inserting null and then patching).

The instruction buffer is comprised of 16-bit parameters, however for CHERI Morello pure-cap these capability pointers need to be aligned to a 128-bit boundary otherwise they cannot be used. This causes a significant problem when building the instruction buffer in the pre-processing stage as the pointer needs to be *stored* correctly and then at the WASM execution stage it needs to be *loaded* using the reverse mechanism of the store.

As noted previously, at execution time WAMR may need to skip over an address value, for example in the case where a conditional branch is not taken. Therefore it must be possible to know the space taken up by the pointer at runtime.

There are a number of possible solutions for this problem; these are explained here and the rationale behind the one chosen is explained.

***Possible Solution 1: Treat the Pointer as a Word Buffer***

The naïve approach would be to attempt to ignore the need to align by treating the pointer in memory as an array of 16-bit words, storing these without alignment and performing the reverse at load time. This is indeed how WAMR stores and loads 32-bit and 64-bit values.

However, knowledge of CHERI tells us this **simply cannot work** for CHERI pointers because if it could then this would be a massive security risk. Any CHERI capability pointer includes a tag which is held in secured memory and not a part of the 128-bit capability pointer itself. As soon as the pointer is treated as a simple 128-bit number, the tag is invalidated and reading the pointer back later becomes useless.

Hence, this solution #1 is not a workable solution.

***Possible Solution 2: Store the Address part only***

We can note that the target for the branch will itself be somewhere in the instruction buffer, and therefore we could store the 64-bit address part of the capability pointer and use it at load time to generate a full capability by combining it with the *frame\_ip* pointer.

This solution is workable, but there are a number of issues which make it non-ideal:

1. It does not favour the ethos of “speed over size” by resolving full addresses and values when possible instead of needing to process them at execution time (albeit a likely 2 extra cycles on Morello only)
2. It is restrictive when considering possible future CHERI sandboxing of WAMR. This approach would mean that *any* address in the instruction buffer could be accessed without being an out-of-bounds address; in future we may wish to tightly restrict functions to be able to access instructions within that function. If we can use a full address then the bounds can be tightly controlled.

***Possible Solution 3: Store full Pointer, Aligned, Bespoke Storage Size***

To be able to store the full pointer it is mandatory to align it, which means aligning up the storage location to the next 128-bit boundary. Fortunately this is a single assembly instruction – on Morello this is the *ALIGNU* instruction operating on a capability register – so it is a fast operation.

This effectively inserts a pad into the instruction buffer stream, and then the 128-bit pointer. The length of the pad will vary and depends on what else is present in the buffer before this.

For this to work, WAMR needs to know the length of the pad for each occurrence. This though is where this solution becomes problematic and complicated, because:

In the first pre-processing parse, WAMR calculates needed sizes only. Therefore WAMR will need to actually perform the alignment on the running size to work out how much pad will be needed each time a pointer has to be stored. Further, WAMR must do enough processing to accurately work out *exactly* where in the instruction stream the pointer will be otherwise the alignment is unknown and hence the pad and subsequent space used is unknown.

In WASM execution, whenever an address needs to be skipped then WAMR will need to calculate the alignment to know exactly how many bytes to skip. In some cases, WAMR skips multiple addresses and parts of an instruction’s operands but it cannot rely on a known size for each operand since for pointers it will be variable.

As well as increasing complexity, there will be an increase in processing required because not only would an *ALIGNU* be needed for every occasion that the pointer needs to be skipped, but also additional processing is needed to parse instruction operands that could have been skipped. This approach would save memory, because a pad is only inserted when needed, but this goes against the ethos of favouring speed over size.

This solution #3 is therefore slightly flawed.

An enhancement to this solution would be to insert the pad size into the instruction stream before the pad and the pointer. The pad size only needs to be 4 bits, so in practice a single byte, and this would mean that this would reduce the need to calculate the alignment each time. This approach would look like this, in the example 1 byte is used to record the pad size and a further 9 bytes of pad is needed to align the capability pointer:

The enhancement, though, is limiting because it still makes it necessary to read additional data (the pad length) which can be different for each pointer that is stored. It is not really any better than having to calculate the alignment each time, and in fact would be more machine instructions on Morello.

Therefore this solution is rejected if a better one can be found.

***Solution 4: Store Full Pointer, Aligned, Fixed Storage Size***

This solution is the one chosen. It follows on from solution #3, but instead of variable length storage for each address the storage size is always fixed at 32 bytes.

32 bytes is the size of the pointer + the maximum size of pad + 1 to align to a 2-byte boundary as mandated by WAMR instruction buffer

In this approach, 32 bytes is always allocated into the instruction stream whenever an address needs to be stored and therefore WAMR is able to assign a fixed size as it can for any other type. This means that:

1. In the pre-processing initial parse, WAMR does not need to process the full instruction operands to calculate a variable length alignment and can instead just add the fixed size of 32-byte to cover the storage space for an address
2. In the WASM execution phase, WAMR can safely skip over any addresses along with other operands since the size is known up front and is fixed (and this size is a compile-time constant)

This approach means that performance is maximised as the only time when an alignment is needed is when storing and loading the pointer itself, and for all other times a fixed size can be used.

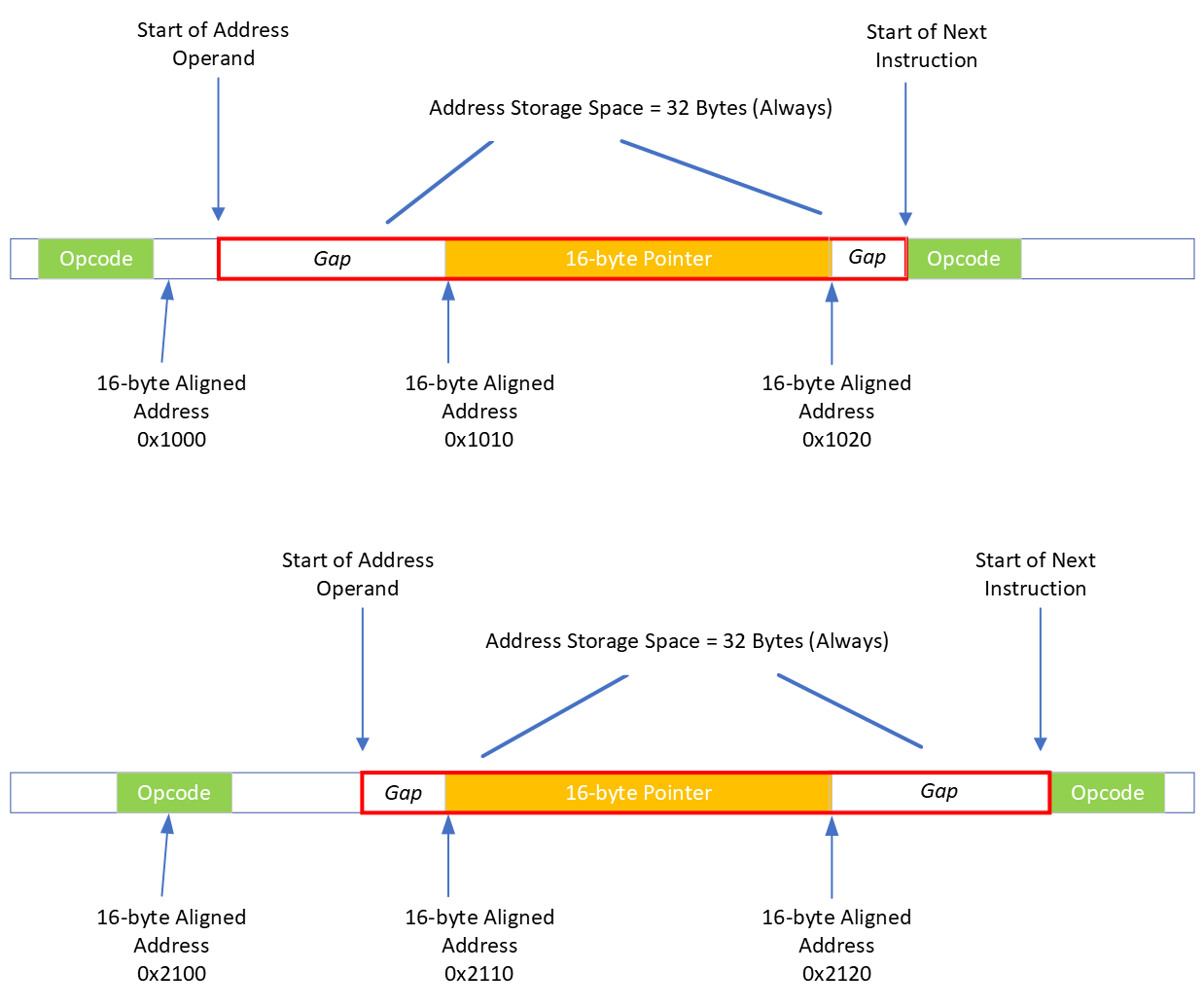
##### Solution Explanation

The storage space is fixed at 32 bytes in the instruction stream.

At some offset within this 32-byte space will be the 16-byte full capability pointer. The offset within the storage space depends on whatever “gap” (pad) needs to be left in order to achieve alignment. This is calculated by aligning up the instruction pointer to the next 16 byte boundary (which is guaranteed to be within the first 16 bytes of the address storage space).

This offset is calculated when the pointer is stored, before writing it, and when the pointer is loaded, before reading it.

This is illustrated in two examples shown below, each one with a different alignment pad being needed:



In the first example, a large pad happens to be needed to achieve 128-bit alignment and so the pointer appears closer to the end of the 32-byte operand space.

In the second example the layout of previous instructions is such that a small pad is needed to achieve the alignment and so the pointer appears near the beginning of the 32-byte operand space.

In both cases, the instruction stream always sees a 32-byte region which is reserved for storing addresses and it is only necessary to perform the alignment when actually writing or reading the pointer.

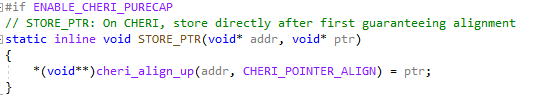
*Downside*

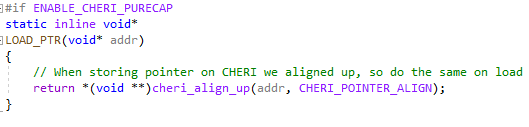
The downside of the solution is that additional memory is needed compared to a variable length storage. Given the 2-byte instruction buffer pointer granularity, then a pad value of 0, 2, 4, 6, 8, 10,12, 14 or 16 bytes is required. If we presume that there is an equal chance of any size pad being needed then on average 9 bytes of pad are needed which means for a fixed storage size of 32 bytes there are 7 bytes wasted for every saved address compared with the variable length solution #3.

However, this must be weighed up against the performance degradation of solution #3 and given the WAMR fast interpreter prefers performance over code size then this is deemed an acceptable memory increase for not only a faster performance but also a simpler codebase.

*Implementation*

The macros *STORE\_PTR()* and *LOAD\_PTR()* are modified in the CHERI case to align-up the storage address before writing / reading the full capability address, as follows:

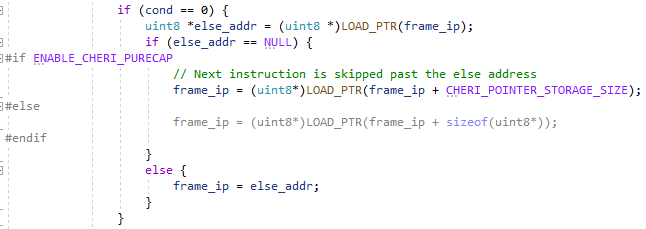




The actual address passed to the functions is always the address read from the instruction buffer of the storage space, i.e before alignment. So for example when saving the location to be patched following pre-processing the code remains unchanged (see *apply\_label\_patch()* in <https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/core/iwasm/interpreter/wasm_loader.c>):

**STORE\_PTR(node->code\_compiled, ctx->p\_code\_compiled);**

When skipping over the address range, the size is fixed. For example, if there is an “if” with no “else” then the “else” branch address is null and so instruction execution continues after its space:



The macros *CHERI\_POINTER\_ALIGN* and *CHERI\_POINTER\_STORAGE\_SIZE* are used for the alignment value and the storage size to make it easier to support possible future CHERI platforms:

**#define CHERI\_POINTER\_ALIGN \_\_BIGGEST\_ALIGNMENT\_\_**

**#define CHERI\_POINTER\_STORAGE\_SIZE \**

**((sizeof(void \*) + CHERI\_POINTER\_ALIGN + 1) & ~0x1)**

On CHERI Morello pure-cap *CHERI\_POINTER\_ALIGN*==16 and *CHERI\_POINTER\_STORAGE\_SIZE*==32 as explained above.

### Code Size Issue

The Fast Interpreter label address mechanism requires that the delta to any address can fit inside an *int16* value. Therefore, this mandates that the total size of the fast interpreter source code module is at maximum *sizeof(uint16\_t)* which is equal to 16,384bytes.

Unfortunately CHERI bloats the codebase somewhat since every address cannot be resolved via just an integer but instead a capability which needs to derive from a base address with address, bounds and permissions modification as needed. The consequence of this is that, with all WAMR features enabled, the codebase was found to (slightly) exceed this 16kByte limit when building debug builds.

The solution to this is to force “optimise for size” (*-Os*) in the compiler *for this module only*,even when the rest of the codebase is optimised for performance. These changes are handled in the *CMakeLists.txt* file.

Fortunately this only affects debug builds – benchmarking uses release versions, and all code is built with maximum performance optimisation for benchmarking.

## Interpreter mode Testing & Validation

WASM test cases were developed to test both classic & fast interpreter modes. These were developed by hand (as WASM Text format, converted to WASM via the *wat2wasm* tool) and as C code and then compiled to WASM via *WASI-SDK*’s Clang version.

Initially all testing was done by hand using the following steps:

1. Build WAMR for either classic or interpreter mode
2. Transfer *iwasm* to Morello board over SSH
   1. For native testing, also transfer the native test *so, libiwasm.so* and other needed dynamic libraries and copy to */purecap-lib* folder
3. Transfer WASM test files to Morello board
4. Execute *iwasm* from a shell on the Morello board with correct arguments; observer *stdout*, log points and reported return value

Later (as described in this document under *Testing*), and automated harness was developed.

#### Test Cases

Initial test cases were very simple, but as soon as any WASI libC is pulled in then even a simple C code program will generate very lengthy and complex WASM due to the inclusion of the C standard library being compiled to WASM. The C *printf()*, for example, involves numerous layers and function calls before the native *wasi\_fd\_write()* function is called.

Test cases were added to the *verifoxx-cheri-wamr* git repository and can be found in <https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/tests/wamr-linux-cheri-purecap-tests>. Sub-folders *./src* and *./wat\_dumps* contain the original C code or WAT (WebAssembly Text) and a textual representation of the WASM as necessary.

Tests of note are as follows:

*Initial Simple Tests*

The simplest test does not involve calling any native functions, however this cannot do anything useful because there is no output and no return value. This test was though used to run through the process and check that WAMR was functional. Two simple tests scripts were performed (shown as WAT):

**(module**

**(type (;0;) (func (param i32 i32) (result i32)))**

**(func (;0;) (type 0) (param i32 i32) (result i32)**

**i32.const 99**

**return)**

**(export "main" (func 0)))**

This very simple test has no *exit()* and so does not actually return a value from the WASM module. It can though be used to check the interpreter mode can run cleanly, call a function from interpreter to bytecode and manipulate the stack with no errors.

**(module**

**(type (;0;) (func (result i32)))**

**(type (;1;) (func (param i32 i32) (result i32)))**

**(func $\_\_original\_main (type 0) (result i32)**

**(local i32 i32 i32 i32 i32)**

**global.get $\_\_stack\_pointer**

**local.set 0**

**i32.const 16**

**local.set 1**

**local.get 0**

**local.get 1**

**i32.sub**

**local.set 2**

**i32.const 0**

**local.set 3**

**local.get 2**

**local.get 3**

**i32.store offset=12**

**i32.const 99**

**local.set 4**

**local.get 4**

**return)**

**(func $main (type 1) (param i32 i32) (result i32)**

**(local i32)**

**call $\_\_original\_main**

**local.set 2**

**local.get 2**

**return)**

**(table (;0;) 1 1 funcref)**

**(memory (;0;) 2)**

**(global $\_\_stack\_pointer (mut i32) (i32.const 66560))**

**(export "memory" (memory 0))**

**(export "main" (func $main)))**

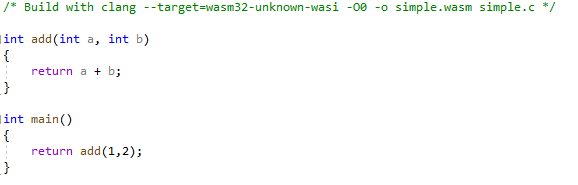
This second test is more complex, having some parts inherited from the way *WASI-SDK* generates WASM. It includes a memory export, a table and has two functions with one calling the other (it also has an entry point which can be detected directly by WAMR). It also has operations which manipulate the operand stack and the linear memory.

Crucially, though, this example does not call any native code and therefore was a good test which could be used before the WASM-to-native calling mechanism was fully functional.

*Next Step Tests*

The next stage of testing involved calling a native function and writing output, but without including any WASI code. The test file ***hello\_no\_libc.wasm*** was developed to call the WASI API *fd\_write()* directly. This more complex code also included blocks and therefore (once known to work on Classic Interpreter) was a good candidate for testing the CHERI changes on Fast Interpreter. It is compiled with *WASI-SDK* clang without C Standard library or runtime inclusions.

Tests which would call *wasi\_proc\_exit()* were also useful – the simplest is ***simple.c***, which – if working correctly – should cause WAMR to exit with an exit code of 3:



*Complex Testing*

A complicated WASM file is actually one of the most simplest pieces of C code, as shown in ***hello\_wamr.c***:



This is built with the full C standard library and the *printf()* is complex before calling the low-level *\_fd\_write()* WASI libC function.

*Bespoke Testing*

Several files were developed specifically for the different control flow testing of the patch table changes for CHERI that applies to Fast Interpreter mode. These are ***control\_instrs.wat***, which includes many different conditional instructions and blocks, and ***call\_indirect.wat*** which tests the complex *call\_indirect* WASM instruction.

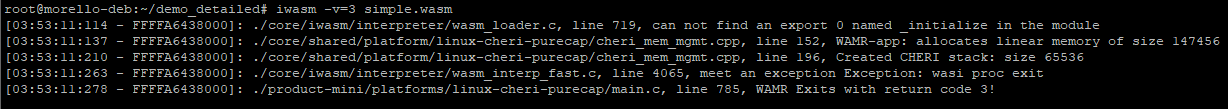
Specific test cases were developed for checking WASM -> native handling:

* ***wamr\_native\_calls.c*** : Calls native functions from WASM
  + Requires the test native library in *..\native\_libs\* to be built, to supply needed native functions
* ***wamr\_natives\_and\_call\_wasm.c*** : This will call native functions which themselves will then call other functions in WASM
  + Also requires the test native library
  + The test native library functions that get called specifically need WASM exports defined in the WASM generated from this test case file

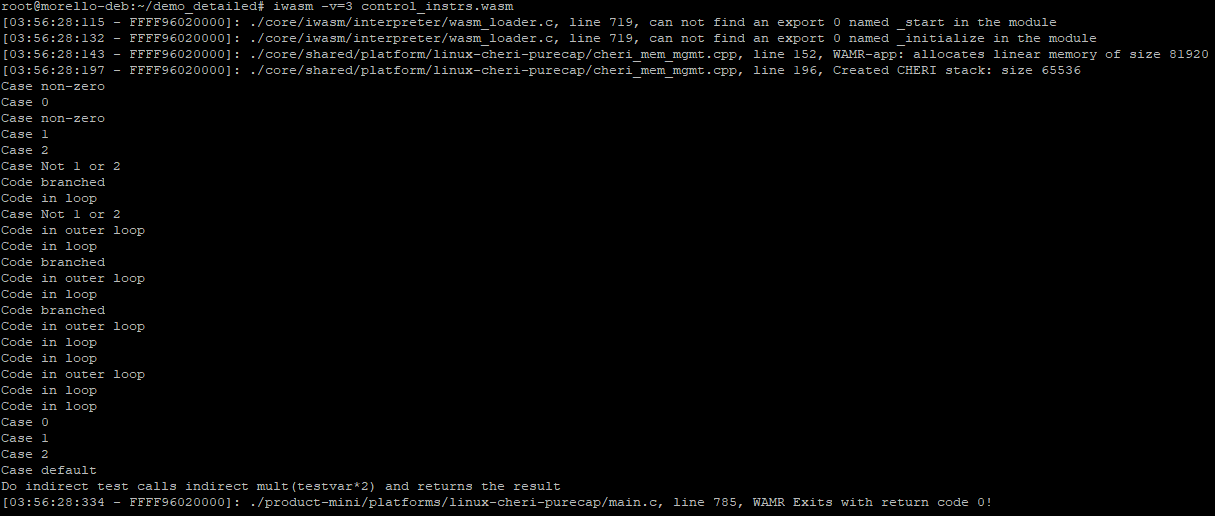
Testing the native function calling is cumbersome because the majority of arguments will be passed to the native function in registers, but passing additional arguments on the stack should also be tested. The *wamr\_natives\_and\_call\_wasm* test does validate a large number of arguments which will make use of the stack, aligned as needed to pass a capability pointer on pure-cap.

#### Example Outputs

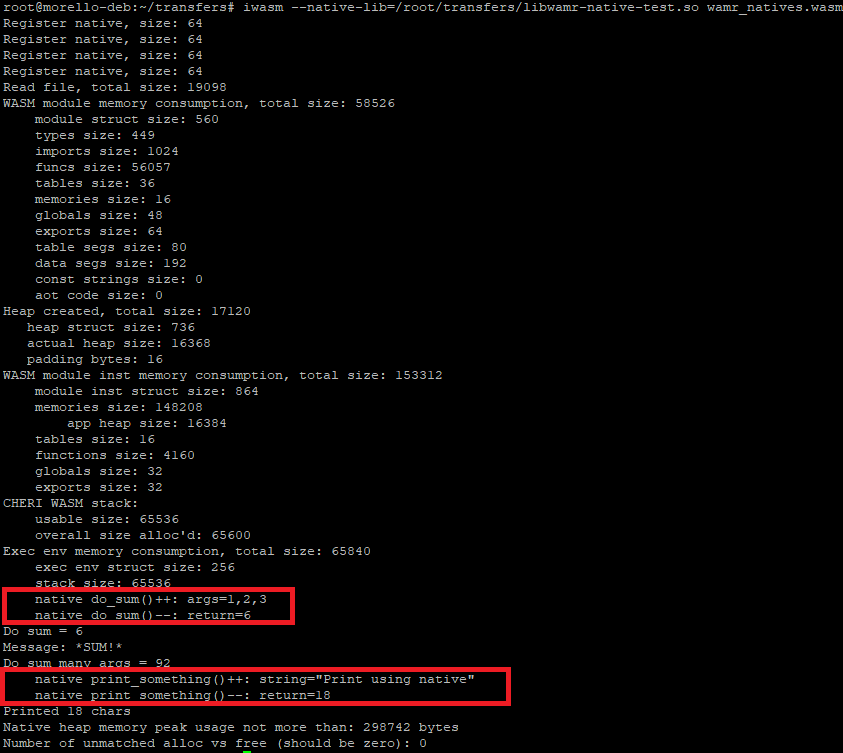
Output from running *simple.wasm* with debug logging is shown (Fast Interpreter). Note the return value of 3, which is as expected, also the report of an exception (although this is just WAMR mechanism to exit the processing loop, not a real exception):



Output of running *control\_instrs.wat,* in which different conditional instructions are tested (Fast Interpreter):



Output from running *wamr\_native\_calls.wasm* additionally with memory tracing on (Classic Interpreter). The native code *printf()*s are highlighted with a red box. The native library to load is specified as a command line argument:



# WASI Support

WAMR provides a *WASI LibC* implementation which enables WASI-SDK compiled native SW to call into standard library functions.

For networking, WAMR also provides its own *WASI-sockets* library file which needs to be compiled to WASM in order to support Berkeley sockets.

Porting WAMR’s WASI support was largely a test & validation exercise as there were no additional WAMR changes required. However as part of the porting it was decided to confirm that secure TLS/SSL connections were functional within a WASM application running in WAMR on Morello pure-cap. To support this it was necessary to build a WASM application which included a secure-sockets library and which would work with WAMR’s version of *WASI-sockets*; *openSSL* was chosen for this purpose.

As well as verifying that TLS/SSL connections were functional with WAMR this also served to generate large, complex WASM files which proved to be useful test vectors going forward.

## WAMR and WASI Components

The relationship between WASI and WAMR is shown in the diagram below:

A diagram of a software system

Description automatically generated

### WASI Components

***WASI***formally specifies a set of APIs by which system-independent WASM can call system-dependent native services.

To support building WASM from C code, the ***WASI-libc***is a port of the C standard library that will end up calling into WASI API functions. *WASI-libc* is actually in two “halves”, the top half is a standard MUSL-libc implementation and the bottom half implements low level system calls and directs them to WASI API functions.

To compile C code to WASM, ***LLVM / Clang*** can be used. Newer versions of clang support compiling WASM, i.e they have a backend to compile LLVM IR to WASM.

The ***WASI-SDK***does not supply anything new itself, but brings together a suitable LLVM/Clang and WASI-libc into a single development environment so that C programs which use C library functions can be built to WASM applications.

### Support within WAMR

WAMR provides a native WASI-libc library which implements the WASI API as native functions. When WASI-libc support is built in to WAMR it includes all the API functions as native functions (in additional to any user defined native functions from a user library).

WAMR registers the WASI API name for the WASI functions and maps these to the implementation function (actual native library implementation) along with the signature. For example, this is some WASM code (as text) specifying the import of “fd\_write” (this was generated by building a C code file):

**(import "wasi\_snapshot\_preview1" "fd\_write"**

**(func $\_\_imported\_wasi\_snapshot\_preview1\_fd\_write (type 5))**

**)**

Here is the table entry in WAMR:

**{“fd\_write”, wasi\_fd\_write, “(i\*i\*)i”, NULL}**

It maps *fd\_write* to the function *wasi\_fd\_write()*. The signature happens to indicate the function takes an integer, a pointer, an integer and a pointer and returns an integer – it’s WAMR’s own internal format and is not relevant to this dicussion.

The WAMR code to implement the native function is multi-tiered. It comprises:

1. A wrapper level to actually implement the WASI API functions
2. Code which seems to be copy/pasted from the *wasmtime* application, to implement the methods with low level code
   1. At this point, the *capability handling* is done (see below)
3. A POSIX layer (for Linux based platforms) which actually calls in to Linux system calls

### WASI Capabilities

WASI introduces the concept of “capabilities”.

***IMPORTANT NOTE:*** These are nothing at all to do with CHERI capabilities, they just share the same name because they are ultimately focussed around the concept of *capability based security*.

WASI capabilities relate to WASM sandboxing. They are used to restrict access to system resources (such as files, network accesses) and only allow access if permission has been granted for the resource.

WAMR provides API functions which are available to native code calling WASM, and that includes its own *iwasm* application. WAMR provides the following to enable capabilities:

1. Access to directories
   1. Allows r/w access to specified directories and sub-directories only, including all files in the directories
2. Access to IP addresses (provided in [CIDR](https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing) format)
   1. Only access to addresses on the sub-net specified (i.e which meets the mask)
   2. Applies for any network access including domain resolution, socket connect, etc.
3. Access to domain names (provided with wildcard mask)
   1. Applies to domain resolution

Additionally, WAMR enables environment variables to be passed to WASI via command-line.

*Understanding CIDR Address Masking (Quick Summary)*

WAMR requires IP address masks to be provided in CIDR format which is:

<base\_address>/<bits>

Where *bits* gives the netmask of the *base\_address* which must match to define the subnet. An IPv4 address comprises four parts, within range 0.0.0.0 to 255.255.255.255 and therefore considering each part as a byte there are then 4 bytes or 32-bits within an entire IP address.

As an example, if a router provides local IP addresses within the range 192.168.0.0 to 192.168.0.255 then if one was permitting WAMR to access anywhere on the subnet the one would provide it with *--addr-pool=192.168.0.0/24*. To allow access to any IP address on the internet, *--addr-pool=0.0.0.0/0* would be used.

*WAMR Domain Name Wildcards*

WAMR allows domain name resolution of any name matching a wildcard match, for example *\*.microsoft.com* allows any domain name ending *microsoft.com* to be resolved. To allow *any* address to be resolved, one can simply use *–addr-resolve=\**.

### Impact of CHERI

There was no impact on WASI functionality for CHERI – all was found to work as expected.

## OpenSSL Support in WASM

WAMR and WASI “out of the box” provided support for *Berkeley Sockets* however for the project we wanted to verify that SSL and TLS secure connections would work, such that a WASM application could service SSL/TLS client/server connections with the use of certificates as necessary.

In order to do this, an SSL/TLS library would be needed and this would need to be compiled to WASM. The popular *openSSL* library was chosen for this. This is a complex, large library providing numerous security services but as it is implemented in C code then the *WASI-SDK* should be able to build it to WASM.

A complexity of this is that WASI sockets has not yet been formally specified. WAMR provides its own socket extensions library (*wamr\_sockets\_ext.[hc]*) and therefore it would be necessary to build against this file in lieu of any standard C library sockets code. This restriction caused further problems, as some of the openSSL library source code made use of socket library functions which are not available in WAMR’s socket extensions. This necessitated *patching* the openSSL code as part of the build process.

### WASI-WAMR OpenSSL Version

A github repository was created containing:

1. A git submodule comprising a (linked) recent version of openSSLv3 release. Specifically, this tag:

<https://github.com/openssl/openssl/tree/674b61ebd982d6a6564ac1f90d8cde22371564bc>

1. A set of Linux *patchfiles* and a patch script which will use the Linux patch command in order to modify the openSSL files as necessary to work with WAMR
2. A build script which will build openSSL using the *WASI-SDK* clang
   1. This build script requires a version of WAMR source code so that it can include the WAMR *WASI sockets extension* code
   2. The build also includes a modified version of the *netdb.h* Linux header file, to force use of WASI sockets extension
   3. The build script configures a minimal option configuration for openSSL, to only build what is needed to provide secure sockets support
   4. The build script also builds WAMR’s *wasi\_socket\_ext.c* to an object file as it must be included when linking any user WASM with openSSL
3. A pre-built version of openSSL libraries and the *wasi\_sockets\_ext.o* which are then able to be linked with a C application to generate a full WASM module
4. An example TLS client & server, with build files to link against the libraries and objects from above

Unfortunately, at the time of writing the github repository is private to a user and has not been moved into the Verifoxx organisation due to administrative issues with the Verifoxx organization. However the pre-built openSSL libraries and wasi\_sockets\_ext.o are attached below as a zip file, and the TLS client / server example is included in the zip file:



### Using the openSSL WASM Libraries

**NOTE:** *Libraries are built for compiling on Linux host e.g x86\_64. Requires WASI-SDK (LLVM/Clang and wasi-libc).*

The openSSL libraries generated from the above git project (either built or prebuilt), result in the following structure. Assuming you place them in ~/wasi-openssl:

~/wasi-openssl/libcrypto.a  
 ~/wasi-openssl/libssl.a  
 ~/wasi-openssl/wasi\_socket\_ext.o  
 ~/wasi-openssl/include/wasi\_socket\_ext.h  
 ~/wasi-openssl/include/openssl/\*.h

To generate a WASM application which supports SSL / TLS connections, three steps are needed. Assuming you placed the above in your home folder, then:

1. Write a C program which calls openSSL methods as needed to perform required actions
   1. *#include <openssl/some\_file.h>* and *#include <wasi\_socket\_ext.h>*
2. Compile the C program (no link) to WASM (assumes your C program is *~/main.c*):

**clang --target=wasm32-unknown-wasi -I~/wasi-openssl/include -o ~/main.o -c ~/main.c**

1. Now link along with openSSL libraries and wasi\_socket\_ext.o (assume you want to generate *~/main.wasm*):

**clang --target=wasm32-unknown-wasi -L~/wasi-openssl -o ~/main.wasm ~/main.o ~/wasi-openssl/wasi\_socket\_ext.o -lssl -lcrypto -lwasi-emulated-getpid -lwasi-emulated-signal -lwasi-emulated-process-clocks -Wl,--export=malloc -Wl,--export=free**

1. The resulting WASM will be a *very large file* as it includes so much openSSL code. It is all a self-contained module, and your can run it with WAMR
   1. Be sure to pass WAMR appropriate arguments for *–addr-pool, --allow-resolve* and (if you need any folders for eg X.509 certificates) then *–dir* as required

*Explanation of Linking Options*

It is noted that the link step that generates the WASM includes additional libraries other than the openSSL libraries, namely:

* libwasi-emulated-getpid.a
* libwasi-eulated-signal.a
* libwasi-emulated-process-clocks.a

These libraries are needed to provide the WASM code with emulated Linux system functionality that the openSSL library needs. The above libraries are provided with *WASI-SDK*.

There are all two options to the linker, it is being told to export two functions:

* malloc
* free

This is needed due to the use of the *WAMR Sandboxed Application Heap*. WASM compiled C cannot use the native heap – exporting these functions allows WAMR to implement them using its internal *mem\_alloc* which will ensure the WASM application heap is used instead of the native one.

### Example TLS Client and Server

An example TLS client and server is provided in the attached zip file from above, in the *tls\_client\_server\_example/* folder. A simple HTTPS web client is also provided there.

This example was developed as part of the project work in order to demonstrate secure socket connections from WASM with WAMR, and also as a set of test vectors. The resulting WASM files are huge, and complex, and therefore serve as excellent general test vectors for WAMR.

Please refer to the README.md file in that folder for more instructions, this section is just a summary.

#### Prebuilt WASM

After unzipping the file, in the *tls\_client\_server\_example/prebuilt* folder there can be found the three WASM modules already built.

#### Building the Examples

The *tls\_client\_server\_example/* folder contains a full *CMake* project build environment – it includes a *CMakeLists.txt* and a *CMakePresets.json* and a cmake toolchain file.

The project can be built either standalone or using Visual Studio – refer to the *README.md* for more information.

The project does have a requirement that the *verifoxx-openssl-wasi-wamr* project was already built – i.e that the openSSL libraries, include folder and *wasi\_sockets\_ext.o* are available. The location of this output is specified via the *CMake* variable *WASI\_WAMR\_OPENSSL\_DIR*. In the *CMakePresets.json* this is set to *~/test-openssl/verifoxx-openssl-wasi-wamr/out* but you can modify it to the *./prebuilt/* folder extracted from the zip file, for example.

The project build will generate the three target WASM files by compiling each of the various source code files to object, and then linking each against the openSSL libraries, WASI libraries and *wasi\_sockets\_ext.o* as provided.

#### Running the Examples

The examples are pure WASM and therefore can be run with WAMR on any platform. The only stipulation is that the version of *wasi\_sockets\_ext* must be the same as provided by the WAMR being used; however, in practice this has not changed recently, if ever, in the WAMR codebase and therefore there is unlikely to be an issue.

Each example takes arguments from the *iwasm* command line, which are supplied after the WASM file name. Be sure to provide the necessary *addr-pool, allow-resolve* and *dir* arguments to WAMR as needed for WASI capability permissions.

The examples are as follows:

1. https\_client.wasm : A simple HTTPS client, will resolve a domain name, make an HTTPS (TLS secured) connection to a web server and download the home page
   1. Will obtain the server’s certificate.
   2. Can optionally provide a path to root certs for certificate validation
2. tls\_client.wasm : Simple TLS client, designed to be used with the simple TLS server
   1. Will make a TLS/TCP connection to given IP:port and exchange data with the server
   2. Supply key and certs files from the command line
3. tls\_server.wasm : Simple TLS server, designed to be used with above client
   1. Will listen for TLS/TCP connection on <defaulhost> and given port, accept connection from the client and exchange data
   2. Supply key and certs files from the command line

*Note on X.509 Certificates*

SSL / TSL is secured via public / private cryptography, and a PKI infrastructure (root of trust). In the *https\_client.wasm* example, to authenticate the server then the client can be provided with a folder containing root certificates (e.g Digicert, Verisign or Amazon roots) which can be used to (hopefully) authenticate the server.

For the TLS client and server example, self-signed certificates are provided which can be used. For the client, it is necessary to provide:

* Server’s certificate : (mandatory – so client can authenticate the server)
* Client’s certificate : (optional – if *client authentication* is being used from the server side)
* Client’s private key : (optional – if *client authentication* is being used from server side, so client can sign the challenge)

For the server, it is necessary to provide:

* Server’s certificate : (mandatory – so client can authenticate against the version it uses)
* Server’s private key : (mandatory – so server can sign the challenge from the client)
* Client certificate : (optional – only if *client authentication* is required)

As is the case on the internet, server authentication by the client is always performed. Client authentication by the server is only performed if the server is provided with the client certificate from the command line.

***Note:*** *If using the TLS client and server from a shell, then two terminals will be needed as two instances of WAMR are needed at the same time. If running on Morello (in hybrid or pure-cap) then the easiest thing is to open two SSH terminals in e.g PuTTy.*

#### Test References

The problem with a test vector that is tested against another test vector is they may both be faulty in the same way – the test environment ideally provides a reference. This is the case with testing the *tls\_client\_example.wasm* against the *tls\_server\_example.wasm*.

To this end, a Python test reference client and server are also provided in the example. These can be found in the *./tls\_client\_server\_example/testref/* folder.

The python test references require an installation of python3. No additional packages are required.

The test reference python files take similar arguments to the WASM versions. The idea is that one can run:

* Python client + WASM server
* WASM client + Python server

To test each of them. However one can also run Python client + Python server, if desired.

The Python references are run as follows:

**python tls\_test\_client.py**

**python tls\_test\_server.py**

Note:

* *“python”* may be “*python3”* on some machines
* Run with *-h* option to view command line help on required arguments

See the *README.md* in the ./*tls\_client\_server\_example/* folder for more information.

### CHERI Morello SSL / TLS Issues

No bugs were found – the WASM works well on both pure-cap and hybrid-cap. The three examples mentioned in the previous sub-section are excellent test vectors which have been used in subsequent development and debugging of other parts of the project.

The only minor issues are the error reporting when there is an attempted access outside WAMR’s WASM sandbox. If the openSSL libraries are not built with an export of *malloc()* or *free()* then WAMR’s sandboxed heap will not be used and an attempt to access the native heap is made.

This is caught by WAMR and is prevented – and WAMR will warn with a suitable error message.

However, on CHERI pure-cap the attempt to access outside the sandbox is caught by the *hardware security* of CHERI capabilities. This is of course intended, and is a fine demonstration of how CHERI security features are preventing unauthorized access.

Unfortunately, though, this presents the user with a Segmentation Violation (which is expected) instead of an error message indicating what they did wrong at the build stage. It may be nicer if WAMR could pre-detect the attempted violation and handle it better.

It transpired that this issue was actually caused by a bug within WAMR, whereby the violation was due to a capability going out of bounds in the calculation of an enlargement value inside the implementation of a WASM *memory.grow* operation. The bug was corrected and WAMR behaviour was then found to be the same as on non-CHERI platforms.

## WASI Output Examples

Below shows the output from running the TLS client example connecting to the TLS server example (both running on Morello, *iwasm* built for pure-cap running in classic interpreter mode):

A computer screen with white text

Description automatically generated

A computer screen with text on it

Description automatically generated

The server is launched listening on TCP port 1234, and it has been given permission to access any IP address and read files from the current folder. Certificates are loaded by the *openSSL* library and the socket listens for a connection.

The client is launched to connect to TCP port 1234 on localhost. It has permissions to resolve any host and access any IP address, and it also can read files from the current directory. The client resolves *localhost* to the loopback IP address and connects its socket.

The server reports the client has connected, and then establishes the TLS connection. We can see the client and server have agreed on the block cipher *AES-256 GCM mode* and a *SHA384* hash. The server sends its certificate, this is matched against the one loaded into the client and therefore the client authenticates the server. The client dumps the server’s certificate X.509 issuer and subject fields – we can see it is a test certificate and a self-signer.

As we supplied the server with a client certificate then client authentication is required. The server validates the client’s certificate, as it matches the one it loaded for the client. The server display’s the X.509 issuer and subject fields of the client certificate, as expected it is also a test certificate and self-signed.

The SSL connection is now established and the server blocks receiving data from the client. The client sends “*GET /”* as if it were an HTTP client, and the client then waits on receiving data from the server. The server replies with “*Hello from the Server!”* and both display the respective received data.

After sending its reply the server gracefully closes the (secure) socket, as does the client after receiving a reply from the server. The connection is then closed and both parties exit their respective WASM applications.

## WASI Test Suite

The WASI Test Suite (<https://github.com/WebAssembly/wasi-testsuite>) provides a set of test cases for testing WASI and a test execution system, which is written in Python.

Although WASI Test Suite does not seem to be a formal part of WASI it explicitly supports WAMR (and also *wasmtime*) and in fact WAMR includes WASI Test Suite support as part of its formal test folder.

Therefore during the project work we concluded that we should provide the means to, and actively use, WASI Test Suite as part of the verification for the CHERI Morello pure-cap port. Not only would this add a substantial number of test cases to validate the port along with an automated framework to run them, but also this would help lead to better acceptance of the port by the WAMR and WASI community.

### WASI Test Suite Operation

***Note:*** *Refer to the WASI Test Suite Git repository for full details.*

The WASI Test Suite comprises a pure python test runner and a number of test cases. Test cases are provided as WASM – and source code is also provided - which have been compiled from:

* C code (with *WASI-libc*) (11 tests)
* Rust (12 tests)
* AssemblyScript (12 tests)

Giving a total of 35 separate test cases.

The test runner runs from the command line – as it is a python script this can execute on any platform where Python is installed. The main runner python file is passed one or more *test suite folders,* which contain the test vectors, and a *test adapter* which is responsible for actually executing the webassembly runtime (which in our case is WAMR, i.e *iwasm*). A typical execution of the test suite would be as follows:

**python3 ./test-runner/wasi\_test\_runner.py -t ./tests/c/testsuite -r ./adapters/wasm-micro-runtime.py**

* The above will run the supplied C tests (*tests/c/testsuite*) using the supplied adapter for WAMR (*wasm-micro-runtime.py*)

Perhaps surprisingly, WASI Test Suite does not make use of popular Python test frameworks such as *unittest* or *pytest* (although it does use the *Mock* capability of *unittest*). It does though require the *colorama* python package in order to format the results output (which is written to the *stdout*).

#### Test Suite Folders

To formulate a test case, WASI Test Suite requires:

* The input vector, i.e a WASM file
* Additional data needed for the test, e.g files or folders which will be used by the WASM file
  + In the case of AssemblyScripts, this includes TypeScript files
* Config / arguments to be passed to either *wamr* (for example *–dir*) or the WASM file itself

WASI Test Suite supports the handling of additional data by the fact that the test runs in situ, so all data (e.g test folder structures) are available to the test case. The test case may modify the test data (e.g generate an output file) but the WASI test suite runner takes care of restoring the environment.

To supply arguments and configuration data for the test, a JSON file is created which must be named the same as the WASM file (e.g *foo.json* for *foo.wasm*). The keys of this JSON file are optional (each can be omitted or empty) but they can be used to supply the following:

* *env*: Construct an environment from *key=value* pairs
* *dirs*: Add each folder listed as a permissions-granted folder to WAMR (i.e the *–dir* argument)
* *args*: Arguments to supply to the WASM application itself (i.e not WAMR)
* *exit\_code*: What the WAMR exit code should be – default is 0
* *stderr*: If supplied, stderr output from WAMR should match that provided
* *stdout:* If supplied, stdout output from WAMR should match that provided

The WASI Test Suite Test Executor parses the entire folder and detects any WASM files supplied as test inputs, and constructs a test case for each within the test suite. Additionally, a *manifest.json* should be provided in the folder which gives the test suite a name.

#### Test Adapters

An *adapter* is a Python file which is responsible for actually executing the WASM VM (i.e WAMR in our case) with all the correct arguments.

WASI Test Suite launches a new shell to execute the adapter, actually running it as if it had been run from the command line as a python script. It passes all of the configuration from the test case JSON as command line arguments, along with the name of the WASM file to be executed.

The test adapter would then launch a new process to actually execute the WAMR program. To do this, the adapter needs to know the full path to the WAMR program. This can be supplied by the user by setting an environment variable *TEST\_RUNTIME\_EXE* which falls back to just *“iwasm”* in the WAMR case.

#### Example

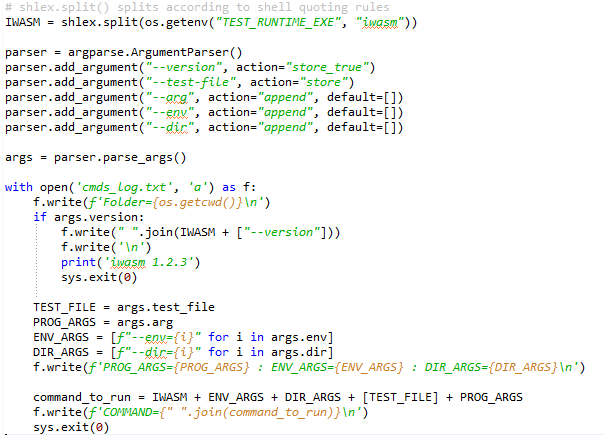
To help understand how WASI Test Suite uses the test adapter, as part of the project work we developed an adapter which would not actually execute WAMR but which would simply dump the command that would be executed to a file.

A file was chosen because:

1. Each test case runs the adapter in a completely isolated environment, as a new program
2. Any output to *stdout* would affect the test result, as WASI Test Suite’s runner monitors output and return value from the adapter program run

These restrictions do make it more complex to exchange runtime information with the adapter, so in this case the output is prepended to a file so we can see what each test case is doing. However it does make it possible to test an adapter by running it directly from the command line.

The simple adapter is shown below (python imports omitted for clarity). Note that its look and feel are very similar to the WAMR case, from which it was derived:



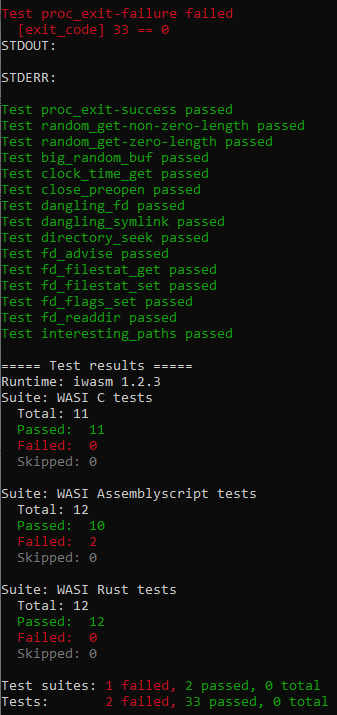
We note a few points from this:

1. Arguments are passed on the command-line, as stated, and the return value is handled via a program exit code because the adapter is run as an executable in a new shell
2. WASI Test Suite will set the working directory to the test suite, where the WASM and other test files (e.g test folders) are provided
3. It is possible for WASI Test Suite to run the adapter with a *–version* argument which is to retrieve the version of *iwasm* being run

The above adapter can be run with WASI Test Suite to test all of the supplied test cases, as follows:

**python3 ./test-runner/wasi\_test\_runner.py -t ./tests/c/testsuite ./tests/assemblyscript/testsuite ./tests/rust/testsuite -r ./adapters/simulate\_run.py**

A snippet of the output is shown below:

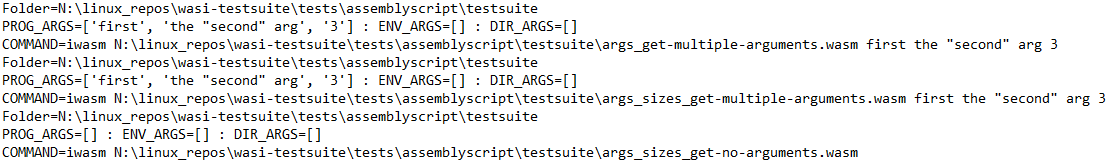


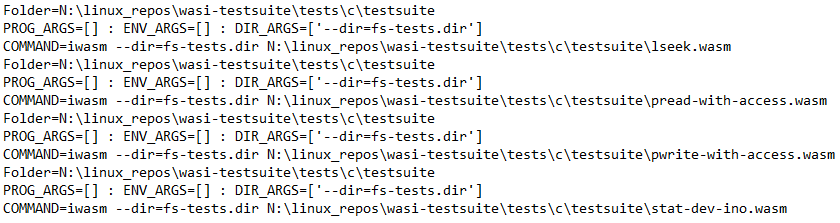
As can be seen from above, each test case result is shown as the test is executed (green for pass, red for fail) and at the end each test suite is summarised individually and then with total passes and failures. The version of the executable is also shown (faked in our case to *1.2.3*).

When a test fails, WASI Test Suite additionally dumps *stdout* and *stderr* which were generated by executing the adapter.

In our case we have 2 of 35 test cases failed. The first case (not shown above) is because a particular *stdout* is needed but in our faked adapter we do not generate any output. The second case was (shown at top of the image) because a test was told “*exit\_code: 33”* but our adapter exits with 0.

The adapter logged information to a file – below are two snippets of these:





From these we can see that:

* The working directory is where the test suite files are located, i.e the WASM and any test data
* All folders are relative to the working directory, but the WASM file is always specified with a full path
* Program arguments can include quotes and whitespace, and should be copied verbatim

This information will prove useful in adapting the test setup for CHERI.

### Modifications to support CHERI Morello

WASI Test Suite is designed to run locally – the working folder for the WAMR program is the actual provided *testsuite* folder, and WAMR is a program located somewhere on the local machine.

However in the case of Morello pure-cap (or hybrid-cap) then WAMR is running on the Morello machine. The options then for WASI Test Suite are either to make it run on the Morello board (which would require an install of python and necessary packages on Morello), or to execute the actual command remotely via SSH.

The latter approach was chosen because this was felt to be easier and also more versatile as it makes it possible to run the tests on Morello and also on the local machine (for verification on existing platforms) from the same WASI Test Suite.

It was also important to not need to modify WASI Test Suite itself to achieve the changes. Therefore a new adapter was created which supports a remote connection to the Morello board.

As WASI Test Suite relies on having the working directory to be test suite folder (which contains the WASM file and all other test data), this folder is mirrored onto the Morello board for the duration of the test. WASI Test Suite runs each test case in isolation (i.e a new shell for each time the adapter is executed), therefore the mirror is set up and torn down each time a new test case runs. This makes testing a little slow, but speed is not typically a factor when running large batches of automated tests (consider for example the concept of the “nightly run” of a regression test set on CI systems).

The python package *fabric* was used to develop the test adapter. This is designed to run build and test jobs on remote machines; it uses the *paramiko* package to access the remote machine via SSH for file transfer and remote process execution.

This necessitated additional configuration information that was not test specific but for the whole run. It is necessary to pass the remote machine IP address, the login information and where on the remote machine to create the file mirror. Again, this is complicated by the fact that (unless we want to modify WASI Test Suite’s fundamental operation) we cannot pass these from the command line to the test adapter file.

The solution was to use a user-supplied environment variable which would contain the name of a JSON file. This JSON is then opened by the test adapter each time it runs, and from this it can load the relevant configuration information needed to access the remote machine.

The resulting flow is shown below:

A diagram of a company

Description automatically generated

* The test adapter is launched, it then transfers the test suite folder via SSH to Morello
* WAMR is executed as a remote process on Morello
* Output streams and the process exit code are returned on WAMR completion
* The test suite folder mirror is then copied back to the local host

This is all contained in the adapter, and as far as WASI Test Suite is concerned it appears that the test case has run locally. Note that the local machine as shown above is a Windows 10 PC as this is what was used, but it could have run under WSL2 Linux or on a Linux PC. The only constraint is that it must be possible to make a non-interactive SSH connection from the host to the Morello board.

Creating the new WASI Test Suite adapter itself leads to making use of the WASI Test Suite to perform our own testing that had previously been done manually, and also creating a further adapter to execute AOT mode with WAMR. Therefore the details of the new adapters for CHERI are covered in their own section, Section 8.

# WAMR AOT Porting

In WAMR, AOT *(Ahead-of-Time)* is WASM which is compiled to object code that is specific for the target where WAMR will run. Although this eradicates the portability of WASM, it means that if the target is known *a priori* then WAMR can execute extremely fast (albeit at an increased file size compared to the source WASM file).

AOT is generated by the WAMR Compiler (*wamrc*). This uses LLVM libraries, via the LLVM C++ API, to generate LLVM IR by parsing WASM and then generate target-specific object code from the IR. The generated AOT file is then an object file format (e.g *ELF* on Linux platforms) along with some WAMR specific header information.

The WAMR runtime will detect an AOT file, load sections and then parse it to resolve all link-time dependences before executing the object code via the native execution functionality already present in WAMR.

The challenge for CHERI Morello porting was that *wamrc* is really designed to run on the same target platform as WAMR, whereas for WAMR we avoid trying to port *wamrc* to run on Morello. Instead, the AOT file is compiled on Linux x86\_64 (the PC host) and cross-compilation used to generate Morello object code which can then be transferred and run on the Morello platform.

## AOT Mode Operation

The full AOT flow is shown below:

A diagram of a computer program

Description automatically generated

### AOT Compiler

The AOT compiler, *wamrc*, is responsible for generating an AOT file from a WASM input. Parts of the AOT file are similar to an ELF file, in that code sections (compiled functions) are machine object code on the target in question and then data sections are included which pass in all the initialized data needed by the WASM (that would be stored in linear memory in WASM). Additionally, *symbol tables* are included in the AOT file which reference the initialized data and any imports (external references) that are needed.

The AOT file also includes “WASM sections” i.e non-ELF parts that need to be copied from the WASM file for use at WAMR runtime. Examples include the initialised WASM memory data, tables and globals.

Clearly the object code and the symbol table are target specific, so *wamrc* needs to be able to convert WASM for any target. This is done by means of an *Intermediate Representation (IR)*, which works in the same way it does for any compiler:

1. The programming language is compiled to IR; the developer of the programming language would implement the mechanism to generate IR
2. IR is then optimised for speed and / or code size
3. IR is then converted to object code for the target in question; target developers would implement the target backends

The AOT compiler uses the LLVM toolchain, and therefore the parser generates LLVM from WASM.The *wamrc* program utilizes the internal LLVM API in order to manipulate the LLVM toolchain.

Although the LLVM internal API is implemented in C++, there is a C wrapper around it which makes use of incomplete references to the LLVM C++ objects that get passed back to the C code (a standard approach to an *extern C* wrapper around C++).

The majority of *wamrc* is written in C code, and therefore makes use of the LLVM API C Wrapper, although there are a couple of files which are written in C++.

The AOT compiler is complex; it involves several phases. These are shown at a high level, below:

A diagram of a machine

Description automatically generated

First, a *Target Machine* is determined from arguments passed to the *wamrc* program. This uses the LLVM backend to retrieve the *Data Layout* for the target.

The WASM interpreter then generates LLVM IR from the WASM file – this needs to respect the target machine – and also internal structures for e.g initialized memory and tables which are not part of the LLVM output.

The IR is then *validated and optimised* to ensure it is error-free and efficient, and then the *Binary (ELF)* model is generated by the LLVM backend libraries.

Finally, the AOT can be produced which incorporates binary ELF sections as well as WASM structures.

Each of these components is examined in more detail, below, so that we can then later discuss where there were issues which ad to be addressed for CHERI support.

#### Building and Running the Compiler

The AOT compiler is built as a *CMake* project – there is a *CMakeLists.txt* and *main.c* found in the *wamr\_compiler/* folder (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/wamr-compiler>).

Like WAMR, the AOT compiler requires a number of *CMake* build flags to be set such as *WAMR\_BUILD\_PLATFORM* and *WAMR\_BUILD\_TARGET*. These can be added to a *CMakePresets.json* configuration, which was done for the project so it could build under Visual Studio as described in Section 3.5.

The AOT compiler has a dependency on LLVM. Even if an external compiler is used to compile IR to target object code, LLVM libraries are still needed.

By default the *CMakeLists.txt* assumes that an LLVM installation built from source code will be found in *../core/deps/llvm*. A python script is provided which will fetch LLVM from a git repository and then built it. If instead it is desired to use a custom LLVM then the flag *WAMR\_BUILD\_WITH\_CUSTOM\_LLVM=1* can be set. However *wamrc* build still assumes that this LLVM will be on your path.

When built, *wamrc* will include source code from WAMR (for the WASM interpreter and common functionality), the *main.c* from the *wamr\_compiler/* folder and also code from *core/iwasm/compilation* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/core/iwasm/compilation>) in order to emit IR, object code and an AOT file for a given WASM file.

The built target is *wamrc*. This has numerous options, but the main ones are to specify a target to build for (by default this is the host where *wamrc* is currently running). Instead of taking a target-triple and a feature list, WAMR build the target from several provided options:

* *target* (e.g x86\_64, aarch64)
* *target-abi* (e.g gnu, eabi)
* *cpu*
* *--cpu-features* (comma-separated list with feature enable/disable e.g +feature1,+feature2,-feature3)

#### Wamrc Operation - Setting up the Target

The first step in the AOT compiler is to resolve and set up the backend LLVM target. LLVM from the command line works with the *target-triple* which has the following format:

**<arch><sub\_arch>-<vendor>-<sys>-[<env>]**

* + for example *aarch64-unknown-linux-gnu*

Additionally, certain *features* can also be specified to enable different functionality within the target machine.

The LLVM API is used to resolve the target from the target-triple string and enable features, and internally create the *LLVM Target Machine* object. The implementor of the specific target machine would then implement various methods of this object to obtain information relating to the memory layout, data word size, code model etc.

Once there is a target machine then an *LLVM Data Layout* can be created. The data layout specifies how to lay out structures, the alignment for each data type, the size of a native pointer and the alignment for the pointer. This information is associated with a particular target machine.

Another important aspect of a Data Layout is the *Address Space*. This is an abstract concept as far as LLVM is concerned, but it enables a target to have variations in a data layout for different types of memory, for example. This is an important concept when it comes to Morello, because C64 uses a different *address space* than A64 and this affects the pointer size and alignments required for a full capability pointer (16 bytes) vs a standard AArch64 pointer (8 bytes).

The data layout can be expressed as a formatted string, which has meaning to LLVM. This is written into the IR so that any processor of the IR can understand the data layout.

Note that although LLVM IR is a generic language, it is still necessary to set a Target Machine before generating IR in order to get information about the Data Layout. This is because, for pointers, the IR needs to specify sizes and alignments and this can only be known if the target is known.

Consider the below, which is four lines of LLVM IR code:

**%global\_ptr\_tmp = getelementptr inbounds i8, i8\* %aot\_inst, i32 672**

**%global\_ptr = bitcast i8\* %global\_ptr\_tmp to i32\***

**%global = load i32, i32\* %global\_ptr, align 4**

**store i32 %global, i32\* %l0, align 4**

It can be seen there are no machine specifics in this IR, however there are offsets (e.g 672) and alignments specified which require the target machine to be known.

#### WASM Parser

The WASM parser implementation “front” is very similar to WAMR’s interpreter mode. Each function in the WASM file is parsed, and then for each WASM opcode action is taken. However this action is not to execute the WASM instruction but instead to generate LLVM IR for it.

*wamrc* implements methods to compile WASM opcodes to LLVM IR by using the LLVM API to instruct an LLVM internal *Builder* to generate and populate *Blocks* of execution. In this way, WAMR can generate a representation of the IR inside an LLVM object model.

As will become apparent for CHERI Morello porting, one key LLVM operation is to generate a *Get Element Pointer (GEP)* instruction. The GEP is used to define how to access some memory location given a structure, index or array for example (an LLVM *load* or *store* instruction tells how to r/w the memory location). It therefore needs to know the alignment of pointers from the data layout for the target machine.

*Parsing Other Sections*

A WASM file is more than just function code – it also includes globals, tables, memory, imports and initialised data sections. These are all parsed in same way as for the interpreter mode and internal WAMR *module* structures are filled in, and data copied from the WASM file buffer as needed.

#### LLVM Verification and Optimisation

After generating the IR object mode, *wamrc* will verify the module and then perform an optimisation step (unless the user requested no optimisations from the command line).

The Verification step uses the LLVM API to run an internal verification pass of the module. This will check the module to ensure it is error free. There are many steps to this, essentially they involve checking for type and value errors – more information is available online.

The optimisation step creates an LLVM *Pass Manager* which configures the different analysis and optimisation passes that *wamrc* wants to run through the object code. The pass manager API includes the ability to set an optimisation level – this comes from the *wamrc* user, from the *wamrc* program command line. The optimisation passes are then run through the LLVM API.

There are numerous types of LLVM optimisations – refer to *aot\_apply\_llvm\_new\_pass\_manager()* in *aot\_llvm\_extra2.cpp* for more details of the ones used by *wamrc* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/core/iwasm/compilation/aot_llvm_extra2.cpp>). However one optimisation is of particular interest – the *LoadStoreVectorizerPass()*.

This optimisation is designed to eliminate multiple load/stores of sequential memory locations, with the assumption that this could be quite time consuming to have to access potentially slow memory frequently. Instead, memory is accessed as a *vector load / store* – a means to access an amount of memory in less cycles than accessing individual locations, typically to a vector register. On a SIMD architecture this could lead to performance benefits.

However on Arm this optimisation is not being used by the backend and there is a bespoke optimisation pass which makes better improvements. This is significant for two reasons:

1. As Morello is an *aarch64* architecture, it adds no value to perform this step
2. As will be discussed later, this step cannot be performed on pure-cap due to C64 using a non-default address space with *index size != pointer size*, and LLVM will assert if it is attempted

#### Object Code Generation

The AOT compiler can function in one of two modes to generate target-specific machine object code:

* Use the internal LLVM library API
* Use an external compiler or assembler

*wamrc* allows an external compiler to be used if environment variable *WAMRC\_LLC\_COMPILER* is defined to the compiler to use, or will use an external assembler if *WAMRC\_ASM\_COMPILER* is defined.

*External Compiler*

In the case of an external compiler then an LLVM IR file is generated as a temporary file, and an external compiler used to generate target object code to another temporary file. This is then parsed back in, and used as part of the AOT file which is generated. Arguments to the external compiler are generated based on the *target-triple* which is generated from the *Target Machine* set previously.

Alternatively, an environment variable can be used to specify custom arguments to pass to the command line string of the external compiler program.

*External Assembler*

This is much the same as for an external compiler, except instead of IR an assembly file is generated. The object code is then created by assembling this file, and this is then read back in for inclusion in the full AOT file.

In the same way as for the external compiler, command-line arguments are generated for the target needed or alternatively an environment variable can be set to pass custom arguments.

*Internal LLVM Library*

In this final case there is no need for a file as the object data is emitted directly to a memory buffer, using the LLVM API call to generate it from the internal IR object model. This object code is then in the buffer just as it was for the previous two methods. The LLVM backend libraries specific for the *Target Machine* are used to compile IR to object code.

The flow for the different modes is shown below:

A diagram of a computer program

Description automatically generated

#### AOT File Generation

An AOT file can be thought of as an amalgamation of an ELF file (or COFF for some targets), all the WASM sections which are not related to function code, and header areas which are used to pass information about the target and the AOT file.

The AOT file format does not seem to be specified anywhere. Instead, the format is determined by the order in which the different parts are output – this matches the AOT loader in WAMR, which reads the file in the reverse order it was output.

To output the AOT file, *wamrc* first creates the *ELF Data* and then emits the different sections (which includes ELF sections but also the WAMR WASM data structures).

*AOT ELF Data Creation*

Using the LLVM API, the object data in the memory buffer from previously is converted to a *Binary* format which for targets on Linux is basically ELF. An ELF “blob” comprises a number of named and sized sections along with data for those sections, for example *.text* would be a code section and the code data could be disassembled to assembly instructions. Other important sections are *.rodata* for initialised (read only) data, *.symtab* and *.rel[a]* for the symbols relocation table information which are used to resolve symbol mappings at runtime.

*wamrc* needs to parse all of the relevant sections and extract the metadata and contents into its internal AOT structures. Once it has done that then (later) this information can be emitted into the AOT file itself.

Specifically, wamrc looks for the following sections, in the order given (not all may be present and the order doesn’t matter as wamrc will always pass through the whole list):

| Section | Type | Description |
| --- | --- | --- |
| Header | Metadata | Describes the target information, binary format (ELF, COFF, etc.) and endianness |
| .text | Code | Target specific object code |
| .literal | Data | Constants for assembler on some platforms |
| .rodata /  .rodata.cst\* /  .rodata.str\* /  .rdata | Data | Initialized data of various types |
| .rela.\* / .rel.\* | Tables | Symbol Relocation Information |
| Function Mapping  (not an ELF section) | Internal | Wamrc builds a symbol mapping for all AOT functions within the text section, so it can convert function name to offset within the *text* section |

**NOTE:** When parsing the symbol relocation information wamrc will build structures of data which can be used to create the actual relocation table data itself – it seems the LLVM API doesn’t directly provide for this. This resolves:

* Symbol mapping type (tells a linker what to do with the other information)
* Offset into symbol table to find the address of the symbol to be used (known at runtime)
* Offset into the section (e.g *.text*) for which e.g instruction needs to be patched at runtime
* *Addend* (only applicable for *.rela*) which is the additional amount to add to the address of the symbol (which may be zero)

After all that is done, WAMRc has enough information to actually emit the AOT file.

*AOT File Emitting*

The actual output of the AOT file itself takes place by writing the different parts to a buffer, in strict order, and then writing the buffer to a file.

The AOT format includes an overall header and is then “self-describing” i.e the size of subsequent chunks of data is then given, followed by the data – a form of *Type, Length, Value* (TLV) fields.

The following is output, in this order (simplified here for the aid of understanding):

| Part | Sub-Part | Details |
| --- | --- | --- |
| AOT File Header |  | A text header “*\0aot”* followed by the current version number |
| Target Info |  | Basically an ELF header: machine type, endianness, ABI type etc. |
| Data Sections | Memory Info | WASM Memory metdata +  Initialized WASM *Data* sections |
| Tables | WASM Import / Initialised Tables Data |
| Functions | List of all WASM Functions (names, params count, types) |
| Global Imports | WASM Global Imports details (names, modules) |
| Globals | WASM Declared Globals |
| Function Imports | WASM Imported Function Declarations |
| Misc | WASM Stack / Heap size etc. |
| Object Data Sections | Data sections parsed directly from the ELF, copied directly (so named *.rodata* etc.) |
| Text Section |  | *.text* from the ELF, i.e the object code |
| Function Mapping |  | Function metadata built earlier (function names, mapping into ELF etc.) |
| Exports |  | Any exports defined in the WASM |
| Relocation Info | Symbol Table | Symbol table generated dynamically from relocation info parsed previously from the ELF |
| Relocation Tables | Relocation Data parsed previously from the ELF |
| Custom Native Symbols |  | Any native symbols (as function names) of native code needed at runtime |
| Custom Name Section |  | A WASM custom name section (typically used by a WASM debugger) |
| Custom Raw Section |  | A WASM custom section passed from the WASM |

The above is, then, the structure of an AOT file. Not though that the AOT file also includes various lengths and other fields during the output, along with section headers comprising Type, Length, Value.

#### Generating non-AOT File Formats

For test and debug purposes *wamrc* can also be told to output an LLVM IR file or a target specific object file.

Emitting an LLVM IR file is simple, because the LLVM API will already have the object model of the IR. A simple API call to LLVM will request it to print IR to a file.

For an object file, the LLVM IR needs to be compiled to the target specific machine code. In the case of an external compiler or assembler being used this is done via a temporary LLVM IR file being created and then the object file produced by the external tool. In the case of using the internal LLVM libraries, LLVM is told to emit a file in object code format which it will generate using the target machine backend.

**NOTE:** *Object file generation is effectively a subset of the process which is described in Section 7.1.1.3.*

### AOT File Format

The AOT file format is shown below:

A green square with white text

Description automatically generated

There is first the header preamble, which comprises a string “\0aot” (note: “\0” == NULL byte as per C string notation) and a magic number (32-bits) which indicate this is a valid AOT file. This is followed by a file binary version which is just a 32-bit integer. If the version does not match that hard-coded by WAMR then the AOT file is rejected due to incompatible version (no attempt is made to support backwards compatibility of AOT versions within WAMR).

Each part begins with a Type identifier, and this is then followed by the length of the data which follows immediately after the length. Note that the WAMR AOT loader does not require parts to always appear in the order shown above, although this is in practice how they are generated. Although all parts are not mandatory, in practice they are all always generated apart from the *Custom* part. Note that the different part type IDs are checked to ensure they are all below a terminating value, apart from *Custom* which is set to an ID == 100 (presumably to allow for expansion if needed in future).

Within each part the data structure varies, there may be sub-parts which are in a predetermined order. Section 7.1.1.6 covered the details of sub-parts within the AOT file.

### WAMR AOT Support

WAMR supports loading an AOT file using the *AOT loader* and then running it using the *AOT runtime* instead of the interpreter or JIT runtime. To enable support for AOT in WAMR then the *CMake* build flag *WAMR\_BUILD\_AOT=1* should be used.

When AOT is included in the build, WAMR builds in the AOT parts of the codebase from *core/iwasm/aot:*

(<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/core/iwasm/aot>).

This is:

* The AOT loader, parses AOT file into AOT module data
* The AOT runtime, executes AOT file
* Architecture specific code to support relocation symbol resolution at runtime

WAMR detects at runtime whether the input module is an AOT file or a WASM module. It does this as part of *wasm\_runtime\_load()* in *wasm\_runtime\_common.c* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/core/iwasm/common/wasm_runtime_common.c>). A WASM file starts with *“\0asm”* whereas an AOT file starts with *“\0aot”*, which is how WAMR detects the file type.

For an AOT file, WAMR then loads the AOT file format which has been described previously and instantiates the module data. This will provide all the information about the module, including:

* data that came directly from the WASM e.g initialized memory
* Code to be executed, as target specific object code which can directly be run natively
* ELF style symbol and relocation data which is used with the object code to resolve native function addresses and data addresses at runtime

When the module is available, it can then be executed by calling a particular user supplied function or an entry function such as *main()*. For AOT, this uses the *aot\_call\_function()* in *aot\_runtime.c* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/core/iwasm/aot/aot_runtime.c>) to call native code to execute the function. This is because all code in AOT mode is already compiled to machine object code, and the AOT file informs the offset within the *.text* section to the function entry point.

AOT code then runs and during execution it may call functions (natively) in *aot\_runtime.c* such as *aot\_check\_app\_addr\_and\_convert()* to convert pointers at runtime and *aot\_set\_exception\_with\_id()* if an error occurs. These functions were added as externals by the AOT compiler’s WASM interpreter to IR compilation step, so are called as part of the standard object code processing.

The AOT function then completes, and the native function exits in the normal WAMR way and the program terminates with suitable return code as for an interpreter mode program.

The high-level flow of WAMR in AOT mode is shown below – details then follow in the subsequent sub-sections of this part of the document.

A diagram of a software company

Description automatically generated

#### AOT Loader

The AOT loader simply needs to parse the AOT file and read in the contents. It first checks that the AOT version matches the one WAMR is built with, and then loads sections into a *sections list*. This can be done without knowing details of the section because, as described for the AOT file, each section starts with a *Type* followed by a *Length* and then *<Length>* bytes of the sections *Value* (i.e its data).

There is one bit of processing which happens though at this early stage – the *.text* section is identified and then a buffer is *mmap()*ed to contain the data. This is mapped with READ | WRITE | EXECUTE permissions – later the buffer could be switched to READ | EXECUTE on some platforms. This is necessary because the code is loaded as data, but later needs to be executable.

The code section data (i.e the object code) is then copied to the new buffer, and to make everything “just work” for future operations the pointer to the data in the sections list is updated to point to the new buffer.

Note that if the AOT file was built with the *wamrc* option XIP (execute-in-place) then the *mmap()* and copy is not done and instead the execution happens from the buffer holding the AOT file. Therefore the “pointer to section list data” mechanism works for both XIP and non-XIP.

Once the sections list is created, sections can be loaded via *load\_from\_sections().*

Loading the sections is the reverse process of what was done when each section (or “part”) was created in *wamrc*. Sections can appear in any order, and once the section is identified then the individual layout is know and it can be parsed.

The following describes operations of loading the different sections, also referred to as “parts”:

| Part | Sub-Part | Operation |
| --- | --- | --- |
| Target Info |  | Validate the ELF fields and check target identification matches the target WAMR is running on (e.g for pointer size) |
| Data Sections | Memory Info | Load details of the WASM *memory* section into module data, and copy any initialized *Data* into to allocated module structures |
| Tables | Parse WASM table information from AOT file buffer and copy table data into allocated module structures |
| Functions | Parse function information and build a list of function types (num params, return type etc.) |
| Global Imports | Build list of all WASM global imports |
| Globals | Append to global imports list with all WASM globals |
| Function Imports | Create a list of named functions, mapping name 🡪 native function’s symbol address. Native symbols are those supplied by the user or e.g WASI.  Map the function’s type to one of the function types parsed in *Functions* sub-section, above. |
| Misc | Load WASM Stack / Heap size by parsing AOT file data |
| Object Data Sections | Create memory for the ELF data sections (e.g *.rodata)* using *mmap()* with protections = READ | WRITE.  The contents of the section is then read from the AOT file data and written into the newly mapped buffer.  **See below for more information.** |
| Text Section |  | Init pointers to the code and load the *PLT (Procedure Linkage Tables)* – **see below for more explanation** |
| Function Mapping |  | Load the function mappings. These were previously set up as an offset into the .text section. But we have a buffer for the code now, so sets actual pointers into the code for each function. |
| Exports |  | Loads table of all exports from the AOT file data |
| Relocation Info | Symbol Table +  Relocation Tables | Load symbol table and the relocation table information.  Use these to actually patch up the code with the actual value, for each symbol (architecture specific based on relocation type).  Finally, now the code manipulation is complete, set the *.text* to *mprotect(READ|EXECUTE)* and initialized data to *mprotect(READ)*.  **See below fore more detail.** |
| Custom Native Symbols |  | Builds linked list of native symbols, with addresses resolved to known native functions in AOT code and libc |
| Custom Name Section |  | Loads into module structures  ***Note****: This functionality is incomplete in the current WAMR implementation* |
| Custom Raw Section |  | Loads as raw data into module structures  ***Note****: This functionality is incomplete in the current WAMR implementation* |

Some parts of the loading, relating to actions a linker or a C runtime would normally perform, are complicated and deserve more explanation. In particular these had to be well understood for this project work as they needed to be changed for CHERI pure-cap. These are described in the following sections.

#### AOT Procedure Linkage Table

This section explains in more detail how the WAMR AOT PLT is implemented.

*Procedure Linkage Table Overview*

The problem with calling functions loaded from shared libraries, or code which is *Position Independent (PIC)* is that only at runtime is the actual address of the function to call (in the caller’s address space) actually known.

To deal with this a linker would implement a PLT and resolve the symbol address to be an actual address during running the PLT. The PLT then acts as “trampoline” to call on to the actual address. At runtime the actual address is resolved and patched up into the PLT entry. The symbol itself would likely come from the *Global Offset Table (GOT)* which maps all actual address to symbol offsets at runtime.

This is all handled by creating a *.plt* (or similar) ELF section which contains the “trampoline” code. The picture below demonstrates the process:

A diagram of a call center

Description automatically generated

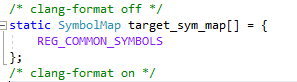
The PLT code for a symbol can simply save the caller’s return address, read the actual address filled in at runtime from a GOT offset, call it and then return to the caller’s return address afterwards.

*Setting up Procedure Linkage Table in WAMR*

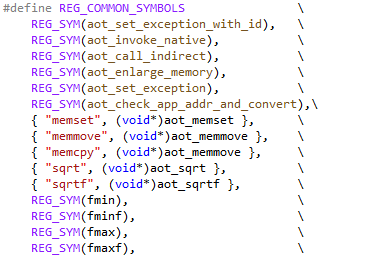
The process is slightly different in WAMR because there is no linker. Instead, WAMR implements the PLT and instead of a GOT it fills the address into a location within the PLT when mapping symbols that it knows about because they are part of the WAMR executable.

WAMR also needs to take care of the patch-up of the symbol relocation as well, to direct the call to call into the correct handler in the PLT. Reference *symbol relocations patching,* below, for details on this.

WAMR maintains a *symbol table* of all native symbols which are available to it at runtime (i.e they will be built into the executable). Here we can see it for *AArch64*:



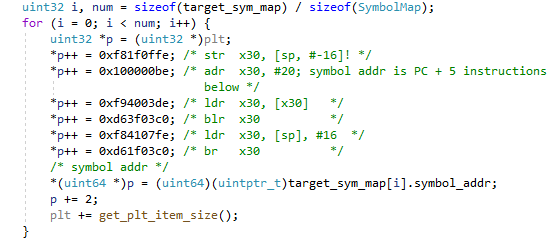
A snippet of the *REG\_COMMON\_SYMBOLS* macro is shown below:



Taking *memset()* as an example, we can see that this table maps the symbol name as a string to the symbol address in native code.

WAMR then builds a PLT entry for each symbol, whereby the address is inserted as a constant into the code. (The PLT entries are written as data but are executable object code). As PC based addressing is used, the actual symbol address to call is given by a known offset from the PC to the symbol address that has been inserted.

Here is how the PLT entry is produced for *AArch64:*



The above iterates round all the symbols in the table and then writes the assembly code shown (as opcodes) and the address of the symbol. In the above:

* Caller calls this PLT entry for the given symbol
* Caller’s return is in the link register (X30) so save it
* Address for symbol address is PC + 20 bytes, writes this to X30 and then reads to address to X30
* Call the actual symbol @X30
* On return restore the original LR (X30) and return back to the calling function

It should be noted that *the PLT entries are created in the order they are listed in the symbol mapping table*.

**NOTE:** WAMR places the PLT at the end of the *.text* section, and allocates space for this when *mmap()*ing the code region.

*Using PLT in WAMR*

When the AOT compiler generated the AOT file, ELF sections were created for relocation information. Any symbols which are now referenced in the PLT entries could not be resolved, so instead they were added to the relocation table for patch-up at runtime.

In the case of the symbols in the PLT, the patch-up type would be a *CALL* type indicating a function call.

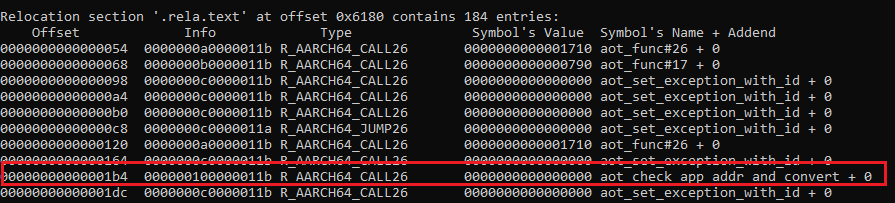
When WAMR process relocation information, it will encounter these as part of standard relocation. WAMR then needs to know how to calculate the resolved address – this depends on what the symbol actually is:

* It may be a function within the *.text* section, i.e AOT code which was compiled
* It might be a data symbol, if it is within a *.rodata* or other data section
* Or it might be a native function in the symbol table

WAMR uses the symbol name to resolve. It checks to see if the symbol name is within the symbol table *target\_sym\_map* shown above. If so, it then reads the *index* into the table to determine the PLT entry address to call.

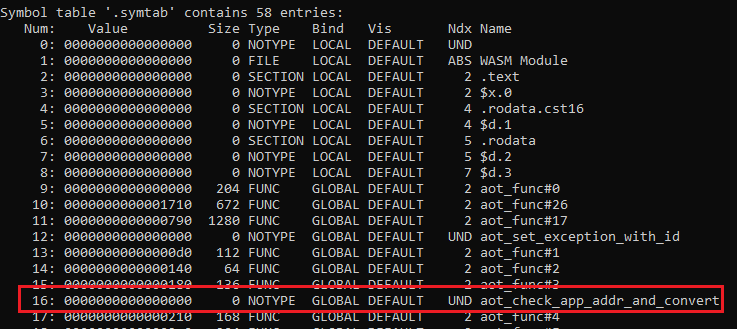
Here is an example, using the WAMR *aot\_check\_app\_addr\_and\_convert()* function. With reference to the above code screenshot, we can see it is in the symbol table at zero-based-index 5.

In this compiled AOT object code, we can see from an ELF dump that there is the need to patch up this symbol:



This tells us the code at *.text* + 0x1b4 needs patching. The entry in the symbol table is 0x10 (i.e 16) and the patch up type is 0x11b which is *R\_AARCH64\_CALL26* i.e a function call.

If we analyse the symbol table from the ELF we see this:



At entry 16 there is the symbol *aot\_check\_app\_addr\_and\_convert()*. As this is external it is UNDefined and NOTYPE because the ELF file has no further information about this symbol.

When WAMR is dealing with this patch-up at *.text + 0x1b4* it will discover that a symbol with name *aot\_check\_app\_add\_and\_convert()* is found at index #5 in the symbol table. Therefore the patch-up address will be:

* + PLT\_start + (5 x size\_of\_a\_PLT\_entry)

In fact, as the PLT starts at the bottom of the text area, the calculation is:

* + *.text actual address (mmap()’d) + size\_of\_code* + (5 x size\_of\_a\_PLT\_entry)

#### AOT Setup of ELF Data Sections

At AOT compile time, when the LLVM IR was converted to a binary it generated a number of ELF sections. As an AOT file is not an ELF file the ELF data was parsed and individual sections were written to the AOT file with appropriate headers.

In WAMR, the runtime needs to actually locate these sections somewhere in memory. The *.text* section has already been explained, this was *mmap()*ed to R|W memory which is then made R|X so it can be executed (or for XIP mode, it used the AOT file buffer itself).

The same needs to occur for the initialized data sections. To do this, WAMR *mmap()*s a buffer for each as R|W data and then copies the contents of the ELF section from the AOT file buffer to the *mmap()*edbuffer.

This step is (and needs to be) done before patching relocation symbols, since object code needs to resolve data from “this offset in an ELF file” to “this actual location in the processor’s address space”.

#### AOT Symbol Relocation Patching

In order to enable symbol relocation resolution, the following must be in place – which is why all of these are processed first:

* The object code (*.text*) needs to be in its final destination address
* The PLT must have been completed
* The initialized data sections must be in their final destination addresses

The relocation types are target specific because the opcode instructions which must be resolved are target specific. For AArch64, WAMR implements the relocation code in the same file which builds the PLT, *aot\_reloc\_aarch64.c* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/core/iwasm/aot/arch/aot_reloc_aarch64.c>).

The symbol relocation process involves parsing the *relocation tables* and the *symbol table*. This is a general ELF concept carried out by a linker and runtime, so the general part of the process is just summarised here.

*Symbol Relocation Explained*

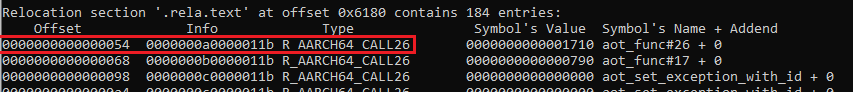
Each of the relocation tables will apply for either *code* or *data* and either include an *addend* or not (*.rela* vs *.rel*). An entry in the relocation table informs the following:

* Offset of where the patch is needed
* Info, which comprises (on AArch64) 32-bits as an index into the *symbol* table and 32-bits as a symbol relocation type identifier
* The addend (if applicable)

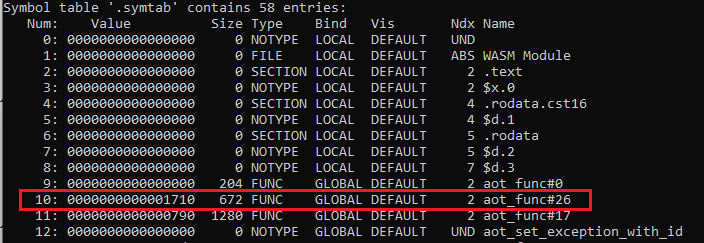
The symbol table can then be examined, at the given index, to resolve the run-time symbol. If the symbol exists in the ELF then it’s offset from the start of a given section is provided. If it is external symbol this is set to zero, as the ELF does not know where it is.

**NOTE:** *In the ELF file, the .strtab keeps a list of all names as strings – ELF formatted data maps values to a string index, which is how tools such as “readelf” can display the names*.

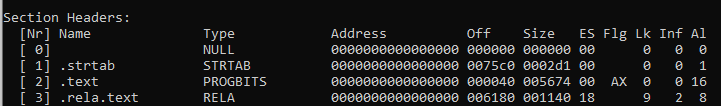
The below shows an example dump of a relocation table, *.rela.text*. The first entry shows *.text* + 0x54 needs patching – and it’s a type *R\_AARCH64\_CALL26* - the addend is zero and if we consult the symbol table index 0xa (10) we can find the details.



The symbol table snippet is:



The *Ndx* entry is #2 so this is a section in the ELF file, and the offset to the address to patch with is 0x1710. Consulting the section headers:



Section number #2 is *.text* so the address to patch with is *.text + 0x1710* which the *readelf* tool has already displayed for us as *aot\_func#26 + 0*.

The *type* of patch up needed is *R\_AARCH64\_CALL\_26* which is used in a *BL* instruction. The Arm64 ABI specification tells how to generate this patch type, it is a 26-bit immediate value which is a PC relative address. Therefore the address can be calculated as follows:

* Absolute address required = *.text start + 0x1710*
* *Addend = 0*
* Address to patch = *.text start + 0x54*

Value to write = (*.text start + 0x1710) + 0 – (.text\_start + 0x54)*

More commonly, and in the WAMR code, this is known as *S + A – P* where *S* is the symbol address needed, *A* is the addend and *P* is the address to be patched.

*WAMR Implementation of Symbol Relocation*

WAMR processes symbol relocations during the *load\_symbol\_relocations()* function.

It first loads the symbol table *(.symtab*) into memory, and then each relocation table as a *relocation group*. WAMR then parses each relocation group in turn, which is handled differently if the group is a *code* group vs a *data* group (this is determined by the ELF section name).

For each symbol, WAMR needs to determine the absolute address for the symbol. It does this using the symbol *Type* and looking up by name, rather than attempting to calculate for an offset from the symbol table. The following are checked:

1. If the symbol name starts with “*aot\_func#”* then WAMR deduces it is a function in the object code, and resolves the absolute symbol address using the function name -> address mapping set up previously
2. If the symbol is a section start, then resolve the address from the list of known section start addresses
3. If the symbol is a pre-defined native function, then resolve to the PLT entry of the symbol index
4. Else fail

The symbol address (S) is then known, the patch address and addend (P and A) are known and the relocation type is also now known therefore WAMR calls *apply\_relocation()* which is the target specific function.

The *apply\_relocation()* implements a different process for each relocation type defined for the architecture. The general process is:

1. Calculate *S + A – P* which is the value to be patched in
2. Prepare the value as needed (validate, sign extend if required)
3. Read the data from the address to be patched (@P), which is instruction opcodes
4. Modify the opcode to replace the address value part with the newly calculated value

A real-world example from running WAMR in AOT mode is shown below, for the *R\_AARCH64\_CALL26* type:

1. P = **0xfffff7ff705c**
2. @P *before operation*= **0x94000000 = opcode “BL #0”**
3. A = 0
4. S = **0xfffff7ff8754**
5. S + A – P = **0x16F8**
6. @P *after operation* = **0x940005be = opcode “BL #0xfffff7ff8754”**

#### AOT Module Instantiation

This is the second and simplest part of the setup phase.

The instantiation is almost the same as the WASM Interpreter mode instantiation, and both AOT and Interpreter modes create a *WASMModuleInstance* structure.

From the module data, the different data is constructed ready for the runtime. Recall that any native function (which includes AOT object code functions) will be able to access the *execution environment* and hence the *module instance data*.

As part of creating the module instance data, the dynamic buffer part of the structure is set up. As described for WASM Interpreter mode, this comprises blobs of freeform memory appended after the *WASMModuleInstance* (== *AOTModuleInstance*) structure.

As part of the instantiation phase, if WASI support is being compiled then the initial functions which must be called as mandated by WASI (such as *\_\_initialize())* are called at this time, because the actual runtime function will be native code.

Additionally the execution environment is created, which adds the sandboxed stack and heap.

#### Runtime Function Execution

Function execution for AOT mode ends up calling *wasm\_runtime\_invoke\_native()* in *wasm\_runtime\_common.c* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/core/iwasm/common/wasm_runtime_common.c>) in exactly the same way any call to a native function is made. As stated, the execution environment and hence the module instance data are made available to the function.

The native function will be within the previously configured *.text* region. It may call back to other WAMR functions as already discussed.

## AOT Compiler Changes for CHERI

This section describes the changes which were needed to support CHERI Morello in the *wamrc* AOT Compiler.

This section covers **only** changes which are isolated to the AOT compiler. System wide changes, affecting both compiler and runtime, are documented separately.

The main challenge was due to the *cross-compilation* that was needed. That is, we are compiling an AOT file designed to run on Morello but on a host machine which is Linux x86\_64. This is particularly challenging because pure-cap has a 128-bit pointer size / alignment vs 64-bit on the host machine.

The flow is then:

* Run *wamrc* on Linux x86\_64 but specify an *AArch64* Morello hybrid or pure-cap target
* Generate AOT for this native target, noting that the pointers may be twice the size of the host machine
* Transfer AOT file to Morello and execute with WAMR built for the Morello target

This is shown below graphically:

A diagram of a computer

Description automatically generated

This step is even more challenging because *wamrc* is designed to be able to built for *generic targets* whereas WAMR is always built for a single target. This means that we should be able to generate an AOT file for (for example) x86\_64, Morello hybrid and Morello pure-cap from the same *wamrc* program instance even though we will need a platform-specific WAMR program instance to execute each.

So far as possible, the objective was to achieve this use case of *wamrc*.

*Porting AOT Compiler to Morello?*

Given the difficulties, one option would have been to port *wamrc* to execute on Morello. Then there would be no issues with *wamrc* running on a different machine than WAMR, and the data layout used in generating WASM -> LLVM IR would be the same data layout as will be used at runtime.

However, this option was swiftly rejected because:

1. Porting *wamrc* would have been very complicated with its own issues, and would likely have doubled the AOT development time
2. At the time of writing, LLVM toolchain libraries (a *wamrc* dependency) is not supported on Morello pure-cap and having to run for Morello hybrid-cap would have likely have yielded similar problems
3. At the time the AOT work was done, Arm’s LLVM toolchain port was not particularly mature, in fact there was no official release supporting building C++ for Morello pure-cap. It was felt that it was too risky to try and use the APIs running on the Morello machine, or trying to make this work on pure-cap, even if pure-cap libs were able to be supported.
4. If this *wamrc* port could actually be done, it would almost certainly lead to the same sort of problems trying to build for other targets on Morello and therefore the flexibility in *wamrc* would be lost as it would become versioned to only run on the native host
   * The *ByteCodeAlliance* have designed *wamrc* to be quite flexible; therefore they would likely dismiss any Morello changes made out of hand and there would be trouble getting our work added to the main WAMR repository

### Building for Cross-Compilation

As discussed, *wamrc* requires an LLVM project and a number of LLVM API libraries. In our case we required a version which has the additional *Target backend* of Morello. This meant using the Arm LLVM port, which adds this support as a sub-set of *AArch64.*

It was initially considered that it would be possible to use an external compiler via the *WAMRC\_LLC\_COMPILER* environment variable. Actually, this was the first step to the development – we verified this could be called from the *wamrc* program. This was though not an option because:

* LLVM Internals still needed to know about the target
* It would have been unnecessary “clunky” going forward (too many command line options and different setup for the external compiler)

At the point the AOT work began the Arm LLVM Port for Morello + MUSL libc was mature enough to support using it as the *wamrc* internal LLVM and so this option was pursued.

#### CMake Changes

The *CMakeLists.txt* in *wamrc* is designed to be able to specify a different LLVM than the default, via *WAMR\_BUILD\_WITH\_CUSTOM\_LLVM*. However it does not allow to find an LLVM that is not on the path. To support this, the *LLVM\_DIR* used to *CMake* to find the LLVM package simply needed to be set. This can be done from *CMakePresets.json*, and must point to the location of the *CMake* makefiles in an LLVM release, which is:

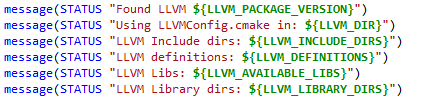
*<llvm\_out\_or\_release\_folder>/lib/cmake/llvm*

Here is an example from the *CMakePresets.json* for the project:



Note the additional variable *WAMRC\_TOOL=1*. This is because we have a root-level *CMakeLists.txt* which then includes either *wamrc’s CMakeLists.txt* or WAMR’s, depending on if *WAMRC\_TOOL* is 1 or 0. This is just a construct to be able to run from Visual Studio, see Section 3.5.2 for more on this.

For increased *CMake* logging output, to verify the correct LLVM folder was being used, some LLVM Macros were echo’d to the build output as shown below (modified *wamr-compiler/CMakeLists.txt* file):



The final *CMake* problem was, given the LLVM targets are not on the path, then how can *wamrc* find them at runtime?

One solution is to statically link; however this leads to an unmanageably huge image *and* causes the build to be incredibly slow and even to run out of RAM on the Linux machine (!) when preparing a debug image. This latter issue was solved by using the *gold* linker instead of *lld* or *ld* (pass *-fuse-ld=gold* as a linker option from CMake), so static linking is possible but ill-advised.

The real solution is to make sure the *wamrc RPATH* is set correctly – this can be done in *CMake* with a simple flag, to force it to be the same as the link path:

**set(CMAKE\_INSTALL\_RPATH\_USE\_LINK\_PATH TRUE)**

### Creating Morello Targets

For *wamrc* it is not possible to specify the same arguments as e.g *clang* to set the target machine. Internally though, it was needed for LLVM to be instructed by *wamrc* to use the correct *Target Machine* for both the Morello hybrid-cap and pure-cap cases.

Clearly Arm had changed the backend target library for *AArch64* but this was either undocumented or a document source code not be found. Therefore the *wamrc* and LLVM source code was analysed to understand exactly how the *AArch64* target needs to be configured for both hybrid and pure-cap.

The solution though had to stick with the way *wamrc* currently works – which is to attempt to deduce the *target-triple* “<vendor>-<sys>” from the ABI argument and then discard the ABI. This won’t work on Morello as we need to pass an ABI in the pure-cap case. It is also important to ensure *wamrc* won’t stop working on any other targets, including any non-Morello AArch64 targets.

The complete solution works as follows:

1. *Target* passed as “aarch64”, and this remains the *<aarch><subaarch>* part of the triple
2. If it is a Morello target, set *<vendor>-<sys>* to *unknown-linux*
   * Detect Morello by presence of *+morello* in *cpu-features* argument
3. *ABI* argument is passed directly as *<env>* part of the triple
   * This is default behaviour in *wamrc* so nothing needs to be done here
4. *wamrc* does not permit setting the actual ABI to the LLVM library, therefore:
   * If *cpu-features* includes “+morello” and “+c64” and *ABI* includes *“*purecap” then conclude it is Morello pure-cap and so set the LLVM “ABI” argument == “purecap”
5. Pass CPU features “as is”

The one slight annoyance is that *wamrc* mandates CPU features is only allowable from the command line if “CPU” is also set. We therefore set CPU = *“rainier”* as the Morello is comprised of Rainier clusters, it seems the closest match although it is not actually used by the backend.

The resulting triple generated is then:

**aarch64-unknown-Linux-<environment>**

Where *<environment>* is one of:

* musl\_purecap
* purecap
* cheri\_purecap
* gnu / don’t care (for hybrid)

As mentioned, the *--target-abi* option needs to contain the string “*purecap*” to enable Morello pure-cap mode.

The --*cpu-features* argument must then be “*+morello,+c64*” for pure-cap and “*+morello*” for hybrid.

#### Command Line Arguments

The result is new options are added to the *wamrc* command-line arguments for *ABI*. The following explains how to tell *wamrc* to generate for Morello hybrid-cap and pure-cap targets.

Example command-line for Morello pure-cap:

**wamrc --target=aarch64 --target-abi=musl\_purecap --cpu=rainier --cpu-features=+morello,+c64 -o test.aot test.wasm**

And hybrid-cap:

**wamrc --target=aarch64 --target-abi=gnu --cpu=rainier --cpu-features=+morello -o test.aot test.wasm**

Notes:

* The feature *+morello* is required, and for pure-cap *+c64* is also required
* As well as *musl\_purecap*, *gnu\_purecap*, *cheri\_purecap* and *purecap* are also valid pure-cap target ABIs
* If *c64* is specified, the ABI must contain the string *purecap*
* The CPU is not explicitly required, but *rainier* is the closest match

As mentioned, if the *morello* feature is enabled, then the target triple string is built as **aarch64-unknown-linux-<ABI>** for example **aarch64-unknown-linux-musl\_purecap**

This matches the arguments passed to *clang* when building for Morello.

Shown below is output from running *wamrc* with a request to display valid ABIs. Newly added ones shown in red box:



### Morello Address Space Support

When examining the LLVM *Data Layout* for AArch64 target machine with Morello pure-cap enabled it was noted that Arm have defined a new *LLVM Address Space* which must be used for all capability pointers. If the default address space is used then pointers will be treated as 64-bit, not 128-bit.

This can be seen by examining the LLVM Data Layout string for a Morello pure-cap enabled AArch64 Target (e.g as output in LLVM IR bitcode):

**target datalayout = "e-m:e-pf200:128:128:128:64-i8:8:32-i16:16:32-i64:64-i128:128-n32:64-S128-A200-P200-G200"**

Here the *pf200:128:128:128* informs that for *Address Space* 200 then pointers are 128-bits in size and alignment, but not for the default Address Space (0) where they are 64-bit.

*A200, P200* and *G200* are also useful here as they inform the *Address Space* used for program, globals and allocated memory. On pure-cap this is the address space 200 not the default one.

The Address Space is important even for the *wamrc* parser step because it defines how GEP instructions will be output (refer to Section 7.1.1.3 for more on this).

The problem is that *wamrc* does not specify an Address Space (AS) in any of the operations where one could be used – this will fall back to the default AS. This necessitated modifying the code throughout in order to specify an AS.

#### Obtaining Morello Address Space

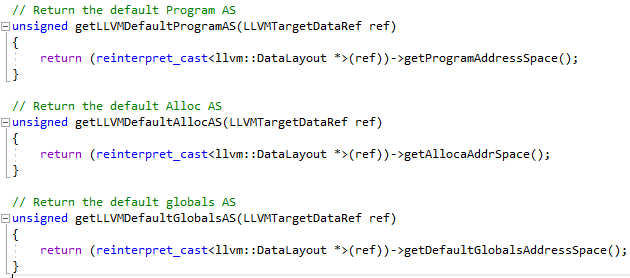
The Morello address space for capability pointers has been set to 200, for unknown reasons. However it would be poor practice to hard-code this in *wamrc* not least because the Morello ABI is still in draft.

Instead, we utilise the fact that the default AS for program, globals and allocated memory:

* Is 200 on Morello pure-cap
* Is 0 on Morello hybrid-cap
* Is 0 if not specified, i.e the default

This means that, for any architecture, we can request the address space and use this in any operation in the *wamrc* parser which requires it.

To support this, the following code is added to *aot\_llvm\_extra2.cpp* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/core/iwasm/compilation/aot_llvm_extra2.cpp>). It makes use of an LLVM API which is only available in C++, hence the wrappers to allow the functions to be called from C code:



The *Data Layout* is always available from the *Target Machine* so once this has been determined, we can use these functions. For example:



#### Using the Address Space

The *wamrc* code saves the pointer size for the target into its *AOTCompContext* structure instance that is passed to all the compilation functions.

Previously, the pointer size was set like this in *aot\_llvm.c* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/core/iwasm/compilation/aot_llvm.c>):



This API function internally sets address space to be equal to 0 and then passes this on to *LLVMPointerSizeForAS(data\_ref, AS)*.

So now, a different API call is used which passes in the address space:



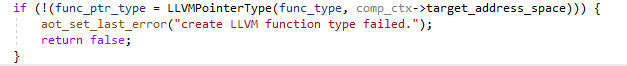
The new version always works (on Morello or otherwise), since if an AS has not been explicitly specified then it falls back to the previous version. This code would, in fact, eliminate errors if any other processor is using a non-zero AS.

Calculating the AS involves obtain the data layout from the target machine and then getting the AS – several lines of code. There are numerous functions in *wamrc* code which have an AS default argument that default to zero; all now need modifying. It is convenient to be able to simply pass an AS (AS is an integer value) directly into these functions without having to retrieve it from a separate function call or copy/pasted code everywhere.

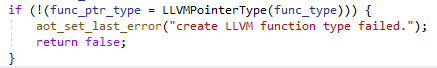
To this end, the AS is now set and stored in the *AOTCompContext* structure instance as this is already available to all the compilation code which needs it (e.g for GEP instruction generation). Below is the comp context structure updated:



It is now used in like this:



Which previously read like this:



The changes mean the code will now *always* work – for any target – and arguably *wamrc* developers should have implemented it with passing in the address space when they originally developed the codebase.

### Enforced Reduced Optimisation Pass

As discussed in Section 7.1.1.4, one of the optimization passes is to vectorize load/store operations.

However this step cannot be performed on Morello pure-cap, because this optimisation step mandates that the *pointer size* == *index size*. (There is [this LLVM commit](https://git.morello-project.org/morello/llvm-project/-/commit/945b7e5aa639353f5660415935e58601d01cf270) which explains why the optimisation will cause an assert if *pointer size != index size*).

The index size is the width of the indexing storage variable used in LLVM’s GEP instruction, which on Morello is 64 bits (versus a pointer size of 128 bits). This can be seen from the *Data Layout* string section:

**e-pf200:128:128:128:64**

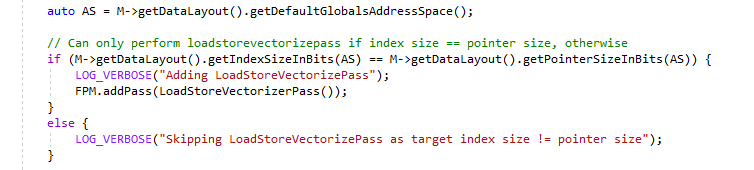
The last value, set to 64, is the index size. This applies only to the capabilities address space.

Note that this optimisation step is not applicable to AArch64 anyway, so there is no performance loss in not performing it.

#### Code Changes for CHERI port

To support the need to skip this optimisation pass code changes have been made which only apply whenever *pointer size != index size*. This is therefore another non-Morello-specific change to the code, but will prevent an assert on Morello pure-cap and any other potential platform having the (unusual case) of an index size being different from the pointer size.

To aid the user a log point is added to indicate whether this pass was skipped or not. Code changes to *aot\_apply\_llvm\_new\_pass\_manager()* in *aot\_llvm\_extra.cpp*:



***NOTE:*** *As mentioned previously, on AArch64 it leads to no worse performance if this optimising is not done as AArch64 implements its own optimisation step*.

## WAMR AOT Runtime Changes for CHERI

This section describes the changes which were needed to support CHERI Morello in WAMR for AOT mode.

This section covers **only** changes which are isolated to WAMR. System wide changes, affecting both compiler and runtime, are documented separately.

### Target Info support

As part of loaded AOT file verification WAMR checks the AOT *Target Info* section matches the actual runtime target. The fields are the same as in an ELF header.

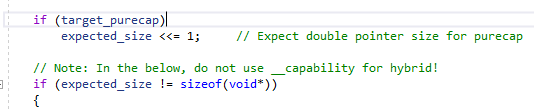
For Morello, the word size is always 64-bits but WAMR then assumes this is also the pointer size and therefore checks *expected\_pointer\_size == sizeof(void\*)* which on pure-cap will fail.

The solution is to note that CHERI Morello Pure-cap ELF files have the *eFlags* field set to 0x10000 to distinguish from “standard” AArch64. This flag is already being read by the AOT runtime, so we check it to establish if we are in pure-cap mode or not and if we are then we expect the pointer size of a capability pointer to be twice the machine word size. Now the pointer size check will pass on Morello pure-cap as well as Morello hybrid.

Code changes are shown below:



And then:

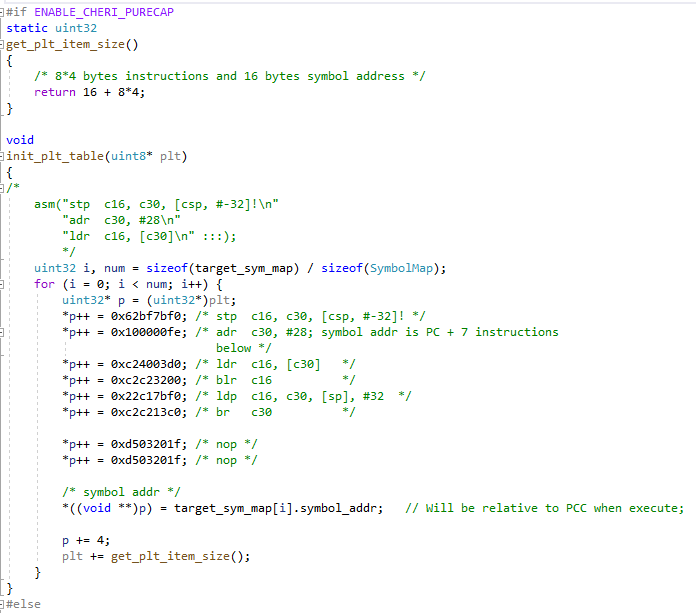


### Morello Pure-cap Procedure Linkage Table

WAMR builds the PLT table by generating a small bit of assembler for every address to be resolved - this is added to the code section read from the AOT file.

In the CHERI Morello pure-cap case, capabilities will be used for all symbol addresses and the assembler used to dynamically write the PLT must be written using C64 assembler opcodes. This is added to the *aot\_reloc\_aarch64.c* file and used if *ENABLE\_CHERI\_PURECAP=1* i.e WAMR was built for Morello pure-cap.

The PLT entry code is shown below; Morello assembly opcodes are generated:



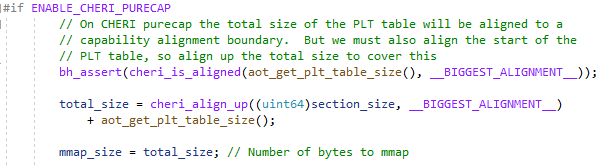
**NOTE:** The function *get\_plt\_item\_size()* is also modified for the pure-cap case to reflect the different size of the code region written for a PLT entry.

Note also the use of *nop* instructions to ensure the whole PLT entry block is a multiple of 16-bytes in size (and therefore all PLT entries will be capability-pointer-aligned).

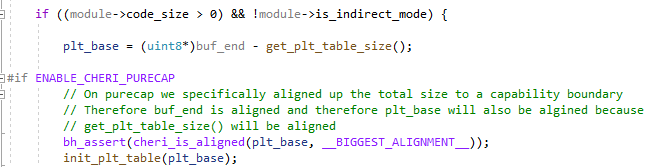
#### PLT Alignment

The PLT needs to be aligned to a capability-pointer boundary; 16-byte alignment. Additionally each PLT entry need to start on such a boundary.

As explained above, PLT entry alignment is achieved by making each entry an aligned size. To align the whole PLT, we simply *cheri\_align\_up()* the end of the *.text* area before then adding the PLT (and ensuring we *mmap()* enough space for the alignment in the first place):



* + In the above, *section\_size* is the size of the text section and so we align up to build in enough space for the alignment when placing the PLT



* + Here, *buf\_end* is the end of the area *mmap()*ed for code, which was previously aligned up
  + Therefore, and given the total PLT size must be an exact multiple of *\_\_BIGGEST\_ALIGNMENT\_\_,* then the start of the PLT (*plt\_base*) must also be aligned.
  + This is checked and if not the code asserts because that would be a coding error

### Morello Symbol Relocations

In Section 7.1.3.4 the relocation tables were explained, including the *relocation type*.

Many Morello relocation types are the same as defined for *AArch64*, but Morello does add quite a few new ones. A number of these are not applicable to WAMR because the AOT compiler’s parser will never generate code that needs them, but there are a few for which implementation had to be added to support WAMR AOT Mode.

These were done in *aot\_reloc\_aarch64.c* and as they only apply to Morello, they were surrounded by *#ifdef \_\_CHERI\_\_* directives.

The new types are:

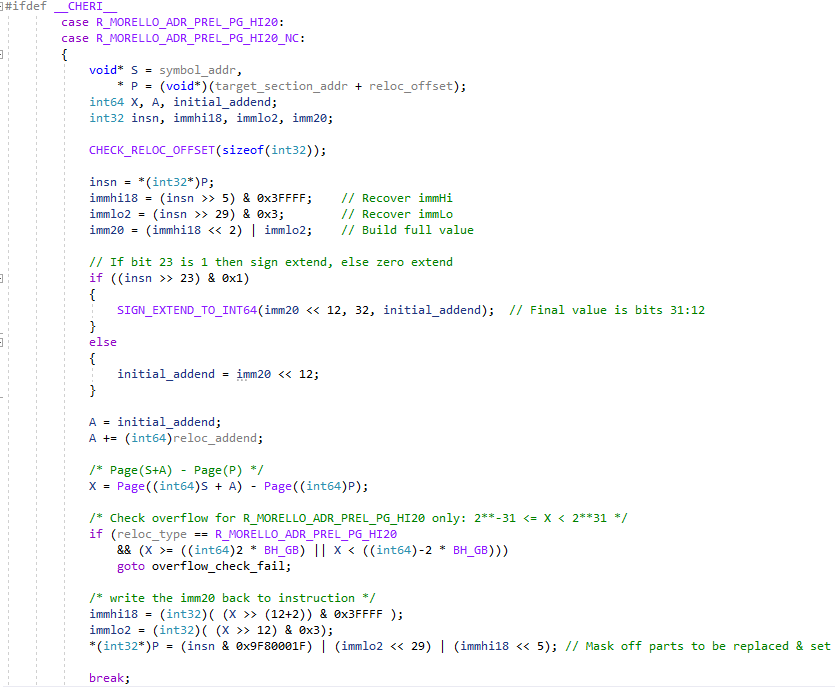
* *R\_MORELLO\_JUMP26*
* *R\_MORELLO\_CALL26*
* *R\_MORELLO\_ADR\_PREL\_PG\_HI20*
* *R\_MORELLO\_ADR\_PREL\_PG\_HI20\_INC*

***NOTE:*** *Arm provide a full explanation of the Morello relocation symbol additions,* [*here*](https://github.com/ARM-software/abi-aa/blob/main/aaelf64-morello/aaelf64-morello.rst)*.*

*R\_MORELLO\_JUMP26* and *R\_MORELLO\_CALL26* operate the same as the AArch64 counterparts *R\_AARCH64\_JUMP26* and *R\_AARCH64\_CALL26* and so they are handled by existing WAMR code for those AArch64 types.

*R\_MORELLO\_ADR\_PREL\_PG\_HI20* and *R\_MORELLO\_ADR\_PREL\_PG\_HI20*\_*INC* are similar to *R\_AARCH64\_ADR\_PREL\_PG\_HI21* and *R\_AARCH64\_ADR\_PREL\_PG\_HI21\_INC* but have a different length of address with a separate bit to indicate a positive or negative offset from the PC register (which is a PPC register on Morello).

Therefore new code is added to handle these new symbol relocation types, as shown below:



* + Note the sign extend or zero based on b23 in the instruction
  + Also note the logic to deal with the 20-bit immediate value (not 21-bit as for *AArch64)*

### Setting up the Code Region

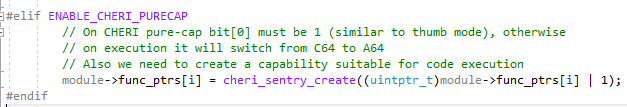
The code region for *.text* and the PLT tables is *mmap()*d. Initially this needs to be mapped as data and then later switched to code. As described in Section 4.2.7 it is necessary on CHERI to *mmap()* with full access protection to obtain a capability pointer with the correct positions, and then later use *mprotect()* to restrict the permissions (to READ | EXECUTE).

This is handled implicitly since we already modified *os\_mmap()* to behave this way.

Additionally, on CHERI in order to actually call the code we need to perform two steps:

* + Mask the LSB of the address (analogous to ARM v Thumb mode) in order to maintain C64 mode when the object code is called from native
  + Convert the capability to a *sealed entry* capability, with appropriate permissions

WAMR runtime maintains a pointer to the each function within the code – these are themselves capabilities derived from the *mmap()*d capability. When each function call data is loaded this is the place where we need to configure the capability correctly. The code to do this is as follows (in *aot\_loader.c*):



### Aligning Module Structure Buffers

AOT mode, like WASM Interpreter mode, makes use of the WAMR execution environment and regions of the *WASMModuleInstance* that are allocated as a byte buffer and used to contain different regions for tables, memory instance, extra data etc.

The problem with these (ensuring the areas are aligned) and the CHERI solution is fully explained in Section 4.2.3, and will not be repeated here. Unfortunately WAMR implements an amount of very similar code in both interpreter and AOT runtime instantiation functions, so the changes needed to support CHERI alignment had to be added to added to the AOT runtime code as well.

## AOT System Wide Changes for Morello

### WAMR Pure-cap PC Relative Addressing to Data Regions

This necessitated a change to the AOT File format, therefore it affected both the *wamrc* AOT compiler and the WAMR AOT runtime.

The generated AOT object code may include instructions such as *ADRP* which generate a PC-relative address (page) to an initialised data section that was also present in the AOT file (e.g *.rodata*).

As per any symbol address, this address is patched up via symbol relocation using the resolved location of the appropriate initialized data section which is wherever WAMR allocated it. Recall that WAMR *mmap()s* initialized data section to memory buffers somewhere within the program’s address space, and then copies the data from the AOT file buffer to this space. As these are within 128Mbytes of the code area then PC relative addressing can be used to obtain the location of the symbol.

#### Problem on CHERI Pure-Cap due to Capability Bounds

This presents a problem on CHERI pure-cap because all pointers are *bound capabilities*.

WAMR *mmap()*s the code region (*.text + PLT*) and the system will return a capability for this area, bound to the range of *region\_start 🡪 region\_start + requested\_size* (although due to *page alignment* to a 4kByte page, *requested\_size* may have been increased to the next page boundary).

Each initialized data area will also be *mmap()’d* somewhere, and this will also return a capability bound to that initialized data’s *mmap()’d* range.

Now there is a problem, because we need to derive a PC-relative address from the PPC to somewhere in the initialized data region. But the PPC capability will be derived from the code regions *mmap()* and therefore its bounds to not extend outside of the code region. Hence, at the point we attempt to access the PC-relative address, we will be out-of-bounds and a segmentation violation will occur.

This is illustrated below:

A diagram of a computer program

Description automatically generated

#### Solution to Capability Bounds Problem

The solution is obvious – as the system security prevents us extending a capability bounds to memory we have not been granted access to, then we need some way to derive a PCC relative address which can also cover the range of the data regions.

Another way of looking at this is that the original capability from which the bound PCC will be derived must somehow – magically! – include the data region as well.

But the CHERI security restricts us doing this. In lieu of some sort of “capability manager” running in the processor executive state, we need the system call to *mmap()* to return a capability which can cover future *mmap()*s too, or some other “magic capability” with almost unlimited bounds.

It turns out that, without something unbelievably complex that is beyond the scope of this work or which tries to defeat CHERI security (!), there are only really three possibilities:

1. Use the DDC as a capability source, since its bounds cover all of addressable memory
2. *mmap()* and reserve a massive region which can then be used to supply capabilities for future buffers we may need
3. Collate the code and initialized data regions together and *mmap()* them collectively

The DDC method sounds like a “short-term hack” and shouldn’t be done on Morello pure-cap. Reserving the region is possible but potentially uses resources that aren’t required; is wasteful and could be prone to error if we end up needing more memory.

The solution chosen was to *mmap()* the code and data regions together.

This did, though, present some problems which necessitated quite significant changes to the codebase, as described in the following sections.

#### Impact to XIP Mode

If *Execute In-Place (XIP)* is enabled as an AOT compiler option than WAMR does not *mmap()* the code buffer and instead executes it directly from its location in the AOT file buffer.

In this case there is no opportunity to utilise a single *mmap()* buffer for all code + initialized data, and the capability bounds of the capability accessing the AOT file buffer will be constrained to this buffer region.

Therefore XIP mode is now not permitted for AOT mode on Morello platforms. If XIP mode is detected when parsing the AOT file information section then WAMR will exit with a suitable error message to the user indicating to rebuild the AOT without XIP mode enabled.

#### Determining Combined Code and Initialized Data Size

To *mmap()* code and all the initialized data sections as a single, contiguous block then we either need to make some assumptions about the total size or calculate it accurately. Making assumptions is wasteful, especially since each region must start on a 4Kbyte page boundary, and code be open to problems if the size is not sufficient.

Calculating the total size of all initialized data is, though, difficult because of the way the AOT file is structured:

* + Text section is a different top-level part than initialized data
  + Initialized data part needs *full parsing* to obtain count of sections
  + Each section needs *parsing* to obtain its size

The way WAMR code currently works for these is like this:

1. Parse text section, obtain size and perform *mmap()* for it
2. Parse rest of file
3. If encounter a data section, keep parsing and setting up structures until we get size of an initialized data section, then we can *mmap()* a buffer for it

If we are to create a single *mmap()* then we need to know the *total* size of all initialized data at the point when the *text* code region is *mmap()’d*.

The further problem is the WAMR AOT runtime does not mandate an order of sections, therefore it is theoretically possible the data section may come before the text section (although at this point the sections are only *created* and it is only when the data section is *loaded* that we will encounter each size).

The WAMR code is just too complex to dramatically change the way it works in the time available, and also this would make it harder for changes to be accepted back into the *ByteCodeAlliance* WAMR project. Therefore the solutions would seem to have to be:

1. Perform a “pre-parse” of everything to collect the data sizes, then the “proper parse” later (a similar technique is done for WAMR Fast Interpreter mode)
2. Make some slight changes to the AOT file structure to enable the *total data size* to be known sooner in the process, so we can *mmap()* the correct amount when the code buffer needs creating

The first option was rejected as AOT is supported to be high performance and so it would be detrimental to this to have to perform intensive parsing twice – it also makes the code even more complex.

The second solution was therefore chosen for this project.

#### WAMR Modifications Required

The changes regrettably necessitate a breaking change to the AOT file format.

* The count of initialized data sections is moved to the **top** of the *Initialized Data Section Info data* in the AOT file
  + Currently, this is buried deep within this section and due to *TLV* format, only available by parsing other fields
* A new data field is added to the **top** of the *Initialized Data Section Info data* in the AOT file, which is the *total* size of all the initialized data
  + This is the sum of individual sizes; individual sizes are still found inside the section data, unchanged

**NOTE:** The reason “*initialized datas” count* is needed is so we can allocate enough additional space to be able to align up the start of each initialized data block – the additional space is *sizeof(4K\_PAGE) x count*.

With reference to Section 7.1.2 the changes to the AOT file structure are shown below (original on left, changed on right):

A diagram of a data processing process

Description automatically generated with medium confidence

The modified WAMR AOT loader algorithm to *mmap()* the space is then as follows:

1. At the time when the loader is parsing each top-level AOT section (during *create\_sections())*, when the *initialized data* section is discovered then pre-read (but do not remove) the new “*initialized datas” count* and *total initialized data size* fields
2. In *create\_sections()*,as it cannot be guaranteed whether the *text* section or the *initialized data* section will be parsed first, save details when both are encountered and delay the *mmap()* until the end of the function
3. When the code size + location in AOT file are known *and* the initialized data size + count is known then perform the *mmap()*. The size required is:
   1. Code size +
   2. PLT size +
   3. Total data size +
   4. *sizeof(4K\_PAGE) x* count of init datas (for alignment purposes)

When the initialized data blocks are then parsed, if we already allocated space for the initialized datas then we need to use this space instead of creating a new *mmap()* buffer.

This is done by maintaining a “current” pointer into the *mmap()’d* region. We align-up the pointer to a page boundary and then use the space, before moving the “current” pointer on by the space used.

There is also the opportunity for a “sanity check” here – the total of all the initialized data individual sizes should be exactly equal to the *total size* field at the start of the block.

The resulting use of the *mmap()* area is shown below:

A diagram of a diagram

Description automatically generated

*When Changes are and are Not Required*

This changed operation modifies the AOT file format and therefore the code changes must be made generically in WAMR, not just for Morello pure-cap mode.

However there are times when it is unnecessary to *mmap()* code and data regions together. It is only required on Morello pure-cap, and only when there is both code *and* initialized data. (In theory, the relocation tables could be checked and this step only performed when there is code + initialized data + a relocation patch-up of a data region in object code. This is, however, far too complex to implement for such little gain).

If there are no initialized data regions, then there cannot be a symbol relocation to them.

Equally, if there were no object code then there cannot be a symbol relocation which gives a PC-relative address (although in practice it would be rather odd to have no executable code the software design does permit it!)

To support this, the code changes implement a flag to state whether the code *mmap()’d* region also includes space for initialized data areas. If not, then the existing behaviour occurs which is to *mmap()* each data buffer separately.

#### WAMR Code Changes

The *aot\_loader.c* function *create\_sections()* builds the section list by reading & parsing the *Type* and *Length* that all AOT file sections begin with. If the section is the *Text* section, then this is code and so the *mmap()* of the code buffer is done here.

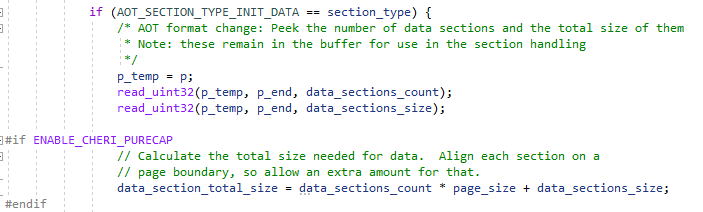
The first change to this section is to additionally detect the *initialized data* section and if it is detected, *peek* (but not read) the new data fields from the section’s *Value* (i.e its info data):

* Number of data sections
* Total size of all used data

We do not read and consume these as we do not want to corrupt the calculations of other parts of the codebase which rely on the section size.

The total size required is then calculated, incorporating the page size for alignment purposes**.**

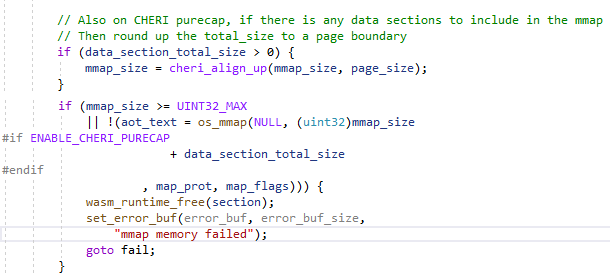
The code to do this is shown below:



We now need to know the code size (and location) for Morello pure-cap in order to *mmap()* the whole buffer. But as the sections could (theoretically) occur in any order, we cannot do the calculation until we know both of those values.

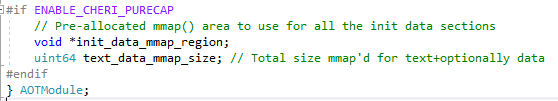
We therefore save the *data\_section\_total\_size* as shown above, and also save a pointer to the *Text* section when we detect it. After parsing through all sections we then have both of these bits of information and so can then safely do the *mmap()* at the end of the function.

In Morello pure-cap, we will need to include the total data size (unless it is 0, in which case there was no initialized data). First it is rounded up so the total size is an exact multiple of the AArch64 page size and then we incorporate the data size in the *mmap()* (if it is 0 then this has no effect):

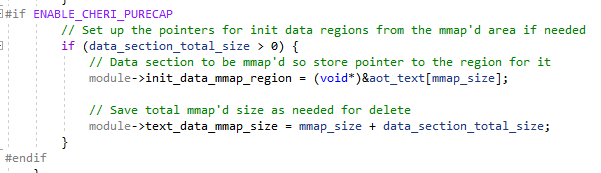


The final part of this function is to setup a pointer to the area which is to be used for the initialized data regions. This capability, derived from the *mmap()* above, will have suitable bounds.

This pointer is one of two new fields added to the *AOTModule* structure to make it available elsewhere. The second is the total size of the *mmap()’d* area. This is *different* to what WAMR stores currently – WAMR explicitly needs the *code* size for many purposes, one of which is to *munmap()* the region. But we now need the actual *mmap()’d* size as well, hence we add the two new values to the module structure (for CHERI pure-cap):



They are both set after doing the *mmap()* at the bottom of *create\_sections()*:

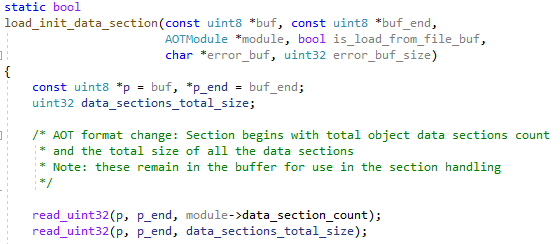


***NOTE:*** *mmap\_size is the size of the code region ONLY, aligned up to a page as needed (which the os\_mmap() WAMR wrapper to mmap() will do anyway)*. *This is therefore the place to put the first init data block.*

*Using the mmap() area for Initialized Data*

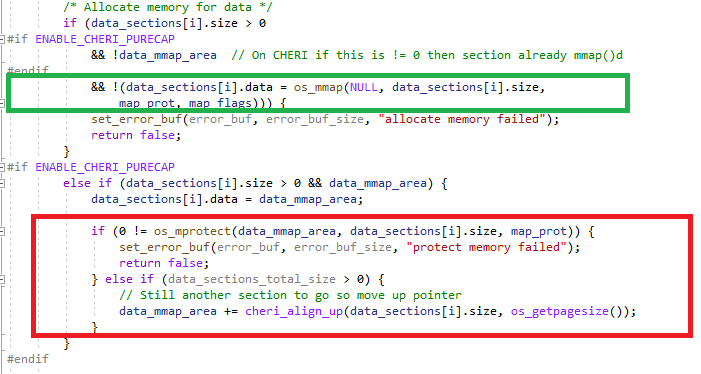
If *AOTModule.init\_data\_mmap\_region* is NULL then no initialized data area was already mapped, otherwise it should be used. The data regions are set up in *load\_init\_data\_sections()* also in *aot\_loader.c*.

The first change is that there are the two fields to read & consume from the beginning of the init data section – this file format change of course applies to **all** WAMR build targets:



The *data\_sections\_total\_size* was technically only needed for the calculation in *create\_sections()* so it could be thrown away. However as this is effectively duplicate information (given each data section will include its individual size) we can use it as a “sanity checksum” to ensure that *data\_sections\_total\_size == sum(data\_section\_individual\_size for all sections)*. The *data\_section\_count* is needed later on in the processing and is retained anyway in the module data.

The memory allocation for each initialized data section happens in *load\_object\_data\_sections()* in *aot\_loader.c*. The modified part is as follows:

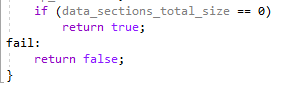


* + The green block is the *mmap()* which is used when no additional data memory was reserved during code buffer *mmap()*
  + The red block uses the existing *mmap()’d* space

In the case when buffer space has already been allocated, *data\_mmap\_area* points to the correct location. We therefore set up the pointer to use this this space and *mprotect()*  to set needed protections.

The *data\_mmap\_area* can then be moved on to point to the next region, assuming space remains. We first ensure we align up to a memory page so that the next data area will start on a page boundary.

Note the use of *data\_sections\_total\_size*. This is the “sanity check” mentioned above. After each init data section is processed, we subtract the actual used size from it. If the AOT file was formatted correctly then after processing all init data sections then the remainder should be zero. If it isn’t, the function returns an error:



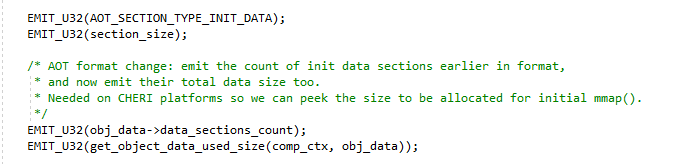
#### AOT Compiler Code Changes

As stated, this is a breaking change to the AOT file format and therefore *wamrc* needs to be modified to emit the updated AOT file.

The AOT file change is restricted to emitting the initialized data part, as follows:

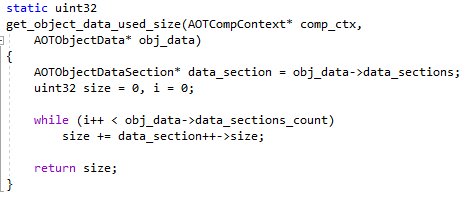
* Data Sections Count is *moved* – now is the first value of the section *data* (32-bit in size)
* Total data size is *new* – is now emitted straight after data sections count (also 32-bit in size)

This change is shown below, at the start of *aot\_emit\_init\_data\_section()* in *aot\_emit\_aot\_file.c* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/core/iwasm/compilation/aot_emit_aot_file.c>):



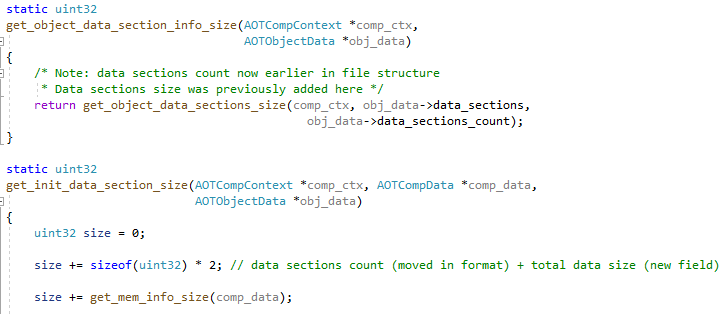
* + As for any section, the *Type* then the *Length* is emitted
  + Then the *Data Sections Count* which is read directly from an internal structure
  + Then the new *Total data used size* which is calculated

The *wamrc* code already includes a calculation to sum the total size of all init data *internal sections* but these include header preambles and therefore a new function is required to just total up the actual ELF section sizes themselves:

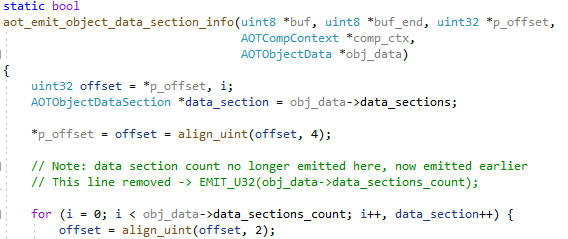


This function, also in *aot\_emit\_aot\_file.c*, can be compared with the existing *get\_object\_data\_sections\_size().*

*Wamrc* calculates the total section size by summing the individual parts. The calculation has now changed slightly as we have added one 32-bit value to the top level and also moved another 32-bit value to a higher internal level. The changes are shown below:



The only remaining change is to remove the emitting of the *data sections count* field because it has already been emitted as shown above:



### Structure Layout Equivalence of WAMR and wamrc on Morello

This problem came about as a result of building for a target with 128-bit pointers on a 64-bit architecture. It is caused by the fact that the layout of structures will then differ between AOT compile time and AOT runtime.

Unfortunately for some structures this does have an effect on the system, because native code (such as the target object code compiled from WASM) is able to access the *WAMR execution environment* and *WASM module data*.

When the WASM is converted to LLVM IR, fields within the *WASMModule* structure are made accessible to the IR by generating LLVM GEP instructions which index into module structures at particular locations. However, generation of these instructions is based on a 64-bit architecture as the host will be running Linux x86\_64.

On Morello pure-cap this is then a problem at runtime, because WAMR is then built on a 64-bit platform but with 128-bit pointers and 128-bit pointer alignment, not 64-bit. Given that many structures contain pointers, the result is a particular field will be aligned differently between the runtime environment and the compile time environment.

WAMR developers were clearly aware of this, and have taken two steps to mitigate against it:

* All pointers are padded to 64-bit, if the machine is a 32-bit machine
* A number of compile-time *static asserts* will assert when building WAMR if a structure offset does not match the expected value
* There is a clear note in the code comments that the data layouts must match the on both compile side and run side

Unfortunately this is not sufficient for Morello pure-cap, which is as stated a 64-bit machine but with capability pointers. Therefore two problems need solving:

* Enforcing matching alignments on both *wamrc* and WAMR, despite the different architectures the programs are running on
* Ensure that the changes do not affect other target machines *wamrc* can build for, and WAMR can run on
  + **This is the especially difficult part!**

The solutions to these are described below. Unfortunately, without a great deal of work and switching to an object-oriented style dynamic type, it was not possible to avoid a build-time preprocessor flag to *wamrc* to specify the *maximum* size of a pointer on any version of WAMR which will be supported. This was a compromise: although target information is now being provided at *wamrc* build-time, it is still possible to compile AOT for any target without needing to rebuild *wamrc*.

#### New Build Flag

The *WAMR\_BUILD\_AOT\_CHERI\_PTR* flag is introduced, which is used both for *wamrc* and WAMR building. This should be set to the byte size of a pointer on the supported target which is the largest of all supported targets on **all** platforms *apart from* the target with the maximum pointer size *unless* this is the same as the size of the pointer on the machine where *wamrc* runs.

The default value for *WAMR\_BUILD\_AOT\_CHERI\_PTR* flag is zero, in which case it has no effect.

This is because *wamrc* supports multiple compile-for targets, and WAMR is built to be target specific.

The below table shows the settings for this flag in the case that Morello pure-cap is to be a supported target, and if it is not. “Supported target” means that we want to be able to use *wamrc* to compile AOT for that target, and we will be building a WAMR (*iwasm*) on that target.

| Morello Pure-cap supported? | *wamrc* Build  on x86\_64 | WAMR x86\_64 | WAMR Morello Hybrid-cap | WAMR Morello pure-cap |
| --- | --- | --- | --- | --- |
| Yes | 16 | 16 | 16 | 0 |
| No | 0 | 0 | 0 | 0 |

Explanation:

* If Morello pure-cap is to be supported then pure-cap has the maximum pointer size (16 bytes) and this is bigger than x86\_64 at 8 bytes. Therefore *WAMR\_BUILD\_AOT\_CHERI\_PTR=16* on all not Morello pure-cap platforms
* If Morello pure-cap is not to be supported then the maximum pointer size is 8 bytes (on x86\_64, Morello hybrid) and this is not larger than the *wamrc* build machine and so *WAMR\_BUILD\_AOT\_CHERI\_PTR=0*

**NOTE:** In the case that the value of *WAMR\_BUILD\_AOT\_CHERI\_PTR* is the same as the size of the pointer on the machine the build is taking place on then it has no effect, but it will introduce more complexity (and potential for errors) therefore recommended to set it to zero in this case

#### How it Works

Any pointers in structures which are used by both the AOT compiler and accessed by the AOT object code at runtime (in WAMR) must be aligned equally, i.e the offsets of fields in the structures **must** be identical. If this does not happen, there will be hard-to-trace errors in WAMR at runtime and likely *Segmentation Violation* errors especially on the capability-backed Morello pure-cap.

Suppose we are building WAMR on Morello pure-cap. Any pointers in a structure will be 16 bytes in size, and they will be aligned on a 16-byte boundary. The compiler will insert pads as needed, consider the below code and the comments which explain the offsets:

**struct Bar {**

**uint32 i; // Offsetof(struct Bar, i) == 0**

**/\* Compiler inserts pad of 12 bytes here \*/**

**void \*p; // Offsetof(struct Bar, p) == 16**

**/\* sizeof(p) == 16**

**uint8 j; // Offsetof(struct Bar, j) == 32**

**};**

Now consider the same structure used in *wamrc* which runs on a Linux x86\_64 machine. In order to achieve the same layout we need to insert pad bytes to maintain a suitable alignment and pad out the size, to get the same offsets. Here’s how we could implement the structure above to have the same layout as on Morello pure-cap:

**struct Bar {**

**uint32 i; // Offsetof(struct Bar, i) == 0**

**uint8 pad1[12]; /\* Achieve 12 byte pad here \*/**

**void \*p; // Offsetof(struct Bar, p) == 16**

**uint8 pad2[8]; /\* sizeof(p + pad) == 16**

**uint8 j; // Offsetof(struct Bar, j) == 32**

**};**

We have now solved the problem by ensuring that *wamrc* has suitable pads to line-up with how the structure is naturally laid out on Morello. But now suppose we use this version of *wamrc* to compile an AOT file which we wish to run on Morello hybrid-cap. Suddenly there would be another problem, because Morello hybrid-cap requires a structure without the pads as its pointer size is 8 bytes.

As we are defining the structure layouts at wamrc *build* time we cannot dynamically modify them at wamrc *run* time when we know the actual *Target Machine* we have selected. Therefore to continue using this simple build-time modification the only option is to insert the pads on *all* 64-bit-pointer-size build targets. Clearly the pads aren’t to be used on Morello pure-cap as that is already laid out ok.

The problem with this approach is that in the example shown the pads were hard-coded. But now imagine we ran *wamrc* on a 32-bit machine, or that one of *wamrc’s* supported targets was a 32-bit machine. We’d then need to change the pad values to get the right fit on those machines. This sounds like a lot of *#if TARGET\_X #else TARGET\_Y #endif* constructs because the same physical source code file is used on all targets.

A better approach is to calculate everything from the desired alignment / pointer-size in the preprocessor since this knows how to calculate the size and alignments. So, to align everything to the 16-byte Morello pure-cap pointer size, our structure can be written like this:

**#define DESIRED\_SIZE 16**

**struct Bar {**

**uint32 i; // Offsetof(struct Bar, i) == 0**

**void \*p \_\_attribute(aligned((DESIRED\_SIZE)));**

**// Offsetof(struct Bae, p) == 16**

**uint8 pad[DESIRED\_SIZE – sizeof(void\*)];**

**uint8 j; // Offsetof(struct Bar, j) == 32**

**};**

This forces the pointer to be aligned to the required size, and the pad after the pointer is automatically set to the correct number of bytes. Using the *DESIRED\_SIZE* macro means we just need to change one value to back the code work if we decide we need a different size.

We can see that so long as *DESIRED\_SIZE* is set to the size of the pointer on the biggest supported platform, and the above code is used on all platforms, then all will work.

However on the biggest-sized-pointer platform the alignment and pads are not required. Although they will still work, it is additional complexity and preprocessor macros that are superfluous. Therefore if there was an easy way of doing it, we could set *DESIRED\_SIZE* to 0 and detect this as a special case and eliminate the alignments and pads on that platform.

**NOTE:** We must ensure that *DESIRED\_SIZE* is always set to the maximum of all supported platforms otherwise there will be a build error if the pointer size > *DESIRED\_SIZE*.

*Making the size configurable*

One option could be to hard code the *DESIRED\_SIZE* macro in the codebase to always be 16. However this is potentially wasteful, as if the maximum pointer size on all supported targets is only 8 bytes then we will waste up to 16 bytes of memory for every structure field that is a pointer and add extra complexity for no reason.

By making the size configurable from the build options, it always the user to determine what needs be done.

*WAMR\_BUILD\_AOT\_CHERI\_ PTR*

For WAMR and *wamrc* the flag *WAMR\_BUILD\_AOT\_CHERI\_PTR* is used for what is being called *DESIRED\_SIZE*, above. If this is set to 0 then the code will fall back to how it would be without the alignment / pad constructs shown.

This ability to use 0 to “remove” the code is useful not only to reduce unnecessary preprocessor operations. It additionally:

* Means we can remove the code entirely if AOT mode is not supported on WAMR (without AOT mode none of this stuff is needed as there is no *wamrc* to worry about)
* Means the code changes can be built out entirely, making the code fall back to the original *ByteCodeAlliance* layout
  + This may make it easier if the *verifoxx-cheri-wamr* changes will be accepted into the *ByteCodeAlliance* project in the future

#### Applicability to WAMR and wamrc Code

The mechanism described above must be applied to the *WASMModuleInstance* structure and all the other structures it contains, wherever a pointer is in use. WAMR developers already have a *DefPointer* macro to wrap all pointers in the structures which add a 32-bit pad if the machine word is 32-bits and not 64-bits.

However this is not sufficient for Morello pure-cap because it does not consider a machine can be 64-bit but have a 128-bit pointer size, and that a pointer may need alignment different to the natural machine word alignment.

We therefore introduce a modification to this *DefPointer* macro which applies whenever AOT support is enabled (*WAMR\_BUILD\_AOT=1*) **and** when *WAMR\_BUILD\_AOT\_CHERI\_PTR != 0*:

**#if <modification\_needed>**

**#define DefPointer(type, field) \**

**type field \_\_attribute\_\_(aligned((REQUIRED\_SIZE)); \**

**uint8 field##\_padding[REQUIRED\_SIZE-sizeof(void\*)];**

**#else**

**#define DefPointer(type, field) \**

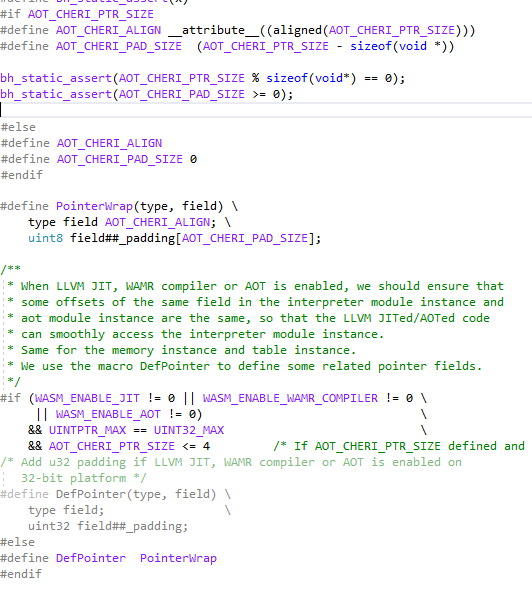
**type field; \**

**uint32 field##\_padding;**

**#endif**

That is not exactly how it looks in the code, though, due to the need to factor in the different build flags (only when AOT mode supported and WAMR\_BUILD\_AOT\_CHERI\_PTR !=0) and also we do include static asserts to check the WAMR\_BUILD\_AOT\_CHERI\_PTR size is a multiple of the actual pointer size on the build machine and that the logic has worked.

The full code snippet is shown below:



**NOTE:** *AOT\_CHERI\_PTR\_SIZE is the C code equivalent of WAMR\_BUILD\_AOT\_CHERI\_PTR CMake* flag

The code does the following:

* Defines *AOT\_CHERI\_ALIGN* and *AOT\_CHERI\_PAD\_SIZE* to simply the other macros (in the same way as shown with *DEFINE\_SIZE* above)
  + Note this is only done if *AOT\_CHERI\_PTR\_SIZE != 0*
* Checks for user supplied errors with *AOT\_CHERI\_PTR\_SIZE* (i.e that it is not less than the machine pointer size, and is a multiple of the pointer size)
* Defines the *PointerWrap* macro which will be used for *DefPointer* 
  + If *AOT\_CHERI\_PTR\_SIZE* was 0 then this falls back to the original definition of *DefPointer*
* Uses the new *PointerWrap* for *DefPointer* **unless** we are running on a 32-bit machine **and** the desired pointer size is not larger than 32-bits
  + This is the original WAMR code left included, so this behaviour is unchanged

**NOTE:** In the above code snippet there is no check for not including the code when AOT mode is not being build *(WAMR\_BUILD\_AOT = 0*). This is handled in the *CMake* build files; if AOT is not enabled or *WAMR\_BUILD\_AOT\_CHERI\_PTR* is not defined then *AOT\_CHERI\_PTR\_SIZE=0* is set in the source code.

#### Build-time Structure Checks

WAMR already features a number of *static assert* checks on various fields in the *WASMModuleInstance* structure, to ensure that the offsets line up to what is expected. This was included to catch misalignments on WAMR on the basis that the pointer size would always be 64-bit (or padded as such).

This *only* applies on WAMR and of course on WAMR we know the target machine, because WAMR is built differently for different machines. Further, we know that in the original codebase WAMR developers were forcing a pad of all pointers to be 64-bit aligned (there was the 32-bit pad on 32-bit machines).

This means only the following are applicable:

| Setting | Check |
| --- | --- |
| CHERI Pure-cap Enabled | Pointers == 16 bits natively, so check offsets using *sizeof(uintptr\_t)* |
| CHERI Pure-cap not enabled and AOT\_CHERI\_PTR\_SIZE == 16 | Pointers *padded* to 16 bits – we can use 2 \* s*izeof(uint64)* as the expected size of a padded pointer |
| CHERI Pure-cap not enabled and AOT\_CHERI\_PTR\_SIZE <= 8 | WAMR original *ByteCodeAlliance* will pad 32-bit pointers to 8-bytes, so the minimum space used by a pointer will be 8 bytes.  In this case:   * we are padding to 8-byte pointers (AOT\_CHERI\_PTR\_SIZE == 8) * or 0 < AOT\_CHERI\_PTR\_SIZE <= 4 in which case WAMR original code will force a pad to 8-bytes space for a pointer * or AOT\_CHERI\_PTR\_SIZE == 0 because it is not in use, which falls back to the original WAMR code which is 8-bytes space for a pointer   In summary, if this condition is met then we can calculate pointer offsets as *sizeof(uint64)* |
| Any other case | This is not supported and is a compile-time error.  Note that this means it is not possible to have a pointer size > 16-bytes.  At this time, the system cannot support a build target which has a pointer size of > 128-bits. |

The static assert checks can be found at the top of *aot\_runtime.c* (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/core/iwasm/aot/aot_runtime.c>).

#### Aiding the User - Warning Message

To use the new *WAMR\_BUILD\_AOT\_CHERI\_PTR* concept the user must set a *CMake* build flag correctly on *wamrc* and also WAMR.

To aid the user, at *wamrc* runtime the target machine pointer size is checked and if *WAMR\_BUILD\_AOT\_CHERI\_PTR != 0* and also *WAMR\_BUILD\_AOT\_CHERI\_PTR != <target\_machine\_pointer\_size>* then a warning message is output to the user that they should ensure they rebuild WAMR with *WAMR\_BUILD\_AOT\_CHERI\_PTR* set correctly.

### AOT File Binary Version

Due to the changes described above, which mandate a modified AOT file format and a rebuild of WAMR for any build platform, as part of this project work the AOT Binary version number (*AOT\_CURRENT\_VERSION)* was incremented from 3 to 4.

This will at least prevent any AOT built before the *wamrc* changes from being accepted by a modified WAMR program.

## WAMR AOT Bug Workaround

During AOT Testing an issue was found which turned out to be a bug in WAMR that is unrelated to CHERI modifications. It happens to affect Morello because of a build option, but equally could affect other existing WAMR build targets as well. The bug was found on *ByteCodeAlliance* most recent release and on *main* branch as of circa May 2023.

A ticket has been raised in Verifoxx’s issues repository to cover this problem, and it should be addressed with WAMR at some point.

To enable AOT development and testing to continue the bug was worked around, and a build flag used to include or exclude this workaround.

The workaround is described in this section as although not CHERI specific it is work that was done as part of the project which was essential to be able to verify AOT porting was successfully completed.

#### Problem Explanation

By default on Linux platforms WAMR builds with *WAMR\_DISABLE\_HW\_BOUND\_CHECK == 0.* This causes a guard region to be provided around all stack and heap memory usage, and if access is made to these regions then a signal is generated which WAMR will catch internally and process to safely handle the memory access error.

On CHERI platforms this feature is not required because the capabilities perform a HW based guard check; it is therefore pointless to try and port this feature to CHERI Morello architecture. Some other WAMR supported build platforms also do not have this feature enabled, and as per WAMR documentation the feature can optionally be enabled or disabled at build-time.

However, WAMR uses this feature as a seeming “back-door” mechanism to cleanly exit AOT file processing when the WASI API function *wasi\_exit()* is encountered (which calls the native *was\_proc\_exit()* in WAMR code). But when *WAMR\_DISABLE\_HW\_BOUND\_CHECK!= 0* then this mechanism is unavailable, and it causes AOT processing to hit unreachable code, which generates an exception within WAMR.

#### Understanding WASI Exit Function in AOT Mode

When compiling to WASM with WASI-SDK support enabled then code is generated which makes use of the *wasi\_proc\_exit()* function. This is designed so that WASM code can return a value to the calling system (e.g Javascript, or in our WAMR case the Linux OS) in the same way that a C *main()* function can return a value as a system exit code.

When *wasi\_exit()* is called in WASM code it should not return; instead the WASM application should terminate and return the exit code passed to *wasi\_proc\_exit()*. Therefore in WASI generated WASM you would expect to see “unreachable”after a *wasi\_exit()* call.

When running WASM in Interpreter mode WAMR handles *wasi\_proc\_exit()* by generating a special exception whenever it occurs. This causes WAMR to exit the processing loop and hence return from the calling function (which may be the entry function). WAMR then clears the special *wasi-proc-exit* exception, and sets the return code to the value passed to the *wasi\_proc\_exit()* function.

In AOT mode, the same exception handling mechanism is used however WAMR cannot just exit the function processing since this processing is occurring within the compiled object code (i.e in native code).

Instead, WAMR makes use of the HW Bound Check Guard mechanism - the exception handler makes a forced, intentional write to memory in the guard region. This will therefore generate a segmentation violation signal.

WAMR catches this signal, and then uses libc’s *setjmp()* and *longjmp()* to set execution to an address which is saved before the call into the native code that implements the WASM function. The result is to effectively terminate execution and unroll the call stack to the point that the function was called - but with an exception being set. As per interpreter mode, if the exception was just the *wasi-proc-exit* pseudo-exception then the “error” is cleared and the program exit code is set accordingly.

#### Impact on CHERI

On platforms which disable the HW Bound Check, i.e *WAMR\_DISABLE\_HW\_BOUND\_CHECK=1,* none of the signal handling / jump code is compiled. In any case, it is likely this would not work on pure-cap without extensive modification as out-of-bounds access would trigger a HW fault and there would likely be other issues manipulating capabilities.

Nonetheless, as stated this is a generic issue with WAMR that is unrelated to CHERI porting.

#### Workaround Implementation

The workaround will be compiled into WAMR whenever the following *CMake* build flag is set (e.g in *CMakePresets.json*):

*WAMR\_BUILD\_AOT\_EXCEPTION\_WORKAROUND=1*

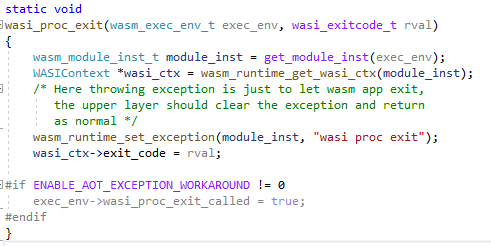
By default, the workaround is excluded.

The workaround implements the following:

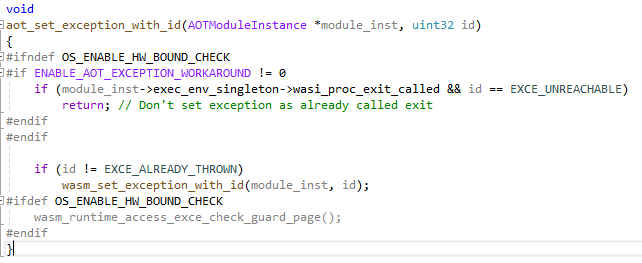
1. If *wasi\_proc\_exit()* is called then set a runtime flag
2. If an AOT exception occurs, and both of the following are true:
   * The exception type is *unreachable* code
   * The *wasi\_proc\_exit()* flag was previously set, indicating *wasi\_proc\_exit*() had been called already
3. … then ignore the exception and continue processing
   * This will return exception from WAMR to the native code

The result of this is that the system behaves as if *wasi\_proc\_exit()* was the last function call made before the object code returns, and the *unreachable* code never happened. This causes WAMR to complete successfully and return the correct system exit code as set in *wasi\_proc\_exit()*.

The code in *wasi\_proc\_exit()* inside *WASI-libc* code can be seen here setting the flag:



And ignoring the error in the exception handler for unreachable code:



## AOT Mode Testing & Validation

At the point AOT work was started there were a significant number of WASM test cases which were being used for WAMR in Interpreter Mode. These were being run manually and, where possible, automatically with the *WASI Test Suite* (Section 6.4).

To support AOT testing, WASM files were converted to AOT files using the *wamrc* version which was completed as per the work described in this section. The AOT files were then transferred to Morello to run WAMR in AOT mode.

Initially the simple WASM files were converted and validated for AOT mode, and then as development progressed the more complex versions were tried until it AOT mode was able to process complex applications such as the *TLS Client / Server* and the *HTTPS Client* test file.

The *wamrc* changes affected all platforms, so significant time was spent verifying that AOT could be successfully generated for Linux x86\_64 targets and run with a WAMR built for this platform, as well as for Morello in hybrid mode.

To verify the AOT file changes were correct it was attempted to run a new AOT file on an existing version of WAMR, and an old AOT file on a new version of WAMR. In both cases, the file was rejected as expected.

A number of tests were also made of building *wamrc* andWAMR with and without compatible versios of *WAMR\_BUILD\_AOT\_CHERI\_PTR* being set. It was confirmed the behaviour was as expected (basically a segmentation fault on WAMR if there is a mis-match), as was attempting to run an AOT file built for one target on a different target.

### AOT Mode Automated Testing

Section 6.4.2 introduced the *WASI Test Suite* adapter file that was created to support running WAMR remotely on Morello and the test environment on a local machine.

With reference to that Section and also to Section 8, which describes the automated test setup in some detail, the steps taken to extend this to support AOT testing is described below.

When testing AOT mode the AOT file can be thought of as an intermediate file, because it is always derived from the WASM file which is the actual test vector. Therefore, for an AOT test case there is an additional step:

* Use *wamrc* to generate an AOT file from the WASM file

And then:

* Run WAMR as existing, but use the AOT file as the input instead of the WASM file

This means that *wamrc* needs to run as its own process first and then the AOT file transferred to the remote Morello machine along with the other test data folders.

Running *wamrc* can be done locally, but for flexibility the adapter was created so that *wamrc* can if desired also be run on a remote machine. This was used within this project whereby the local machine was a Windows 10 PC and the *wamrc* remote machine was a WSL2 VM on the PC. To run the *wamrc* remotely involves similar steps to running WAMR remotely on Morello:

* + Create SSH connection
  + Put WASM file to remote
  + Run *wamrc* and collate output
  + Get AOT file back to local machine, temp folder. Then send to Morello.

If running on a local machine the external *wamrc* process just needs to run locally, and the AOT file written to a temp folder ready for transfer to Morello.

The flow for the AOT flavour of the CHERI adapter file for *WASI Test Suite* is shown below. It can be seen it is almost the same as the non-AOT version but with the additional *wamrc* step:

A diagram of a process

Description automatically generated

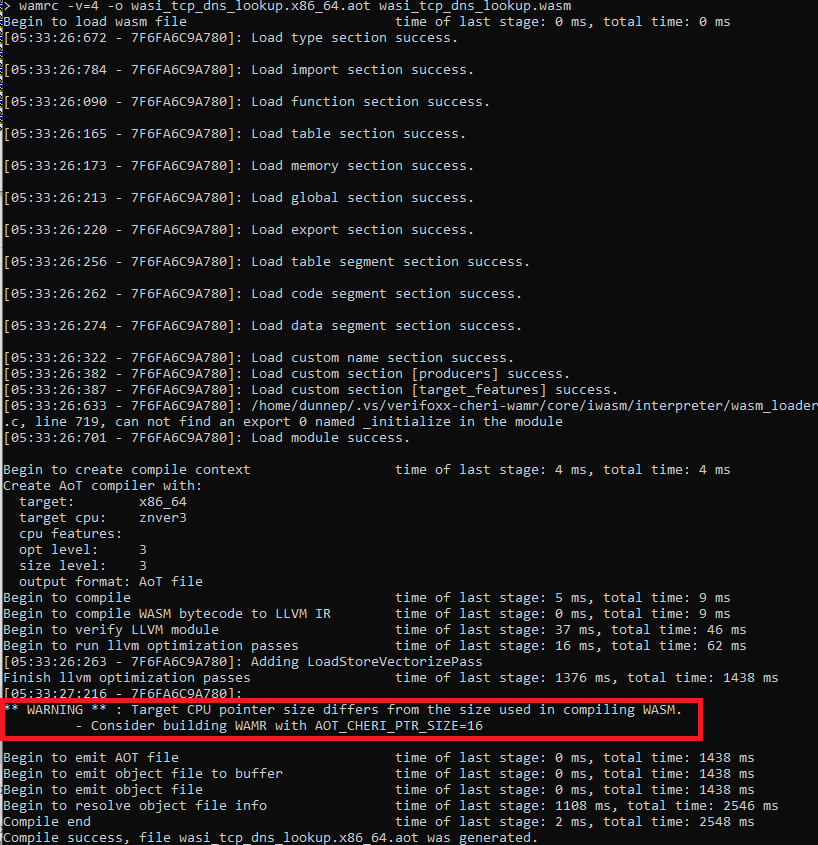
### Example Outputs

#### AOT Compilation

Below is shown compiling WASM to AOT for Morello pure-cap. Key command-line arguments are highlighted, along with the omission of load/store vectorization which is output to aid the user (reference Section 7.2.4 for more on this):



Compiling to default host (where *wamrc* is running), which is x86\_64, is shown below. Note the warning to the user that the target does not match the data layout (highlighted with red box), because pointer pads are set for 16-byte pointers but the target is 8-byte pointers:



#### WAMR AOT Runtime

Below shows running an AOT file with WAMR built for pure-cap on Morello. This particular example resolves a domain name to IP address and then connects on HTTP socket and *GET*s the server output:



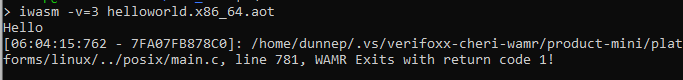
Attempting to run an AOT compiled for Morello hybrid-cap as Morello pure-cap will cause an error to be reported by WAMR because the ABI has a different pointer size (AOT has 64-bits, runtime has 128-bits):



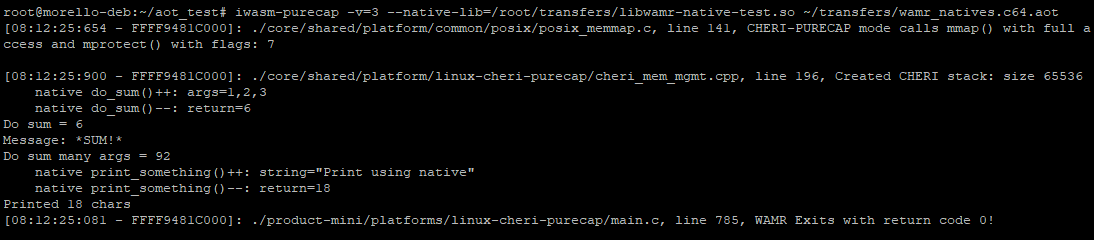
Attempting to run AOT compiled for x86\_64 on Morello hybrid will fail due to incorrect target type, even though the pointer sizes are the same:



So long as we use a WAMR on x86\_64 which was built with the *AOT\_CHERI\_PTR\_SIZE=16* then our *wamrc* compiled AOT works fine on non-Morello platforms:



WAMR AOT mode also works perfectly with native libraries – no differently to WASM Interpreter mode. The example below calls two native functions from the AOT code, *native\_do\_sum()* and *native\_print\_something()*. Both of these are implemented in C code in the shared object library which is passed to WAMR as a --*native-lib* command line argument:



# WAMR External References

External References (also known as *extenrefs*) is a new WASM feature that has not yet formally been accepted into the WASM specification, however it is already supported by WAMR. Therefore we ported this feature to CHERI.

The External References is concept is to allow WASM to manipulate *opaque handles* which are provided from native code. WASM cannot modify these in any way, and so in effect they serve as meta information, but they can be supplied from native code, tabulated and returned to native code as needed.

The new External References feature also adds additional operations to manipulate tables; these are supported and tested as part of the CHERI port.

## Overview

An *externref* is somewhat of an abstract concept; as WAMR is written in C and C++ the WAMR realisation of an externref is a native machine pointers (actually a *uintptr\_t*). The size of the externref pointer is therefore machine dependent and is built as such for WAMR.

Internally, WAMR does not manipulate an externref as a *uintptr\_t*. Instead, when WAMR is given an externref it stores it in a HashMap and represents the externref as an index for all subsequent operations. The index is just a number (actually a *uint32\_t)*.

When an externref needs passing to a native function (either via a parameter in a call to a native function, or as the result of a WASM function returning to native) then WAMR reads the HashMap to access the actual externref based on the given index.

*NOTE: The actual hash table used in WAMR is crude. A global index is used as the key, the “hash” is actually not using the externref value at all as it just returns the index. The index effectively wraps when it hits the size of the available hash space - therefore there will be conflicts as soon as more than the pre-allocated number of externrefs are stored. Conflicts are handled with a linked list for each index, as is fairly standard implementation of a hash table. Note that this code in WAMR has the “feel” that it may be modified at some future time, because the hash function could be passed a uintptr\_t i.e an externref - this is taken into account when making CHERI changes.*

The externref concept in WAMR is shown below:

A diagram of a computer application

Description automatically generated

### Null ExternRefs

The WASM feature specifies the concept of a “null” externref; this is implemented by adding a new reference type, *ref.null extern.*

Unlike in higher level languages there is no concept of a special “null” type in C, though by convention a pointer with the value zero is typically used and this is used when generating an explicity *ref.null extern*. However, and strangely, when passing an externref pointer value *to* WAMR the value of “-1” must be used! This is doubly strange since for a ***u****intptr\_t* then -1 is actually “MAX\_UINT”.

Internally WAMR sets the HashMap index for a null extenref as “-1” (which again is MAX\_UINT). This is a special case and therefore the actual pointer value is not saved to the HashMap in this case.

The below summarises how WAMR works with Null externrefs:

* Passing externref pointer = -1 from native to WAMR is detected as *ref.null*
* Passing an externref null pointer index from WAMR to native passes pointer = 0
* Explicit WASM instruction to set *ref.null extern* will pass pointer = 0 to native

### Command Line Handling

The WAMR *iwasm* front-end executable program provides support for passing extenrefs to WASM functions as command-line arguments. This is handled by converting a numeric string to an integer, and then setting this as the value of a pointer.

This pointer can then be stored in the HashMap in the same way as if it had been passed from native code.

When reporting back to the *iwasm* output, the pointers value is converted to a string and then this text is displayed. This would be the case if, for example, a WASM function returned an *externref*.

#### Null ExternRefs

When passing a null externref from the command line, the string *“null”* can be used to explicitly create a *ref.null extern* inside WAMR.

When printing the string for a *ref.null,* the string “0” is not used. Instead, the string *“extern:ref.null”* is used.

### Additional WASM Operations

The WAMR externref feature enables a number of additional operations as defined in the WASM draft specification, as follows:

* Tables can now be *initialised*
* Tables can be resized – the *grow* operation is added
* Tables can be *fill*ed with an existing externref value from an *elem*
* The *funcref* type is also supported

WAMR handled these new operations internally, using the index of the externref in the HashMap as required.

### WAMR Internal Operation Details

#### Argument Passing from Native Functions

Externrefs are clearly useless without native functions, therefore it is necessary to review how native functions can transfer arguments to and from WAMR. There are three functions which enable native code (or the *iwasm* front-end) to call WASM functions, they differ in the mechanism to pass the arguments:

* *wasm\_runtime\_call\_wasm()*
  + Call WASM via an array of values with correct data layout to match the function
* *wasm\_runtime\_call\_wasm\_a()*
  + Call WASM via a *wasm\_val\_t* structure, specifying type+value for each argument
* *wasm\_runtime\_call\_wasm\_v()*
  + Call the function using variable length argument passing (i.e *va\_args* construct)

However, under the bonnet all of these essentially use the mechanism of transferring the *uintptr\_t* externref to an array of integers and then recovering the extenref later for conversion to the HashMap index. A result of *externref* typeis transferred in the same way, back to the array for returning.

#### Argument Passing to Native Functions

When WASM calls a native import and passes an externref, the *uintptr\_t* is extracted from the HashMap and then passed via standard C function calling convention for the architecture of choice in the same way any pointer would be. This is no different to any other native argument handling, but noting that the function’s signature has a new type for externrefs (which is an “r”) and so it needs to be parsed accordingly in the native function invoker call, *wasm\_runtime\_invoke\_native()*.

Results are returned in the same way - they need to be extracted as necessary from the function return argument as is currently done for other types.

#### AOT Support

For AOT, externrefs are relatively painless because the AOT file can simply treat them as an index since it is WAMR code which interfaces with the native side that does the conversion from externref pointer to / from the index. AOT mode, then, works no differently from Interpreter mode – it just “implicitly” works.

#### Building Externref Support

To build WAMR with externref support, set the CMake variable *WAMR\_BUILD\_REF\_TYPES=1.* As described, this will not only add the externref support but also all of the extended table functions and the *funcref* type.

## Modifications for CHERI

As stated previously, the internal handling of an *externref* is via a hash map index, which is an integer. Therefore no changes are needed to this for CHERI and any new operations added with the feature do not need to change, they will only need to be verified that they will work.

The problem areas are related to the storing of an externref into the hash map (i.e the insertion of an extenref into the system) and the extraction of an externref back into native.

As we have seen, WAMR expects to freely treat the externref *uinptr\_t* to an integer but of course this is not valid on CHERI as pointers are capabilities. The two main problems in doing the conversions are the same as have been encountered throughout the porting work:

* sizeof(integer) != sizeof(capability pointer) on CHERI
* Capability pointers must be aligned (to 128-bits on Morello) and therefore the conversion mechanism of treating the pointer as an array of 32-bit integers will not work

Further complications are encountered when dealing with null extenrefs, and passing externrefs to and from the command line as strings.

### Technological Solutions

This section describes how pointer alignment, sealing and the null capability can be used to solve the problems of insertion and extraction of capability pointer extenrefs into WAMR.

#### CHERI Pointer Alignment and Array Transfer

To pass capability pointer extenrefs using a byte array to and from the extenref HashMap a similar solution to the WAMR WASM Fast Interpreter problem is used:

1. Enough space is reserved for an externref to allow for alignment padding, so 2 x sizeof(uintptr\_t) is reserved
2. The buffer pointer is then aligned up to the next 128-bit alignment boundary
3. The pointer is then copied, treating the array as a uintptr\_t \*.

The process of copying a uintptr\_t to an array on CHERI is given in the below code:

**// "array" is a buffer of uint32\_t  
// "offset" is current index into the array (in uint32\_ts)**

**// "externref" is a uintptr\_t representing the externref object**

**uintptr\_t\* aligned\_addr = (uintptr\_t\*)cheri\_align\_up(&array[offset], 16);**

**\*aligned\_addr = externref;**

**offset += 2 \* sizeof(uintptr\_t);**

Note that for this to work, the array must be large enough to accommodate *2 x sizeof(uintptr\_t)* for each externref variable. WAMR determines the size it needs for all parameters before allocating enough memory for them and uses the signature of the function to determine what the parameter types are, so this will work.

Three new functions have been created in order to support the above mechanism:

* *wasm\_cheri\_write\_externref\_to\_array()*
  + Copy an externref from a capability pointer to an array
  + Used to pass back a capability pointer from WAMR to native, after extracting from the HashMap
  + Can also be used within native code to pass the externref into an array for transfer to WAMR
* *wasm\_cheri\_read\_externref\_from\_array()*
  + Copy an extenref from an array to a capability pointer
  + Used to obtain the externref from a native-passed array ahead of conversion to HashMap index
  + Can also be used within native user code to extract an externref passed back from WAMR
* *wasm\_cheri\_extenref\_size()*
  + (inline function) return the size of the space in the array, i.e *2 \* sizeof(uintptr\_t)*

#### Capability Sealing

As an extenref represents a handle which is transferred, but never used, by WASM / WAMR then it makes sense for this capability to be sealed by the native function before passing to WAMR and unsealed by native code when returned from WAMR.

A change made for CHERI is to use builtin wrapper functions *cheri\_is\_sealed()* and *cheri\_tag\_get()* to warn if an extenref being passed to WAMR is not sealed and/or not valid.

***NOTE****: This is only a warning - there is nothing to specify an externref MUST be a pointer, therefore unsealed and/or invalid capabilities are still accepted under CHERI.*

#### Null Externrefs in CHERI

WAMR assumes an externref is null if its value is -1 (actually MAX\_UINT). We keep this restriction for CHERI, but explicitly require that a capability must be ***not valid and also*** have a value of -1 (MAX\_UINT). A capability which has an address of “-1” but with a valid tag, permissions and bounds is then not treated as a null.

This enhances the WAMR system as a null is now explicitly, on CHERI purecap, something which cannot be a valid memory address.

Additionally, if a null capability is passed for an externref then this is also explicitly treated as a *ref.null extern* (i.e an address and tag set to 0).

### Passing Externref from Native Code to WASM

We can now describe the modified process involved in routing an extenref from native code into a WASM function:

1. Native code makes use of *wasm\_cheri\_write\_extenref\_to\_array()* in order to build the structure suitable for the *wasm\_runtime\_call\_wasm()* function
2. *wasm\_runtime\_call\_wasm()* is called as for non-CHERI flow
3. Within *wasm\_runtime\_call\_wasm()*, *wasm\_cheri\_read\_externref\_from\_array()* is now used to correctly extract the externref as a full capability pointer, hence the original capability from the native code is recovered.
4. This is then converted to an index via a modified *wasm\_externref\_obj2ref(),* as per existing functionality.

Note that the *wasm\_extenref\_obj2ref()* is slightly modified to:

* Detect a null as being a capability value of “-1” or “0”, and with a cleared tag
* Warn to *stderr* if a capability is invalid and/or unsealed

#### Returning Externref Result

When passing back an externref result, once the externref has been converted back from an index via *wasm\_externref\_ref2obj()* then *wasm\_cheri\_write\_externref\_to\_array()* is used to format the capability pointer back to an array for transferring as the result from the native → WASM call.

#### Specialisations when using wasm\_runtime\_call\_wasm\_a()

When *wasm\_runtime\_call\_wasm\_a()* is used, internally the structured arguments are converted to an array before the *wasm\_runtime\_call\_wasm()* is then itself used.

As part of this conversion to an array, the functions *wasm\_cheri\_read\_externref\_from\_array()* and *wasm\_cheri\_write\_externref\_to\_array()* are used. The only difference is that in this case the array needs to be allocated by WAMR itself, it is not already allocated by the native code. WAMR therefore needs to know the size of array to allocate, which it calculates based on the size of each individual parameter that needs passing. For an externref, this is given by *wasm\_value\_type\_cell\_num\_outside()* and therefore for CHERI purecap this returns the following (note: size is in number of *uint32\_t*s not bytes!):

return (sizeof(uintptr\_t) << 1) / sizeof(uint32);

#### Specialisations when using wasm\_runtime\_call\_wasm\_v()

*wasm\_runtime\_call\_wasm\_v()* internally converts the *va\_args* into structured values, and is therefore little more than a wrapper around *wasm\_runtime\_call\_wasm\_a()*. Therefore no additional changes are specifically needed for *wasm\_runtime\_call\_wasm\_v().*

### Passing Externref from WASM to Native Function

When WAMR needs to call a native function (either a WASI built-in one or a user supplied native function), the code in *wasm\_runtime\_invoke\_native()* is used to invoke the native function via an assembly code “trampoline”. This has already been implemented for CHERI and already supports passing pointers (for the purpose of strings and buffers). For extenrefs, we can just re-use this same mechanism.

When an extenref needs converting its capability pointer is retrieved via *wasm\_extenref\_ref2obj()* and then the already-created CHERI function *update\_args\_as\_pointer()* is used to pass the variable to the correct buffer for invoking a native function.

#### Handling AOT Mode

In the AOT case, as is already the case for WAMR even without CHERI, an index only is used which is passed as a uint64. Therefore there are no CHERI changes needed to support this specifically for externrefs.

#### Returning Externref Result

Returning an extenref as a result from a native function is a new concept - WAMR previously only supported a 32-bit or 64-bit result. Without CHERI, WAMR handles this as returning a machine word and casting it to a *uintptr\_t* for the extenref, but this of course won’t work for CHERI.

Fortunately, on Morello CHERI purecap (and as specified by the Arm ABI) the result is returned in the C0 register which is itself a full capability register and therefore capable of holding a capability pointer. To support this in the C code, we therefore:

* Add a new prototype for the assembly *invokeNative* function called *invokeNative\_Ptr,* which will return a *void \** type
* Read this return type into a *void \** variable, which is treated as a *uintptr\_t*, and which is then converted to a reference index via the *wasm\_externref\_obj2ref()* in the same way that is being done when passing an externref in from native code

This mechanism makes it possible for a native C code function to be declared that returns a full capability pointer (as *void \*)* and the assembly invoker will then be able to return this value to our WAMR C code where it is saved to a normal *void \** capability – without all the array manipulation shenanigans that proved necessary for *Passing an externref from Native Function to WASM.*

### Command Line Handling Changes

#### Supplying Externref from Command Line

As outlined previously, WAMR supports calling a function direct from the *iwasm* program (or some other application) and passing the parameters as C style *argv* strings. To allow extenrefs to be passed WAMR will convert a string to an unsigned long integer in order to read an externref pointer on 64-bit platforms.

In the CHERI case, it is not possible to create a full *capability* pointer “at will” by converting a string to a pointer - all capability pointers need to be assigned from a system call such as *malloc()*. Therefore for CHERI we will continue to support passing an extenref from the command line but it will always be treated simply as a value, i.e the tag will be cleared and that capability pointer won’t be valid.

This conversion from a string is done in the same way as for the Aarch64 non-CHERI case; a 64-bit value is converted to a pointer (with a cleared tag) and this is then written in the usual way to the array used to actually call the function. This snippet of code shows the process:

**// Values read from string cannot be pure capabilities**

**// Instead an i64 is stored for the capability - the tag will be invalid**

**uint64 value;**

**if (strncasecmp(argv[i], "null", 4) == 0) {**

**value = (uint64)-1LL;**

**}**

**else {**

**value = strtoull(argv[i], &endptr, 0);**

**}**

**// Write value as uintptr\_t and increment p appropriately**

**wasm\_cheri\_write\_externref\_to\_array((uintptr\_t)value, argv1, &p);**

Note that from the above code, as discussed previously, a string of “null” results in the WAMR representation of a null extenref, which is the value of “-1” with a cleared tag on CHERI.

#### Returning Externref Values from Command Line

When a function on the command line is called, the return value is printed to *stdout*. Returning an extenref is supported; printing for CHERI is handled much the same as for a 64-bit architecture except the externref is “pretty printed” to display its bounds and permissions if it is a full capability pointer.

The code below is what is used:

**uintptr\_t externref = wasm\_cheri\_read\_externref\_from\_array(argv1, &k);**

**if (wasm\_cheri\_externref\_is\_null(externref))**

**{**

**os\_printf("extern:ref.null");**

**}**

**else**

**{**

**os\_printf("%#p:ref.extern", (void\*)externref);**

**}**

Note that the value is first extracted from the array (since the function call returns an array, as it would do for passing the result back to native) and then we check if it is a null externref or not.

As already explained, the *wasm\_cheri\_externref\_is\_null()* checks for a cleared tag and either a value of 0 or “-1” (since WAMR internally uses either of these two values to represent a null extenref). If this is the case then we print the special “extern:ref.null” string, otherwise we print the full capability including permissions, boundary and offset.

## ExternRef Testing & Validation

Externrefs rely on native code, and for CHERI we know this native code needs to change in order to support the manipulation of capability pointers to be able to successfully pass them to WASM functions.

To test the passing of extenrefs to a native function that is called from a WASM function, we have extended the existing user native library code to provide simple functions that can:

* Generate a valid externref, and return it to WASM in response to a function call
* Be called from WASM and supplied with the externref, and check it is valid

To use these new native test functions, a new WASM application is needed and so one is supplied. This simply exercises the new native functions to move an externef around the system.

***NOTE:*** *Unfortunately the WASI TestSuite does not support passing the name of a user-supplied native library on the command-line and therefore at the time of writing this externref test case must be executed manually.*

Although calling a WASM function which takes externrefs from the *iwasm* program can only supply invalid capabilities (since they are converted from command-line argument strings), we still need a regression test to ensure that we can inject (invalid) capabilities into the system and print the value when using the command-line. The test case described above includes a function that can be supplied with two extenrefs, and via a command-line option one or the other, or *ref.null*, is returned.

If we want to call a WASM function from native and pass in *valid* extenrefs on CHERI then *iwasm* is not suitable and we need to roll our own application. To this end, a standalone native test application has been produced which will call dedicated WASM test functions to pass externref arguments and receive back externref values. This also implements the simple native functions mentioned above which can be called from WAMR, therefore testing the passing of externrefs into WASM functions and WASM calling native functions with extenref arguments.

To accompany this bespoke application a further WASM application test case is added. This implements a large set of tests, which exercise all of the table operations added as part of this new WASM feature as well as testing externrefs can be passed to WASM functions using all of the different methods that WAMR provides.

The rest of this specification describes the extended native test library, the new externref test application and all the associated test cases.

### Extended Native Test Library

The existing native library is extended by providing two new functions, *get\_externref()* and *put\_externref()*.

The *get\_externref()* function allocates an array on the (native) heap and copies some predefined test values into it. The function then returns a capability pointer (as a *uintptr\_t*) to the heap buffer, after first *sealing* using a predetermined sealer capability.

The *put\_externref()* function is given a sealed capability. It unseals this using the same sealer as the *get()* function and then treats the capability as a pointer to a heap buffer. This should be the same as the one passed back from the getter function, and so the values in the buffer are compared with the predefined values to check they are the same. If they are the same, the function returns 0 and if they are not the same (or if there was e.g a capability tag error) then the function returns non-zero.

The idea, then, is that a WASM application will call *get\_externref()* to obtain an externref value. It will then call *put\_externref()* with this value in order to pass it back to the native code. If the test passes, then we validate we are able to move an externref in CHERI between native code -> WASM application -> back to native code.

#### Sealing

The sealing operation uses a hard-coded offset which is applied to the AT\_CHERI\_SEAL\_CAP obtained from the auxiliary vector. The sealer capability is obtained using the following snippet of code:

**uint8\_t\* get\_sealer()**

**{**

**static uint8\_t\* sealer = (uint8\_t\*)getauxptr(AT\_CHERI\_SEAL\_CAP) + 0x1234;**

**return sealer;**

**}**

Note that the sealer is declared static, so it only needs to be calculated once.

This function *get\_sealer()* can then be used to obtain the sealer capability for the *get\_externref()* sealing operation:

**return reinterpret\_cast<uintptr\_t>(cheri\_seal(p, get\_sealer()));**

and for unsealing in the *put\_externref()* function:

**auto array = reinterpret\_cast<uint32\_t\*>(cheri\_unseal(p, get\_sealer()));**

#### WASM Application Test Case

A simple WASM application is developed to make use of the new native functions. This was hand-crafted from text (WAT) so that we can make the test as simplistic and specific as possible. This is shown below:

**(module**

**(import "wasi\_snapshot\_preview1" "proc\_exit"**

**(func $wasi-proc-exit (param i32))**

**)**

**(import "env" "get\_externref"**

**(func $native-get-externref (result externref))**

**)**

**(import "env" "put\_externref"**

**(func $native-put-externref (param externref) (result i32))**

**)**

**;; We use a WASI function so export memory**

**(memory $memory (export "memory") 2)**

**;; main: Gets an externref from native lib, returns value**

**;; and native will check is the same**

**(func $main (export "\_start")**

**(call $native-get-externref)**

**(call $native-put-externref)**

**(call $wasi-proc-exit)**

**(unreachable)**

**)**

**)**

It can be seen that the *get\_externref()* returns an *externref* which is the new WASM type, and *put\_externref()* takes an *externref*.

The main function calls the *get\_externref()* and on exit the *externref* is placed on the WASM stack so it is then available as the argument to *put\_externref()*. On exit of *put\_externref()* the result code is on the WASM stack and so this is passed to the WASI exit function, which causes the WAMR application to return it as the program exit value. The program exit value is then zero for success (standard POSIX behaviour) and non-zero for failure.

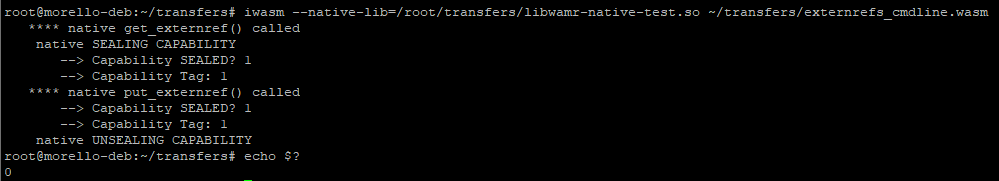
The test flow is shown below:

A diagram of a computer program

Description automatically generated

#### Test Execution and Results

The test needs to be run manually, the command invocation on Morello is shown below along with logged output from the native library:



Note that for both functions the capability sealing succeeded and the tag was valid, and that the return code for the command (obtained via *echo $?*) reported as 0, which is success.

### Command Line Externref Testing

The simple test case from above is extended to support passing (invalid) extenrefs from the command line, and this includes handling *ref.null* externs. Again, hand-crafted WASM Text (WAT) is used which is then converted into a binary WASM application.

The simple WASM function is shown below (as text):

**;; do-test: Simple command line test to take externrefs (not capabilities)**

**;; and return one of them, or return null**

**;; Given idx = 1, 2 or other**

**;; Return first extref if idx == 1, 2nd if idx == 2 else return null extref**

**(func $do-test (export "do\_test")**

**(param $idx i32) (param $ext1 externref) (param $ext2 externref)**

**(result externref)**

**(i32.eq (local.get $idx) (i32.const 1))**

**(if**

**(then**

**(local.get $ext1)**

**return**

**)**

**)**

**(i32.eq (local.get $idx) (i32.const 2))**

**(if**

**(then**

**(local.get $ext2)**

**return**

**)**

**)**

**(ref.null extern)**

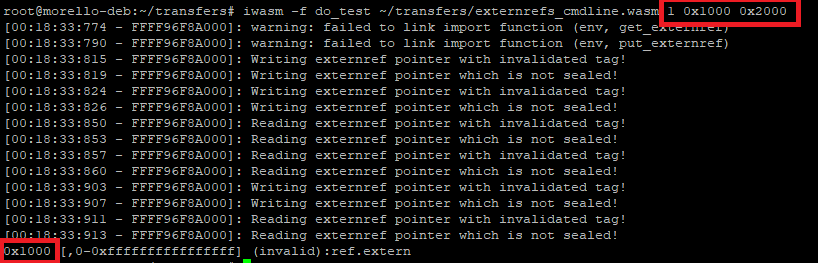
**return**

**)**

This simple function is given an integer followed by two *extenref*, which are loaded from the command line and therefore which will end up as invalid capabilities. The first argument controls which externref is returned, or if neither 1 nor 2 then *ref.null extern* is returned.

Running the test (as a specific named function with arguments from *iwasm* command-line) we can see the results.

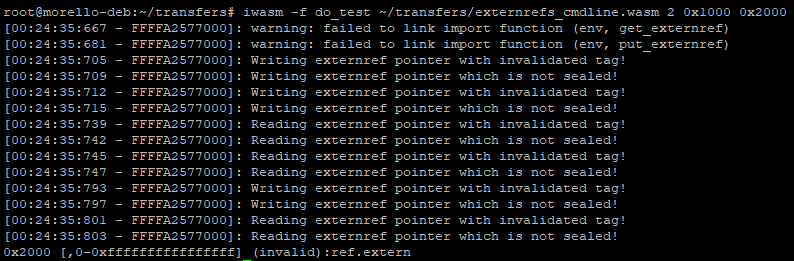
In the first case, we request the function to return the first *externref,* which is 0x1000:



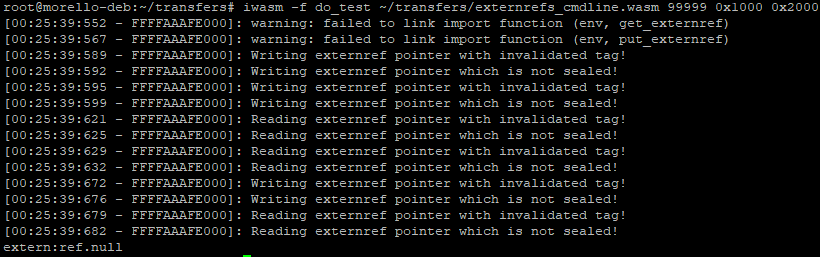
**Notes:**

* Warnings on failure to find import functions can be ignored, as this is because we used the same test case as before but didn’t load the native library this time
* Each time we obtain or excrete an externref we are warned that it is not valid, and also not sealed
  + This is expected because the command-line externref was just converted from a string
* The resulting externref that is printed to *stdout* is pretty-printed, but we note it is invalid (and obviously has no permissions / boundary set to any realistic value)

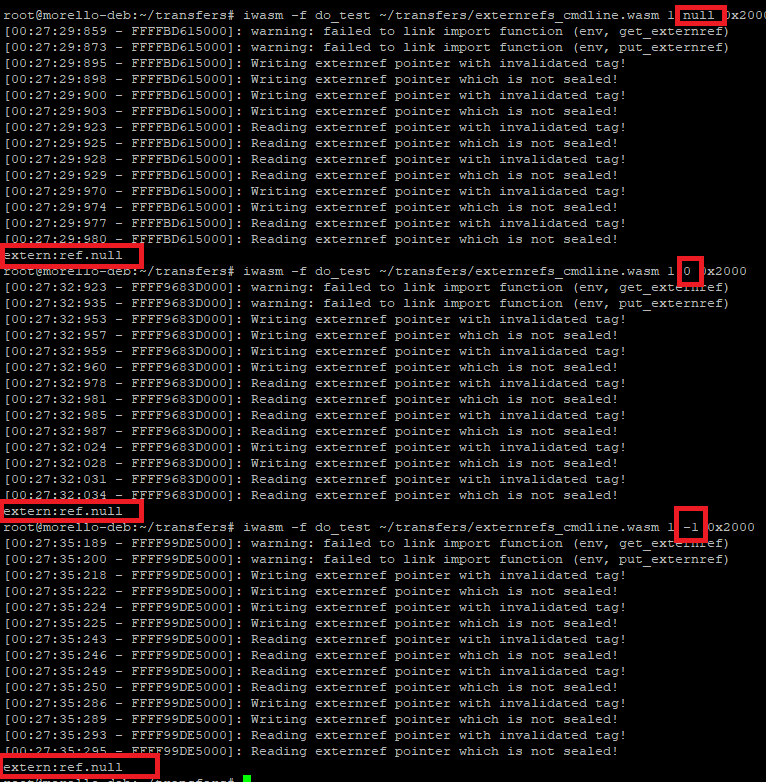
Here is the result for the second case, this time we see we return 0x2000:



And finally, we show returning a *ref.null*:



Note also that (as designed), passing an externref value of “null”, 0 or -1 also causes WAMR to interpret it as a ref.null:



### Externref Standalone Native Test Application

The standalone native test application eliminates the problem of reading externrefs from the *iwasm* command-line because it directly generates valid capabilities and passes them to the WASM function under test. In fact, it uses the same *get\_externref()* as before to do this – although a native export callable from WASM it is just a C++ function and so can be called directly from the native code.

This new application is called *externref-app* and is provided as part of the test system for CHERI. It is very similar to *iwasm* front-end and makes use of the WAMR exported API to load a WASM module, initialize an execution environment and obtain the function to test.

The application is designed to work with a specific WASM function called *externref\_test\_in\_wasm*() which takes two externref arguments and returns an externref, therefore the WASM module being loaded must implement such a function.

The WASM can then call back native functions exported from externref-app in order to send back an externref for validation, and pretty-print an externref.

#### Building the Application

The standalone application is built when the *CMake* variable *WAMR\_EXTERNREF\_APP=1* is set. In this case it is built instead of WAMR’s *iwasm* program. To support use of Visual Studio, a dedicated make target is added for both Morello Purecap and also Linux x86\_64. The non-Morello version is for backwards-compatibility testing as even without CHERI being present the application should work properly.

Naturally, the build configuration will enable *WAMR\_BUILD\_REF\_TYPES=1* in order to build externref support into WAMR.

#### WASM Application Test Case

The WASM function is handcrafted as WAT and is shown below:

**(import "env" "print\_externref"**

**(func $native-print-externref (param externref) (result i32))**

**)**

**(import "env" "put\_externref"**

**(func $native-put-externref (param externref) (result i32))**

**)**

**(func $externref\_test\_in\_wasm (export "externref\_test\_in\_wasm")**

**(param externref externref)**

**(result externref)**

**(call $native-print-externref (local.get 0))**

**drop**

**(call $native-print-externref (local.get 1))**

**drop**

**;; take the first arg and call back to native to put it**

**(call $native-put-externref (local.get 0))**

**drop**

**;; Return the 2nd arg straight back again**

**(local.get 1)**

**)**

The test function works as follows, the full flow with interaction to native is shown below:

1. NATIVE: Generate two externrefs, which are pointers to array of known values
2. NATIVE -> WASM: Call *externref\_test\_in\_wasm()* function
3. WASM -> NATIVE: Call *print\_externref()*, passing **first** externref for printing
4. NATIVE: Pretty print externref value
5. WASM->NATIVE: Call *print-externref(),* passing **second** extenref for printing
6. NATIVE: Pretty print externref value
7. WASM->NATIVE: Call *put-externref(),* passing **first** externref for validation
8. NATIVE: Dump values in buffer pointed by externref to stdout
9. WASM->NATIVE: Return **second** extenref
10. NATIVE: Validate externref is expected pointer value, dump values in buffer to stdout

#### New WASM Feature Operations Testing

As well as the above externref test, the WASM application also includes native-callable functions which will exercise the new table and funcref operations.

The WASM for this is somewhat complicated, but uses three functions which are called from the native code as follows:

| WASM Function | Does | Returns |
| --- | --- | --- |
| *table\_test\_set()* | Grows a table of 2 externrefs to a size of 5.  Writes 4 given externrefs into the table. The fifth entry remains as Null. | Final size of table (should be 5) |
| *table\_test\_get()* | Gets the externref at given index from the above table. | Externref value @ index |
| *table\_test\_null()* | Checks if a given index in the table contains a Null externref or not | 1 if it is Null, else 0 |
| *table\_test\_ops()* | Copies first table entry to last, and overwrites the first two entries with Nulls.  This is done via performing a number of table operations, as follows:   1. Copies first table entry to the last one, overwriting any existing 2. Fills a second table with two Null externrefs (copying from an element) 3. Overwrites first two entries of main table with the Nulls from this new table | Nothing |
| *funcref\_test()* | Creates a function reference table, and adds a function which itself calls the *print-externref()* function. Then calls this via *call\_indirect,* passing a given externref.  Put simply, this should end up pretty-printing the externref it is given using the native code. | 0 if everything worked ok. |

##### Operation for Table Testing

With reference to the WASM application above, the table testing operations are as follows:

*Setup: => Write 0x11, 0x22, 0x33, 0x44 into the table*

1. NATIVE: Generate four externrefs (using integers, i.e invalid capabilities) for testing:
   * 0x11, 0x22, 0x33, 0x44
2. NATIVE -> WASM: Call *table\_test\_set()* function passing the four values
3. WASM: Table is grown from 2 -> 5 entries
   * should then contain 0x11, 0x22, 0x33, 0x44, NULL
4. NATIVE: Validate return value == 5 i.e expected table size.

*Getter: => Verify table contains 0x11, 0x22, 0x33, 0x44, Null*

1. NATIVE: FOR EACH index in range 1..4 of 5
2. NATIVE->WASM: Call *table\_test\_get()* to obtain externref at index
3. NATIVE->WASM: Call *table\_test\_null()* to see if the returned externref is null
4. NATIVE: Verify externref returned is not null
5. NATIVE: Verify externref is the one we expect in the table at this index
6. NATIVE: FOR index = 5 i.e last element
7. NATIVE->WASM: Call *table\_test\_get()* : Should return *ref.null*
8. NATIVE->WASM: Call *table\_test\_null()* on returned externref
9. NATVIVE: Verify value was a null externref

*Operations Test: Perform the operation to change table to Null, Null, 0x33, 0x44, 0x11*

1. NATIVE->WASM: Call *table\_test\_ops()*

*Repeat Getter: Verify returned elements are now Null, Null, 0x33, 0x44, 0x11*

1. [Repeat steps 5..13]

*Funcref Test: Check we can print an externref passed to a function, via indirect call of funcref*

1. NATIVE: Make a (capability invalid) externref, set to a numeric value
2. NATIVE->WASM: Call *funcref\_test()*, passing the externref
3. WASM->NATIVE: Calls *print\_externref(), passing externref*
4. NATIVE: Print the externref (numeric value) to *stdout*
5. WASM->NATIVE: Return result of *print\_externref()*
6. NATIVE: Verify the operation succeeded

#### Testing Flow

There are three different WAMR API functions to call a WASM function, and so the same extenref test is repeated for each one. Then, the table and funcref tests are also performed, as shown in the below code snippet:

**if (0 != externref\_test\_call\_wasm(exec\_env, func, module\_inst) ||**

**0 != externref\_test\_call\_wasm\_a(exec\_env, func, module\_inst) ||**

**0 != externref\_test\_call\_wasm\_v(exec\_env, func, module\_inst) ||**

**0 != externref\_table\_test(exec\_env, module\_inst) ||**

**0 != funcref\_table\_test(exec\_env, module\_inst)**

**)**

**{**

**std::cout << " native testing FAILED" << std::endl;**

**result = -1;**

**}**

**return result;**

The overall result is then 0 if all the individual tests passed.

Each of the *externref\_test\_call\_wasmX()* functions will repeat the main externref test using one of the WAMR API methods to call a WASM function. They generate test data to an array by internally calling *get\_externref()* to allocate a heap buffer of two integers and write test data into this. The test values are read from a predefined set; each time a value is obtained from the set then the next value in the set is used. It is circular, therefore test values will repeat after a while.

The capability pointer to the buffer is then used for the testing, and when returned it is verified to be correct.

#### AOT Support

AOT mode is supported for this test, this is achieved by compiling the *externrefs\_for\_app.wasm* to AOT.

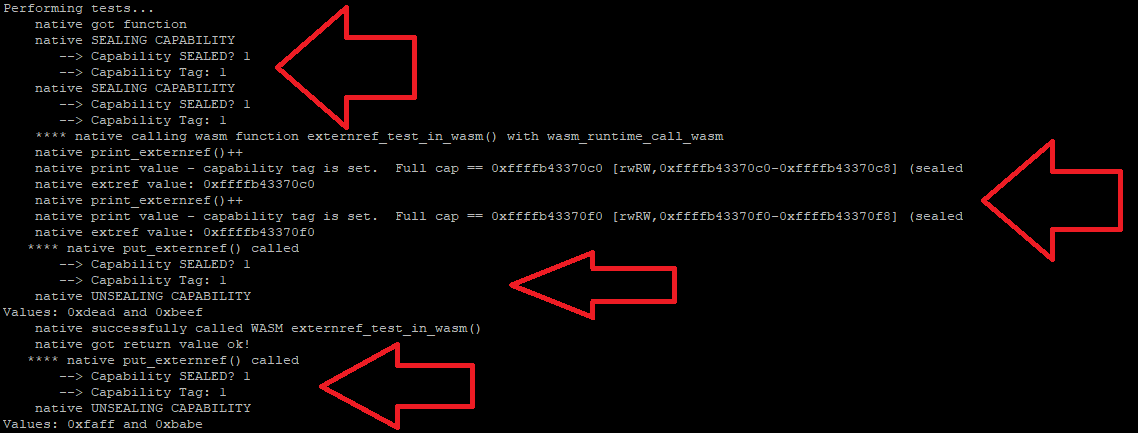
Like the WASM Interpreter mode, AOT is supported on both Linux\_x86\_64 as well as Morello Purecap using the bespoke test application.

#### Test Results

The test is quite complicated as it does a number of steps in one go, therefore the output dump is shown in snippets in this section.

##### Externref\_test\_call\_wasm

This is testing the externref flow via the *wasm\_runtime\_call\_wasm()* function:

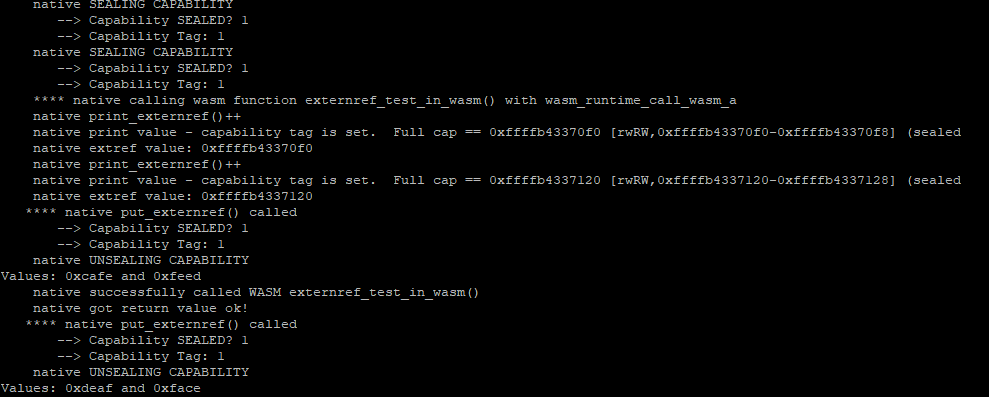


Each step is shown by a red arrow, they are in order:

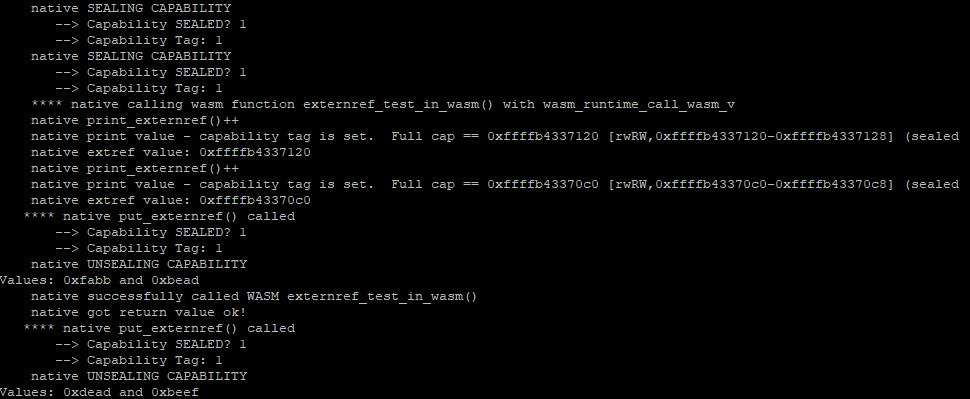
1. Generate two externrefs from buffer of values and call WASM function
2. WASM calls *print\_externref()* for each, we dump both pointers
3. WASM calls *put\_externref()* on the first externref, we retrieve values
4. WASM returns and native calls *put\_externref()* on the 2nd (returned) externref. Again we print the values in the buffer.

##### Externref\_test\_call\_wasm\_a and Externref\_test\_call\_wasm\_v

This is the same flow as above but using *wasm\_runtime\_call\_wasm\_a()*:



And then using *wasm\_runtime\_call\_wasm\_v()*:

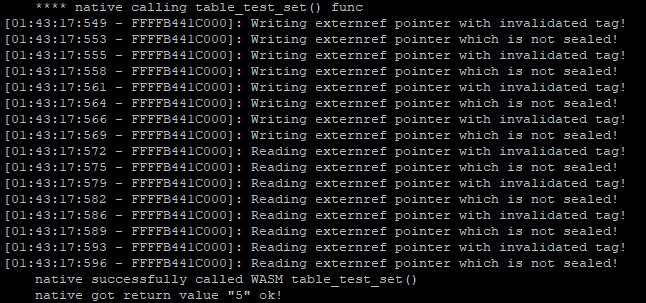


It can be seen the results are the same in both cases, other than different test data values.

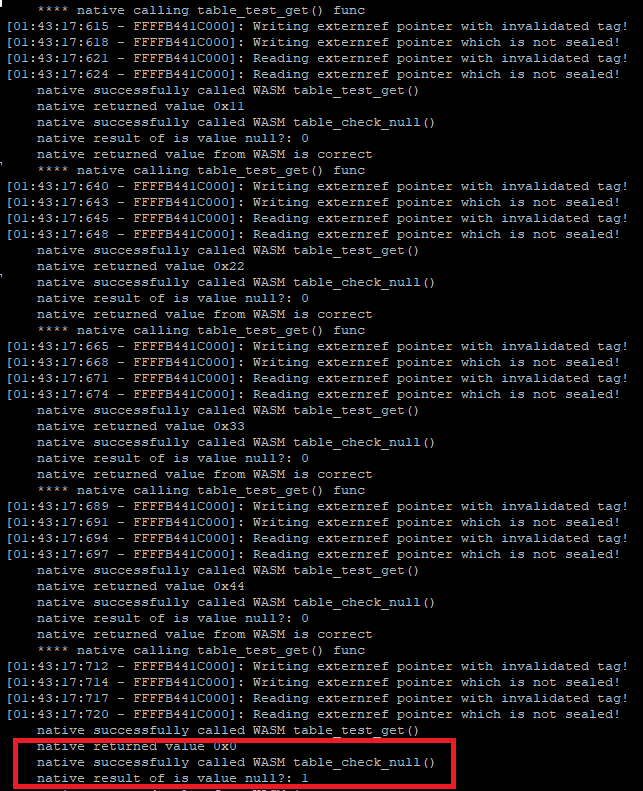
##### Table Test

The table testing performs the table setup, getter, value replacement and repeats the getter. The results are shown below:

*Setter:*

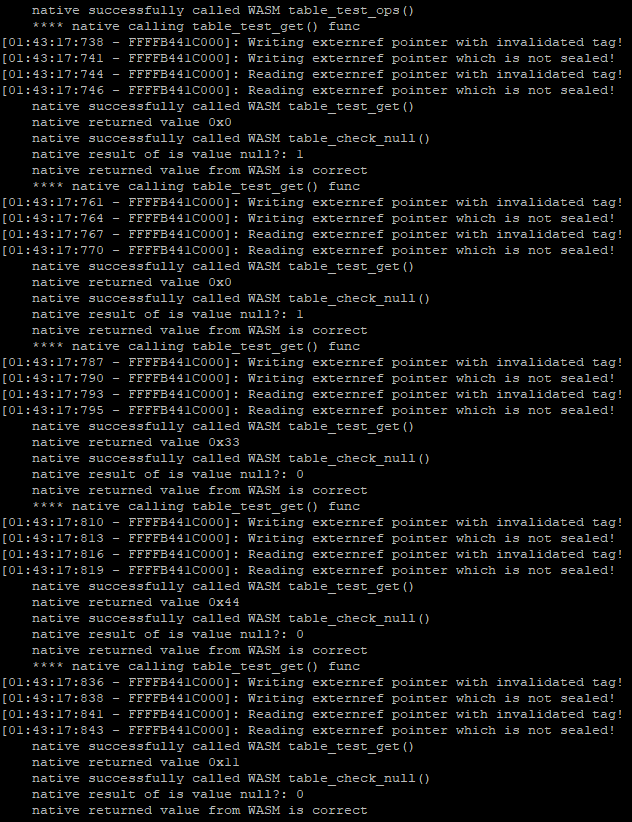


*Getter:*



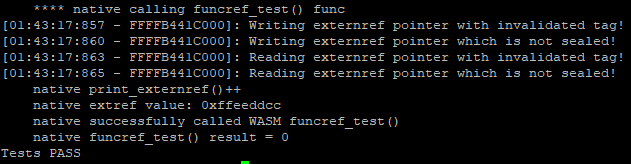
Note the returned value in the last case is 0, and the Null check returns 1 i.e it is null.

*Calling table\_test\_ops() and repeat getter:*



Note that this time the values returned (“*native returned value”*) is 0x00, 0x00, 0x33, 0x44, 0x11.

*Funcref Test:*

**

# Testing

This section describes the test environment in more detail.

Throughout the project testing has evolved as the work has gone on. Initially, a bespoke front-end application was developed to make it easier to test and debug WAMR by calling it from a more controlled environment. Although largely now redundant, this is described for posterity.

As testing then moved to the main *iwasm* standalone executable, test cases were developed which were run manually. Some initial examples of these were covered in the discussion on the Interpreter modes (section 5.3). At this stage, very few tests were run as part of a *regression set* and only occasionally on other platforms (such as Linux x86).

It was realised that some sort of automated test framework – even in a limited capacity – would greatly benefit the project. As described, as part of the WASI support work, *WASI Test Suite* was introduced specifically to allow a number of WASI functionality to be verified. As there were so many of these tests it was not feasible to run them manually, and so it was required that *WASI Test Suite* be adapted to work with Morello pure-cap.

The addition of AOT doubled the number of test cases as each WASM file can now be compiled to AOT and then tested too. It was therefore logical to further adapt the *WASI Test Suite* to be able to test AOT mode as well as Interpreter mode.

The changes implemented to achieve this are described in this section, along with the latest test results. It is now possible to, for a single WASM file, generate 6 test cases: a WASM for Morello pure-cap, Morello hybrid-cap and Linux x86 and an AOT for each of the three platforms as well. All of these test cases can now be run with very little manual input.

## WAMR Front End Application

To support initial WAMR porting development a bespoke front-end application was developed. This eliminated much of the complexity and additional options of WAMR’s *iwasm* program and made it simpler to understand how the interpreter mode worked. Once interpreter mode (classic) was working then development switched largely to using the main *iwasm* program.

The bespoke front-end can be built instead of *iwasm* by adding the CMake variable *WAMR\_APP=1* to the CMake command line or *CMakePresets.json*. The default is 0, which will build *iwasm*.

The bespoke front-end app has its own *CMakeLists.txt* which is very similar to that that in the *product-mini/platforms/linux-cheri-purecap* folder, except it includes the front-end’s *main.cpp* and other associated files. The code can be found in the *frontend/* folder (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/frontend>).

The bespoke front-end is written in C++. The command-line for it is as follows:

**wamr-app <wasm-file> [<fn\_to\_run\_default\_main>] [param1 param2…]**

Where:

* *wasm-file* is the WASM to load and run through the interpreter
* *fn\_to\_run\_default\_main* is the exported WASM function to execute; if none is specified, the program attempts to run a WASM “main()” function
* *param1, param2…* are arguments (if any) to pass to the WASM application

### Code Structure

The *main()* program creates a *std::ifstream* and reads in the WASM file to a *std::vector.* It then creates an instance of a class *Runner* and passes the WASM file buffer and function name to the constructor. Main then calls *Runner::Run(),* passing in any additional command-line arguments as WASM arguments, and exits with the return code from this function.

The *Runner* class constructor initializes WAMR, loads the WASM module and creates an execution environment as well as looking up the specified function as a WASM export.

*Runner::Run()* extracts any command-line arguments, and calls *wasm\_runtime\_call\_wasm()* to run the WASM function previously specified with provided command-line arguments or if no function was specified, calls *wasm\_runtime\_execute\_main()* to run the WASM *main()* function. If *main()* was run, then the program attempts to recover the exit code from *main()* which would have been set by a WASI call to *wasi\_proc\_exit()*.

WASM execution then being complete, the *Runner* destructor tears-down the WAMR runtime environment and the program exits.

## Automated Test System with WASI Test Suite

In Section 6.4 and Section 7.4 an overview of the flow when using WASI Test Suite for WAMR Interpreter and AOT mode testing was presented. In this section we explain more of the detail on how WASI Test Suite *test adapters* were created to enable WAMR automated testing of CHERI functionality.

### Test Adapters

As explained in Section 6.4.1.2, a test adapter is run as a standalone process and must take various common input arguments. An adapter was developed to remotely run WAMR on the Morello board, with the adapter running on the host (either Windows, Linux or WSL2 Linux).

This adapter can be found in the *verifoxx-cheri-wamr* repository, here: <https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/tests/wamr-linux-cheri-purecap-tests/wasi-testsuite/wamr-cheri.py>.

A further adapter was developed to be used for AOT testing. This also runs WAMR remotely, but has the additional step of first running *wamrc* to compile the WASM test vector to an AOT file. The *wamrc* step is run as a python sub-process and can either be run locally or on a remote machine.

The AOT version of the adapter is <https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/blob/develop/tests/wamr-linux-cheri-purecap-tests/wasi-testsuite/wamr-cheri-aot.py>.

The adapters make use of the python *fabric* package for remote machine operations and therefore a *requirements.txt* is included in the folder containing the *wamr\_cheri.py* to install it. As always with python, it is recommended to create a virtual environment (with *venv* or similar) and install needed package dependencies only in that environment.

Due to the command-line argument list being restricted by WASI Test Suite, additional arguments needed for the remote WAMR / wamrc access are supplied from a JSON file. The JSON filename itself is read from an environment variable, *WAMR\_CHERI\_ADAPTER*. This must be an absolute pathname on the filesystem where the script is being run.

#### JSON Settings

A number of settings can be provided in the JSON file. Some of these are mandatory for either the interpreter (*wamr-cheri.py*) or the AOT *(wamr-cheri-aot.py*) adapters, some are mandatory only for the AOT adapters and some are optional. Given that the AOT adapter is to all intents and purposes an extension of the non-AOT adapter, the same JSON can be used for both with the AOT-only settings being ignored when not required.

The following settings are defined for both AOT and interpreter adapters:

| JSON Key | Mandatory? | Description |
| --- | --- | --- |
| hostname | Yes | IP address and optional port of the Morello board for the SSH connection.  Format is <ip>:[<port>] e.g *192.168.0.39:22* where port defaults to 22 (SSH default port). |
| key | Yes | SSH private key file for SSH key authentication. Must be absolute path. Can use *~* for home directory, e.g c:\users\bob\.ssh\mykey can be supplied as ~/.ssh/mykey |
| user | Yes | Username to login to Morello (i.e “*root”*) |
| dest | Yes | Temporary working folder on Morello to host the *file mirror* |
| wamr | Yes | Absolute pathname of *iwasm* program on the Morello machine |
| addrpool | No, optional | If supplied, will be the value of the *–addr-pool=<value>* argument to *iwasm* |
| allowres | No, optional | If supplied, will be the value of the *–allow-resolve=<value>* argument to *iwasm* |

**NOTE:** The *addrpool* and *allowres* will be passed to every instance of *iwasm* and are intended for use with a testsuite whereby every test in the suite is performing some sort of network access.

The following additional settings are defined for the AOT adapter. These will all be accepted, but ignored, by the non-AOT adapter:

| JSON Key | Mandatory? | Description |
| --- | --- | --- |
| wamrc | Yes | Absolute pathname of the *wamrc* program on either the local machine (if running locally) or the remote machine (if running from a remote Linux machine or WSL2) |
| wamrc\_args | No, optional | If supplied, is a list of string arguments to pass as additional command-line arguments to *wamrc*.  Typically used to provide e.g the target-triple for wamrc when cross-compiling.  For example to build for Morello pure-cap, the following would be passed in the JSON file:  **["--target=aarch64",**  **"--target-abi=musl\_purecap",**  **"--cpu=rainier",**  **"—cpu-features=+morello,+c64"]** |
| wamrc\_host | Only if *wamrc* is running remotely | This key provides a JSON sub-dictionary of additional arguments to define SSH access parameters for the remote machine to run wamrc on. This argument is omitted when *wamrc* runs locally on the host that is running the adapter script.  **See below for details.** |

If *wamrc* is to be run remotely (e.g on an additional Linux machine or WSL2) then the *wamrc\_host* key must be supplied in the JSON and its value must be set to an additional dictionary supplying the following:

| wamrc\_host key | Description |
| --- | --- |
| hostname | IP address and optional port of the remote Linux machine running wamrc, for the SSH connection.  Format is <ip>:[<port>] e.g *192.168.0.39:22* where port defaults to 22 (SSH default port). |
| key | SSH private key file for SSH key authentication. Must be absolute path. Can use *~* for home directory, e.g c:\users\bob\.ssh\wsl2\_key can be supplied as ~/.ssh/wsl2\_key |
| user | Username to login to remote machine (e.g your username) |
| dest | Temporary working folder on remote machine. This is used to store the input WASM file and output AOT file when calling *wamrc* |

***NOTE:*** *It can be seen the wamrc\_host dictionary keys are the same names as those used for Morello remote access*.

An example JSON file is shown below:



This example:

* Can be used for AOT and non-AOT adapters
* For AOT, will run *wamrc* remotely on *localhost* using port 2222 for SSH
* For AOT, will build an AOT file for Morello pure-cap target
* Is intended for WASM test cases needing network access, as the optional *addrpool* and *allowres* arguments are provided

Further example JSONs are provided in the *verifoxx-cheri-wamr* repository (<https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/tests/wamr-linux-cheri-purecap-tests/wasi-testsuite/adapter-json-configs>).

#### Remote Access

The test adapters use *fabric* to open a connection to the remote machine, either Morello to run WAMR’s *iwasm* or a remote Linux / WSL2 server to run *wamrc* if applicable. Once the connection is opened then files can be transferred (pushed to remote or pulled from remote) and remote processes are run.

The fabric *Connection* object needs to know the IP address of the remote machine, and the port to use for SSH (default is 22) which are supplied from the JSON file. For logging in, key based authentication is used to avoid having to store a password in plaintext in the settings and therefore the username and a private key file are required.

To use public key authentication for SSH the following pre-steps are carried out:

1. Generate a public / private *keypair* (e.g with *ssh-keygen*)
   * Private key should not require a password
2. Keep the private part private on the host (e.g stored in *~/.ssh*)
3. Transfer the public part to the remote machine (e.g the Morello board)
   * Typically the key is added to *~/.ssh/authorized\_keys* on the target
4. Update the SSH server configuration on the remote machine to accept public key authentication as needed
   * Typically edit the */etc/ssh/ssh-config* file

Instructions on how to enable SSH public key authentication for e.g *OpenSSH* servers are readily available on the internet.

The important point of the access is that it should be able to be automated, therefore these steps are performed one-time only before first attempting to run the WASI test suite.

**NOTE:** Ensure to manually ssh at least once to Morello using PuTTy or the command-line, as it is necessary to accept the use of the key for the first-time login.

#### SSH Access and WSL2

The AOT adapter supports running *WASI Test Suite* on windows and running *wamrc* remotely on WSL2. This requires that the WSL2 install be set up as an SSH server, but the complication of this is that the default SSH port (22) is in use by windows and WSL2 will not have access (at least on Windows 10, there is a built-in SSH server which launches when Windows starts, and this is likely to still be the case on Windows 11).

The solution is essentially to run an SSH server on WSL2 on a different port, and windows will then direct access on that port to WSL2. Typically, an easy to remember user port like 2222 is chosen.

The setup on the WSL2 distribution is then:

1. Install SSH server e.g *OpenSSH*
2. Modify the listening port, e.g change */etc/ssh/sshd\_config “Port”* setting
3. Start the *sshd* service (or enable to start on boot)

It is also necessary to configure Public Key Authorization in the same way that would be done for the Morello board.

As a test, from windows console the following should login to a terminal on the WSL2 Linux distribution (assuming you are using port 2222):

*ssh -i <keyfile> your\_username@localhost -p 2222*

***Note:*** *The above steps enable access only from the local windows machine. It is not advised to allow access from any other machine on the network, especially using an unsecured key based authentication method*.

### Interpreter Test Adapter

The *wamr-cheri.py* test adapter applies to the non-AOT (interpreter) case. The flow of this adapter in operation is as follows:

A screenshot of a computer program

Description automatically generated

1. The adapter first reads the environment variable and loads the corresponding JSON settings file. It is checked:
   * all mandatory settings are present
   * no settings which are not mandatory or optional are present
2. The temporary folder name on the remote machine is determined by appending a subfolder consisting of a random UUID to the provided folder argument
3. The SSH connection is made as explained previously, using supplied arguments
4. The remote folder is created with a remote *mkdir* command (run using fabric and making all necessary paths to resolve the folder name as required) and then the entire contents of the test folder are copied. This will include the WASM test vector file and any folders needed in the test
   * Recall the working directory is set to the test suite folder by the runner which starts the adapter
   * Fabric does not support a “copy folder recursively” option, so we recursively glob the local folder, map to the corresponding path on the target, and copy each file individually
5. The WAMR command string is built in the same way as is done for existing *WASI Test Suite* adapters, and then WAMR is executed remotely via a fabric *Connection.run()* method
   * The working directory for the remote command is set to the temporary working folder on the remote machine, which mirrors using the test suite folder on the local machine
6. On WAMR command exit, *sys.exit()* is called which will exit the adapter program with the WAMR exit code from the remote machine, however…
7. … first the test suite mirror on the remote machine is restored onto the local machine. As this is done in an exception handler *finally* clause, even if the remote WAMR execution failed it still gets executed
   * Again, as fabric does not support getting a folder of the remote machine, an individual copy is done by first searching for all files & folders in the remote folder
   * To support the fact the test may have created additional files and folders, a remote *find* command is used to search for files and folders in the remote folder
   * These are then individually copied back (by first adjusting the path to fit the local mapping), hence the mirror is restored on the local machine
8. The adapter then completes by removing the temporary working folder on the remote machine

### AOT Test Adapter

The *wamr-cheri-aot.py* test adapter performs the above actions as described for the Interpreter adapter, but it has the additional job of first needing to compile the WASM test vector into an AOT file using *wamrc* either locally or an (another) remote machine.

Additionally, once the AOT file has been created it then needs copying to the Morello machine. In the case of the interpreter adapter the WASM file was part of the test suite folder, so it was copied when the test suite folder mirror was created on the remote machine. The AOT file, though, needs to be explicitly specified.

The AOT file is created as a temporary file on the machine where the adapter runs. It is removed from this location during the cleanup after the test. As this is the local machine, where the environment is known, the local machine’s temporary directory is retrieved via *tempfile.gettempdir()*.

**NOTE:** The name of the AOT file is the same as the WASM file, but with the suffix *.aot.*

#### The wamrc Program Runs Locally

In the case that *wamrc* is run locally then the process is quite simple:

1. Run *wamrc* with any optional *wamrc\_args* specified in the JSON
2. Input is *<test\_suite\_folder>/<test\_file>*.wasm
3. Output is *<temp\_folder>/<test\_file>*.aot

The Morello transfer to run WAMR then takes place, with the additional steps associated with copying *<temp\_folder>/<test\_file>*.aot to the Morello mirror folder and then deleting *<temp\_folder>/<test\_file>*.aot when all is finished.

**Note:** When wamrc runs, the working directory is given as the *temp\_folder* therefore the “output to this file” argument (*-o aot\_file.aot*) does not need a path prefix. Recall the WASM test vector file is always passed with an absolute path to the adapter’s command line.

#### The wamrc Program Runs Remotely

A remote *wamrc* is inferred if the JSON settings loaded by the adapter contain the *wamrc\_host* settings.

If *wamrc* runs remotely, then additional steps are required in order to create the temporary file on the local machine, i.e the *<temp\_folder>/<test\_file>*.aot. The steps are similar to those performed for running WAMR remotely on Morello, and are as follows:

1. Open SSH Connection to remote *wamrc* machine
2. Create the temporary working folder on the remote machine
3. Put the WASM file to the temporary working folder on remote machine
4. Build the *wamrc* command string from the JSON supplied arguments + input & output file paths:
   * *wamrc* path supplied by user via JSON file
   * additional *wamrc* arguments to resolve the target supplied by user via JSON file
   * output is *aot\_file.aot* with no path component needed
   * input is the WASM file with no path component needed
5. Run the *wamrc* command remotely:
   * The working directory is the temporary folder which now contains the WASM file, and on completion will contain the AOT file
6. Get the output AOT file from the remote machine
   * It is copied to *<temp\_folder>/<test\_file.aot>* so will now be in the same place it would have been if created with a local *wamrc*
7. Delete the temporary folder on remote, and close the SSH connection

#### Full AOT Flow

The full AOT flow is shown below – parts which are the same as Interpreter mode are not shown in much detail:

A screenshot of a computer

Description automatically generated

## Automated Test Running

Using the WASI Test Suite with the adapters as described then it is possible to configure, merely with settings in a JSON file:

* Remote machine running WAMR *(iwasm)*
* Remote machine running WAMR AOT Compiler *(wamrc)* + wamrc build target machine

In the previous discussions on the adapter the machine running WAMR is always Morello. This supports Morello pure-cap and hybrid-cap, however it also possible to define the remote machine to be a Linux machine and run a Linux x86\_64 version of WAMR, to support a regression set on a non-Morello platform.

As part of the setup of automated testing for the project this has been done and therefore the following combinations have been used to test all WASI Test Suite provided test cases on three different targets for both Interpreter and AOT modes:

| Target | WAMR Mode | Test Adapter | WAMR runs | Wamrc runs |
| --- | --- | --- | --- | --- |
| Morello pure-cap | Interp | *wamr\_cheri.py* | On Morello  Pure-cap *iwasm* version | N/A |
| AOT | *wamr-cheri-aot.py* | On Morello  Pure-cap *iwasm* version | On Linux\_x86\_64,  Cross-compile |
| Morello Hybrid | Interp | *wamr\_cheri.py* | On Morello  Hybrid *iwasm* version | N/A |
| AOT | *wamr-cheri-aot.py* | On Morello  Hybrid *iwasm* version | On Linux\_x86\_64,  Cross-compile |
| Linux x86\_64 | Interp | *wamr\_cheri.py* | On WSL2 Ubuntu  Linux\_x86\_64 | N/A |
| AOT | *wamr-cheri-aot.py* | On WSL2 Ubuntu  Linux\_x86\_64 | On Linux\_x86\_64,  Native compile |

This means that, by specifying one of three JSON files and one of two adapter Python files, it is possible to run 6 test cases on 3 platforms for only 1 WASM file!

### Additional Test Cases

The WASI Test Suite provides 35 input WASM test files, giving a total of 35 x 6 = 210 individual test cases. However a number of test cases were already developed during the project to manually test WAMR, initially for interpreter mode and later for AOT mode as well.

A number of these test cases were adapted to run under WASI Test Suite. The process to do this involved the following steps:

1. Eliminate any tests which could not be fully automated
2. Create WASI Test Suite JSON configuration files for tests as required
   * These are not the same as ready by the adapters; these supply arguments for tests such as expected exit code
3. Modify tests as needed to run correctly under WASI Test Suite
   * This involved making sure only automatically detected entry functions were used, such as *main(),* as WASI Test Suite cannot pass a “name of function to run” argument to WAMR
   * Also eliminating any *stdout* or *stderr* which was difficult to specify in the test config, such as excess log points with variable timestamps
4. Formulate tests into test suites and structure as for the WASI Test Suite’s own tests

#### Tests which cannot be Automated

Out of all of the available manual tests, **15** could be automated. After eliminating tests which were doing the same thing in all but name then there were a further **7** which could not be automated.

This is because:

* These are client/server network tests, which require a client to be running at the same time as a test server or vice versa. WASI test suite does not permit the test runner to do anything other than start WAMR in a new process and wait for it to complete
* The tests require a native library to be loaded as they interact with native code (e.g WASM🡪native or WASM🡪native🡪WASM). WASI test suite has no option to be able to locate and inform WAMR to load a native library
* They are external references tests, which require the bespoke extern references application to be used instead of *iwasm.* WASI test suite cannot support bespoke or multiple applications with a different set of command arguments.

Limitations of WASI Test Suite, then, prevent these additional tests from being automated. Potential improvements to WASI test suite described in Section 8.4 would permit these remaining test cases to also be automated.

#### CHERI Test Suite

The CHERI test suite comprised the basic “hello world” tests, the conditional block tests and additional WASI tests.

The test suite folder was added to *verifoxx-cheri-wamr* and can be found here: <https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/tests/wamr-linux-cheri-purecap-tests/wasi-testsuite/tests/cheri/testsuite>

This test suite can be run directly from WASI Test Suite with a suitable adapter and JSON file. For example, here are the commands needed to run the test suite on Linux (assuming the file system is structured accordingly and that the JSON settings file is set up correctly):

**export WAMR\_CHERI\_ADAPTER\_JSON=wamr\_cheri\_adapter.json**

**python3 ./test-runner/wasi\_test\_runner.py -t ./tests/cheri/testsuite -r ~/verifoxx-cheri-wamr/tests/wamr-linux-cheri-purecap-tests/wasi-testsuite/wamr-cheri.py**

**python3 ./test-runner/wasi\_test\_runner.py -t ./tests/cheri/testsuite -r ~/verifoxx-cheri-wamr/tests/wamr-linux-cheri-purecap-tests/wasi-testsuite/wamr-cheri-aot.py**

#### CHERI Net Test Suite

The CHERI Net test suite was for any tests requiring network access. These tests use a JSON settings file that includes the *allowres* and *addrpool* settings that will pass the extra networking permissions to WAMR on the command line.

Two tests were added: The *https\_client.wasm* which performs an HTTPS request to an internet server, and *wasi\_tcp\_dns\_lookup.wasm* which resolves a domain name to an IP address. Unfortunately the TLS client/server tests are not suitable because they require an accompanying peer to be running at the same time, which is not possible under current WASI Test Suite operations.

The CHERI net test suite can be found here: <https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr/tree/main/tests/wamr-linux-cheri-purecap-tests/wasi-testsuite/tests/cheri-net/testsuite>

The tests in the test suite can be run, again assuming suitable file system layout and JSON settings, as follows from Linux:

**export WAMR\_CHERI\_ADAPTER\_JSON=wamr\_cheri\_adapter\_networking.json**

**python3 ./test-runner/wasi\_test\_runner.py -t ./tests/cheri-net/testsuite -r ~/verifoxx-cheri-wamr/tests/wamr-linux-cheri-purecap-tests/wasi-testsuite/wamr-cheri.py**

**python3 ./test-runner/wasi\_test\_runner.py -t ./tests/cheri-net/testsuite -r ~/verifoxx-cheri-wamr/tests/wamr-linux-cheri-purecap-tests/wasi-testsuite/wamr-cheri-aot.py**

### Automated Testing Results

A total of 48 individual WASM test vectors are available for automated testing, giving 288 separate test cases.

The results for each platform are shown below:

| Target | WAMR Mode | Passed | Failed | Notes |
| --- | --- | --- | --- | --- |
| Morello pure-cap | Interp | 45 of 48 | 3 of 48 | See below re. failures |
| AOT | 48 of 48 | 0 of 48 |  |
| Morello Hybrid | Interp | 45 of 48 | 3 of 48 | See below re. failures |
| AOT | 47 of 48 | 1 of 48 | Rust test *fd\_filestat\_get* fails – see below |
| Linux x86\_64 | Interp | 45 of 48 | 3 of 48 | See below re. failures |
| AOT | 45 of 48 | 3 of 48 | See below re. failures |

**OVERALL RESULTS**

| Total Tests | Run (Automated) | Pass | Fail | Coverage (%) | Pass Rate (%) |
| --- | --- | --- | --- | --- | --- |
| 330 | 288 | **275** | 13 | **87%** | **95%** |

***Note: “****Total tests” includes a potential 7 x 6 = 42 tests which cannot yet be automated.*

**KNOWN FAILURES**

Three WASI Rust Tests cause problems on all platforms. At the time of writing, and at the least for interpreter mode, these also fail on *ByteCodeAlliance* WAMR last release. The questionable tests are:

* *close\_preopen* : Attempts to close file no. 3 (i.e stderr + 1) which wasn't opened.  Causes an exception - **also fails in the same way on wasmtime** therefore the test appears to be flawed.
* *fd\_filestat\_get:*Stat's *stdin* and expects all values to be 0.  In our *glibc*, the "last accessed time" is not defined and set to a random integer which isn't zero.
  + Given *stdin* is not a file, seems reasonable to not expect meaningful results
* *interesting\_paths:*Attempts to open a path which cannot be resolved and doesn't exist.  Expects return code "ERRNO\_PERM" which means "not permitted" but WAMR returns "NOT\_CAPABLE" which means "capabilities insufficient" which seems correct in the WAMR case

Of these three tests:

* + All three fail for interpreter mode on all three target platforms
  + All three fail for AOT mode on Linux x86\_64
  + One (*fd\_filestat\_get*) fails for AOT mode on Morello hybrid-cap
  + None fail for AOT mode on Morello pure-cap

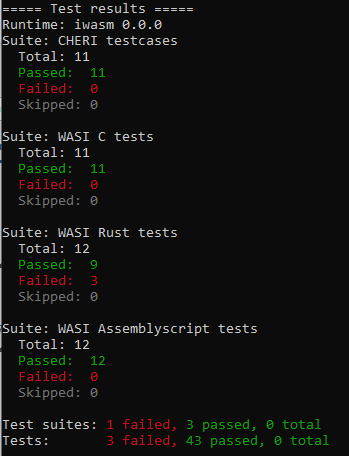
It is unclear why some or all of the above three tests pass for AOT mode on Morello platforms. It is suspected this is due to different compilation for AArch64 / Morello / x86\_64, but this needs further investigation.

**NOTE: There is a ticket raised in Verifoxx’s issues repository to further investigate these anomalies between the different build targets.**

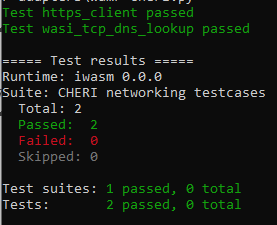
Screenshots showing run results are given below (note: Due to the need to use a different JSON settings file, the *cheri-net* is performed in its own run).

#### Morello Pure-cap Interpreter Mode

All apart from CHERI-net:

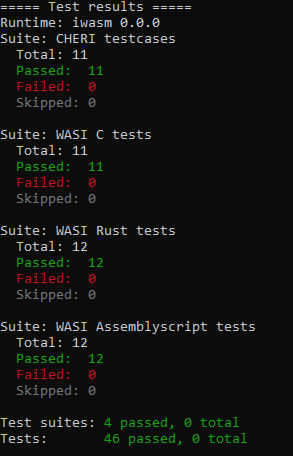


CHERI-net:

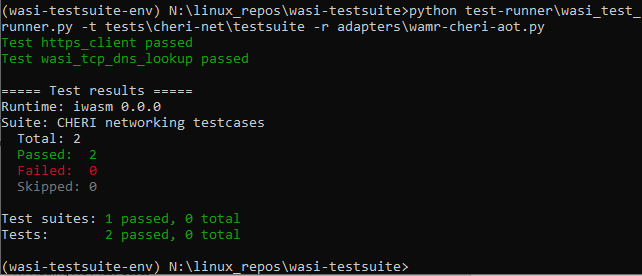


#### Morello Pure-cap AOT Mode

All apart from CHERI-net:

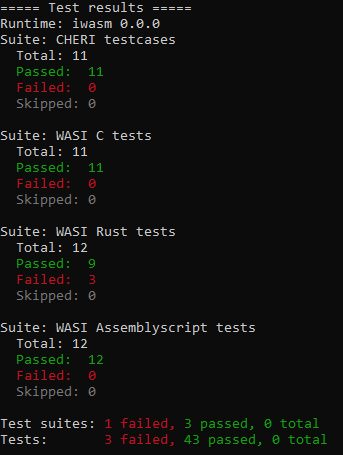


CHERI-net:

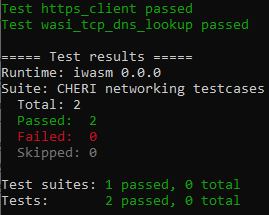


#### Morello Hybrid-cap Interpreter Mode

All apart from CHERI-net:

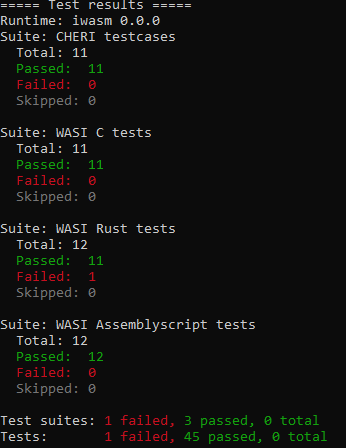


CHERI-Net:

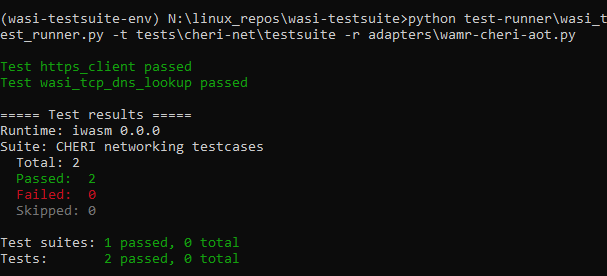


#### Morello Hybrid-cap AOT Mode

All apart from CHERI-Net:



CHERI-Net:

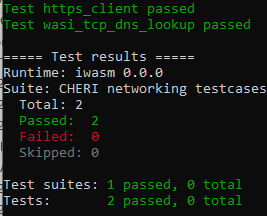


#### Linux x86\_64 Interpreter Mode

All apart from CHERI-Net:

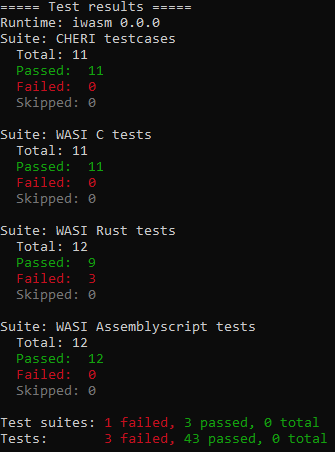


CHERI-Net:

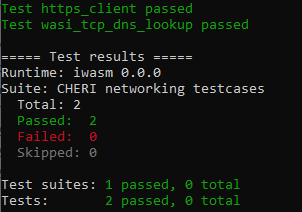


#### Linux x86\_64 AOT Mode

All apart from CHERI-Net:



CHERI-Net:



## Potential Future Testing Improvements

The WASI Test Suite is useful in enabling automated testing, but it has its problems:

1. It is not possible to repeat the automated tests with multiple adapters, as a single execution batch run
2. Other than *--dir*, no other WAMR arguments can be passed so it is not easy to test WASM which performs networking or which loads a native library
3. Due to the fact each test case is run completely independently there is no ability to configure a “once per test suite” setup as can, for example, be done for Python’s *unittest*
4. As the adapter is always run in a new shell then it is hard to pass additional configuration to the adapter, especially if this needs to differ between different test cases
5. The test run is “passive” in that, for example, a test network server cannot be run which can exercise a WASM network client that is under test
6. The presentation of results could be improved, particularly in recording a summary of passed/failed tests for each batch
7. There is no information on time taken to execute each test

Potential future “next steps” for testing would be to address some of these concerns by modifying WASI Test Suite, and then to develop a fully automated test framework using e.g *pytest* along with some of the best aspects of WASI Test Suite.

A first step could be to add the following to WASI Test Suite, which should be reasonably simple to do:

* Ability to specify other WAMR arguments from the test case JSON files
* Better presentation of results
* Ability to run test suites on multiple architectures as a single batch (i.e run multiple adapters / adapter config in the batch)
* Add timing information to the test case to record time spent running each WAMR test

Remaining aspects would be best handled by creating a whole new test framework, which could then be added as the test stage of a CI process. *ByteCodeAlliance* WAMR is already set up to run tests on github, so it would be useful if these could be applied for the CHERI versions of WAMR and extended to incorporate tests described here.

Additionally, although there are a number of test cases it would be useful to have more *targeted* testing – specific test cases that exercise parts of the code which are known to have been modified for CHERI. This may involve introducing white-box testing rather than just the system testing framework used by WASI Test Suite, and therefore is likely to be a longer-term activity requiring dedicated SW test developers.

# Benchmarking

## WAMR Benchmarks

The WAMR developers make available a number of common benchmarks to profile WAMR’s performance on the different architectures that are supported. The approach used by WAMR is:

* Build the benchmark from readily available source code to a native application
* Compile source to a WASM application using *clang* with WASI support
* Use *wamrc* to compile WASM to AOT
* Execute WASM and AOT with WAMR and compare results with native application

The benchmarks that are made available and which are of interest are listed below:

| Benchmark | Link | Description |
| --- | --- | --- |
| PolyBench | <https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1> | Benchmark suite of different numerical operations |
| CoreMark | <https://www.eembc.org/coremark> | Simple benchmark to test different processor cores |
| Sightglass | <https://github.com/bytecodealliance/sightglass> | Benchmark suite intended to test WASM applications |
| Dhrystone | <https://github.com/bytecodealliance/wasm-micro-runtime/tree/main/tests/benchmarks/dhrystone> | Longstanding benchmark for integer operations of a processor; WAMR implements a specific version (see link) |

## WAMR Resource Usage

Benchmarking, and this part of the document, is all about an analysis of the performance of WAMR for CHERI vs non-CHERI however it is also appropriate to consider the other aspect that should be compared: resource usage.

The code size of WAMR is known and is the same for any WASM or AOT application. We discover WAMR has a 30% higher code (and initialized data) footprint for CHERI pure-cap vs hybrid. This is to be expected as additional instructions are needed to manage capabilities versus non-CHERI pointers.

In terms of the data memory footprint, WAMR is fairly consistent for any WASM application because it always creates the same data structures. Stack and heap usage will vary and depend on the WASM application, stack being more variable since the WASM heap size is declared within the WASM script. We would though expect higher usage for CHERI because we need to at times store capabilities rather than non-CHERI pointers.

*Note: The WAMR port introduced the use of the system heap for CHERI data structures as part of initial work to improve the double-sandboxing and support capabilities. This is not considered as an additional resource usage for CHERI vs non-CHERI because the change was applied for both hybrid builds and pure-cap builds of CHERI-WAMR.*

An analysis of the data footprint reveals that it is between 10% and 20% higher for CHERI vs non-CHERI.

When considering the additional resource usage for CHERI we would not consider it overly high. Certainly there is an expected trade-off; nothing comes for free, and increased security brings with it higher resource usage (and slower performance as shall be discussed further in this section). However, especially when running on high-level operating systems and with modern cheap memories being available, the cost of the additional footprint would seem cheap.

## CHERI-WAMR Benchmarking Setup

The objective for the CHERI-WAMR port is to compare the performance of applying CHERI to the performance of not having CHERI extensions. To this end, and given the benchmarks effectively produce a total execution time metric, it is necessary to run the benchmarks on the Morello box both with and without CHERI. Therefore we produce a *hybrid-cap* and a *pure-cap* version of WAMR which will be used to execute the WASM application.

We are primarily interested in execution time rather than code size. We know that purecap applications naturally will require a larger code size since any read-only pointers are twice the size and manipulating pointers requires additional Aarch64 instructions. However we do also consider and analyse the size of the WAMR executable and the size of the compiled AOT. Note that:

* AOT is processor-specific, and therefore compiling AOT for different processors produces a different file size
* A WASM application is (obviously) independent of the interpreting WAMR

### WAMR Applications

In order to fairly evaluate performance, both hybrid and pure-cap WAMR applications must be compiled with the same options. We here prefer performance over compile size. The options (WAMR and compiler are as follows):

* Fast Interpreter chosen over classic (*WAMR\_BUILD\_FAST\_INTERP=1*)
* WAMR Debugging, memory tracing and profiling tracing is disabled (but see below)
* Release not debug mode
* Compiler optimise for speed (*-O3* option)

A hybrid executable and purecap executable are then produced to execute all tests.

The benchmarks generate a metric to *stdout* which is either the total time taken, or time-per-operation or operations-per-second from which a metric can be obtained. We can then, for each benchmark, compare hybrid to purecap and determine a percentage difference between them – e.g we can say “purecap is 110% the execution time of hybrid”, which means purecap takes as long as hybrid and 10% of hybrid’s time longer.

### Benchmarking under an Operating System

We run under Morello Linux, which means benchmarking results will not be the same as running on baremetal. It is therefore difficult to get a like-for-like comparison due to process context-switching, caching and similar concepts.

To mitigate against this, all benchmarks run five times and an average result is taken. As the benchmarks are long-running programs this also serves to mitigate against OS interference.

### WAMR Internal Profiling

WAMR does have the ability to internally profile a WASM application, which is to record the time each function takes to execute. This is of limited use because each function in a benchmark application executes multiple times, however for CHERI (hybrid and purecap) we have extended the performance profiling to additionally record:

* Total time spent in initializing and loading the WASM application at the start
* Total time spent in system calls

These were analysed when running the benchmarks but the measurements were negligible. No system calls are made during benchmarking operations, when the bulk of execution occurs, and the benchmarks run so long that the setup time is not significant.

Note that the profiling also adds an overhead, so profiling was disabled for the benchmark runs that record the results.

### Use of Benchmark ABI

There are certain known issues with the Morello core which lead to longer execution time for certain sequences of instructions. To mitigate this, an ABI implementation has been developed for benchmarking purposes that will perform the affected operations as hybrid Aarch64 operations instead of CHERI. This is fully documented in <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-986.pdf>. Although seemingly primarily intended for CHERIBsD the ABI is also functional on Linux.

For the WAMR benchmarking work we used the normal ABI and the benchmark ABI to compare the two. Then, the benchmarking ABI was used to run the various different tests to obtain definitive results.

### Toolchains

We used both LLVM and GNU Toolchains in the benchmarking, and compared the performance of the two. For the comparison between Hybrid and PureCap we attempted to find the toolchain which gave the best performance on both, and then compared these.

Note though that the Benchmark ABI was only available for the LLVM Toolchain, and secondly note that the LLVM PureCap Toolchain is more readily supported and in a more advanced stage of development than the GNU version.

LLVM was used with MUSL Libc; GNU Toolchain with the GNU Libc (Glibc).

## CHERI-WAMR Benchmark Execution

The polybench and sightglass benchmarks generate a different application for each test; there are some 46 benchmark applications in total and each one needs to be executed a total of 4 times (WASM on hybrid, WASM on purecap, AOT on hybrid, AOT on purecap).

Therefore an automated mechanism was needed to run each of the benchmarks and get the results.

To support this a Python framework was developed which will discover benchmark applications, execute each one on both hybrid and purecap *N* times, parse *stdout* to obtain time metrics and then collate results. Finally, overall results and averages are deduced and presented for human consumption as a CSV file.

The python test framework uses a similar mechanism to the WASI testsuite in that it remotely connects to the Morello machine, executes the WAMR application, and transfers stdout back for processing. This both minimises the additional processing occurring on Morello while the benchmark runs, and reduces the need to extract results independently from the Morello box when test execution completes.

### Building the Executables under Test

Our primary focus is on WAMR executing WASM applications, but we are also interested in AOT compiled applications and native applications as a “best case” comparison.

Exactly what needs to be built for each of the execution cases is different, as is described here>

WASM Application

A WASM application is the same for any target. Therefore for testing a WASM application it is necessary to:

* Compile WAMR code with the toolchain being used
* Compile native library code with the toolchain being used
* Link to create the static *iwasm* application: This is then the same for every test

AOT

The AOT compiler runs on x86\_64 and therefore it is unaffected by the choice of toolchain. However the compilation of the AOT application requires using a suitable LLVM Backend.

* Note: It was not possible to obtain a suitable LLVM Backend for the Benchmark ABI; this was not supported by any toolchain

Each AOT application is compiled separately to a dedicated, target specific AOT file therefore there would be one AOT file for Hybrid and one for Purecap.

Additionally, AOT also requires the WAMR executable as described above

Native

A native application is self-contained. Therefore for each benchmark test case an executable is needed for hybrid and another for purecap.

The Native application, like WAMR, requires:

* Compilation of all source files with the correct toolchain
* Compilation of any library code with the toolchain
* Linking together to create the final native executable

#### Building for the Benchmark ABI

The benchmark ABI is only supported on LLVM therefore the compilation uses clang. To build using the benchmark ABI it is simply necessary to specify a different ABI.

The normal options to build for Morello purecap on clang with a MUSL libC would be as follows:

clang --march=morello+c64 --mabi=purecap --target=aarch64-unknown-linux-musl\_purecap

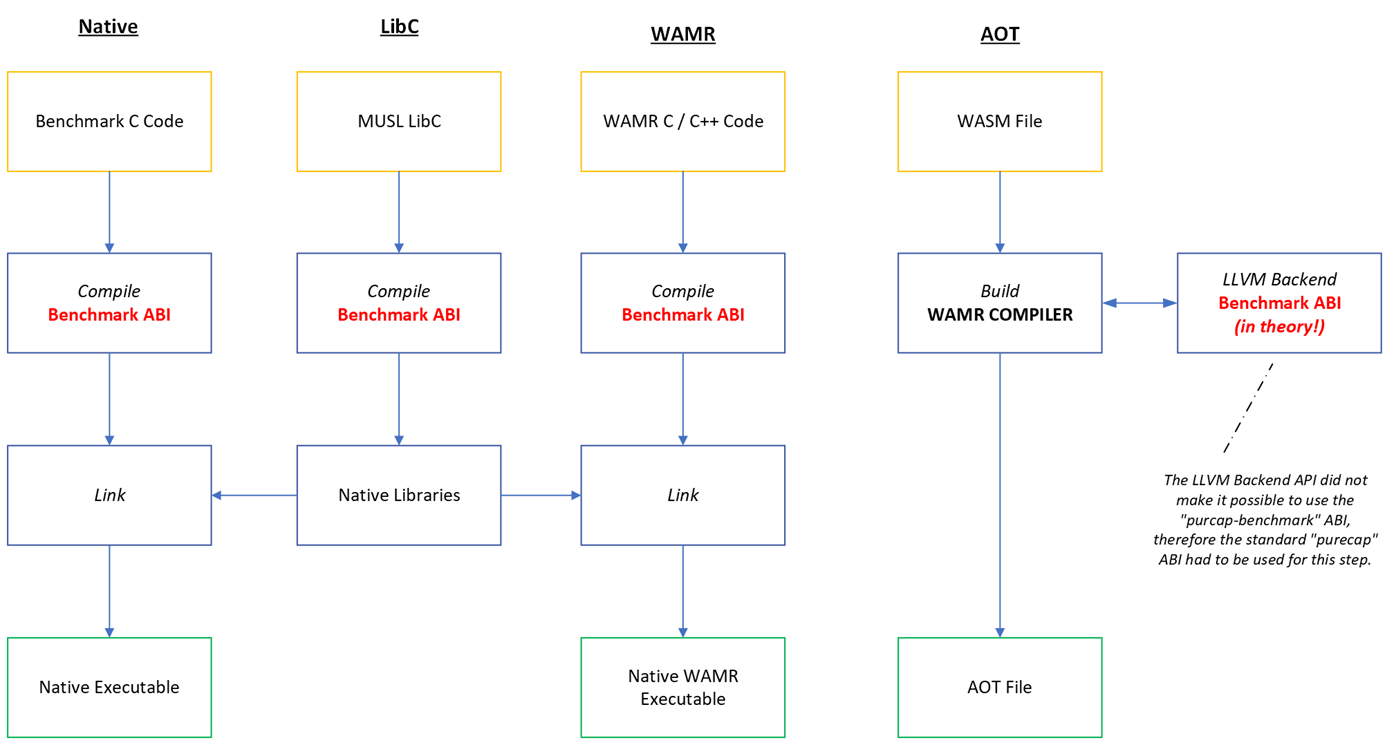
For the benchmark ABI it is:

clang --march=morello+c64 --mabi=**purecap-benchmark** --target=aarch64-unknown-linux-musl\_purecap

As explained above, various different components need to be compiled for the Benchmark ABI and it is necessary to rebuild MUSL Libc using the modified *clang* command shown.

Also as mentioned it was not possible to compile AOT applications for the benchmark ABI as a means of modifying the LLVM backend to support it was not found.

The diagram below shows the compilation flow for the Benchmark ABI:



### CMake and Visual Studio Support for Native Benchmark Applications

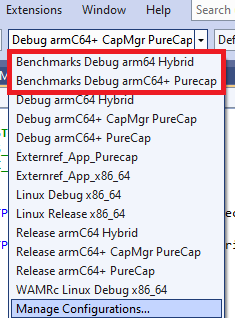
The benchmarks supplied with WAMR are – in almost all cases – just a set of shell scripts which will download the appropriate benchmark from a remote git repository and then build it with the correct options. (The exception is dhrystone, which consists of just two C files and only a single benchmark). This saves considerable time since the polybench and sightglass benchmarks rely on the same git codebase to generate a number of individual benchmark tests.

For automation purposes the benchmarks can be built in a similar way, with scripts, but it was also found to be useful to be able to build the native applications separately under an IDE with debug support so that individual benchmarks could be debugged (e.g when they were giving a *Segmentation Violation* under CHERI).

* Note: Debugging of the WASM applications can be achieved by simply debugging WAMR, so only the native applications needed to be handled in CMake

To this end, *CMake* scripts were produced which can then be used with Visual Studio. Benchmark configurations were then added to the existing Visual Studio list of build configurations – as this is just updating a *CMakePresets.json* file then these can also be used standalone from Windows or Linux to build on a Linux machine capable of cross-compilation to Aarch64 and Morello.

The full list of Visual Studio *CMake* build configurations for WAMR are shown below – the Benchmark additions are highlighted. Note that only Debug configurations are produced for the benchmarks as the purpose of standalone builds is only for debugging purposes:



The CMake scripts that have been developed perform the following operations:

1. Download the coremark, polybench and sightglass git repositories onto the Linux build machine
2. For each of the benchmark types a list of target benchmarks is then defined
3. For each individual benchmark test in the list:
   * Declare an executable and needed source files
   * Set compilation options
   * Compile and link the executable
4. In the case of Polybench, an inner loop is also needed to enumerate a test-specific folder, as polybench files are different for each test

Due to the need for more recent CMake functions, version 3.20 or later of CMake is required. A snippet of the CMake function to download a git repository, and an example of its use, is shown below:

include(FetchContent)

function(CloneRepository repositoryURL branchname projectName sourceDir)

message(STATUS "Clone ${projectName} from ${repositoryURL} into ${sourceDir}")

FetchContent\_Declare(

"${projectName}"

GIT\_REPOSITORY "${repositoryURL}"

GIT\_TAG "origin/${branchname}"

SOURCE\_DIR "${sourceDir}"

CONFIGURE\_COMMAND ""

BUILD\_COMMAND ""

INSTALL\_COMMAND ""

)

FetchContent\_MakeAvailable(${projectName})

endfunction(CloneRepository)

# Usage: Install from git repos

CloneRepository("https://github.com/eembc/coremark.git" "main" "Coremark" ${COREMARK\_DIR})

CloneRepository("https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1.git" "master" "Polybench" ${POLYBENCH\_DIR})

CloneRepository("https://github.com/wasm-micro-runtime/sightglass.git" "main" "Sightglass" ${SIGHTGLASS\_DIR})

### Test Framework Details

The benchmark automated testing framework comprises python scripts which enumerate all benchmark tests and then each one is built, run and its results are gathered.

Each automated benchmark test stage requires inputs; the inputs are those files which are built as part of the existing WAMR benchmark support. These are:

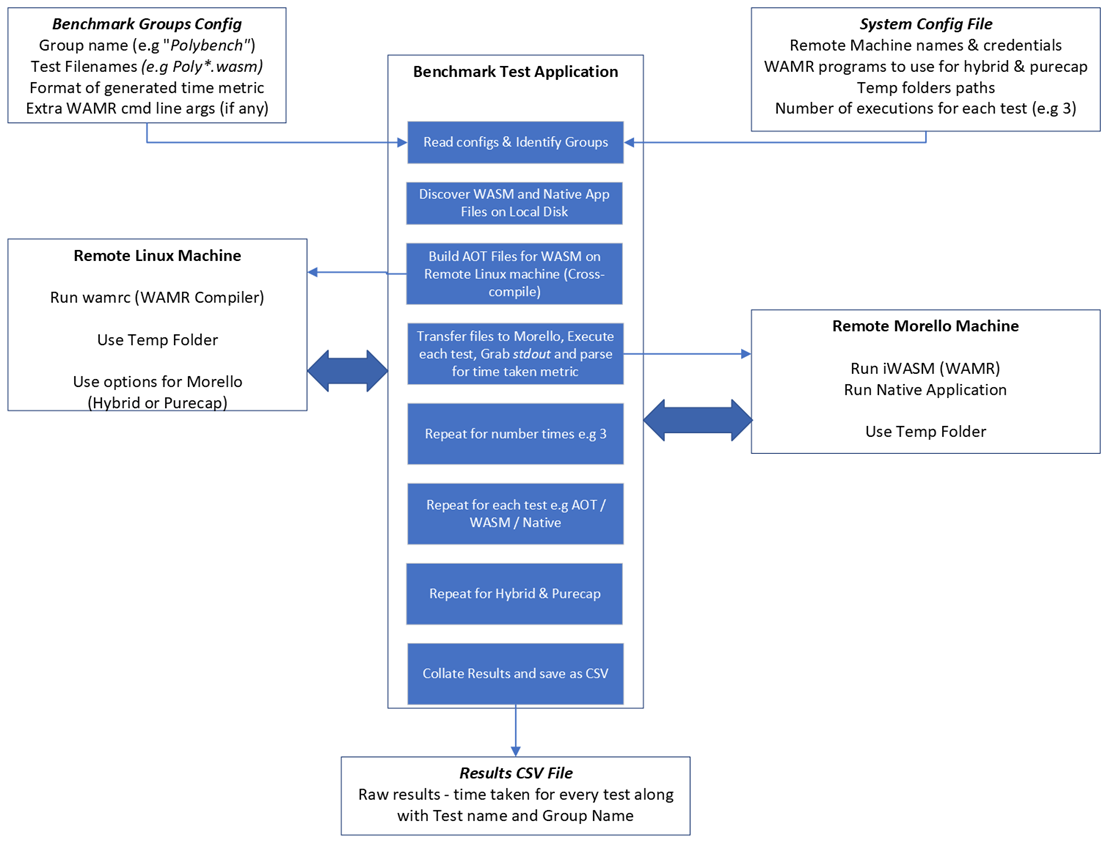
* The WASM application
* The native application
* The WAMR executable i.e *iwasm*

Note that WAMR is the same for every test, so only needs to be built once for the toolchain / ABI combination.

The AOT file is built for each test by the automated test environment, this can be done locally (if running under Linux) or on a separate Linux machine (e.g an x86\_64 machine which is able to cross-compile AOT for Morello Hybrid & Purecap).

The tests are then automatically executed remotely by remote shell into the Morello box.

The full flow of the test environment is shown below, followed by explanation of the execution and results gathering process:



#### Configuration and Setup

The automated benchmark tester is itself a python application which can be run on Windows or Linux and which takes a number of command line arguments. These inform the folder to find test cases, the output file to generate with results and various minor options to tweak the execution run.

The application reads configuration from a JSON file which provides all needed details to be able to run remote execution on the Morello box, and optionally on a Linux machine which can build AOT applications.

The automated tester discovers tests automatically. All tests must be appropriately named to start with the name of the test type (e.g *polybench-2mm.wasm)* and all must exist in the same folder. The test program will scan this folder and discover *.wasm* files; it will then attempt to locate a native files with the same name prefix. Using the above as an example:

WASM File: *polybench-2mm.wasm*

Found under test type: *Polybench*

Name to be used for individual test: *Polybench-2mm*

Searched name for native applications: *polybench-2mm\_native-hybrid* & *polybench-2mm\_native-purecap*

Generated names of AOT files: *polybench-2mm-hybrid.aot* & *polybench-2mm-purecap.aot*

#### Collections and Runs

Each test group is gathered into a “collection” which is a conceptual concept used to identify one of the test groups. The system can generate collections dynamically based on a supplied list; for each collection the list of individual test cases is specified. In line with the benchmark types we support the collections are:

* CoreMark
* Dhrystone
* Polybench
* Sightglass

For each collection each individual test is run for every supported combination, namely:

* WASM Application
  1. Hybrid
  2. Purecap
* AOT
  1. Hybrid
  2. Purecap
* Native Executable
  1. Hybrid
  2. Purecap

Every individual test (e.g a WASM-Hybrid for a single polybench test) can be repeated a given number of times, which is one of the configuration settings that can be supplied to the script. By repeating the tests and then taking the mean execution time value we reduce the impact of any short-lived and unexpected system processes interfering with test execution time. Unless otherwise stated, a repeat count of three was used for all our tests.

#### Gathering and Presenting Results

As is defined in the benchmark tests themselves, and as is implemented in the existing (non-CHERI) WAMR implementations, some benchmarks output their execution time taken in the *stdout* dump and others must be timed using the Linux *time* command. Each test collection defines polymorphic functions which allow the collection to identify whether it needs running under *time* or not. Further, each collection implements a function which is used to parse the *stdout* to extract the actual time taken so this can be presented to the test runner.

The test runner stores the time taken for every type for each test case (calculating the mean as applicable when *repeat\_count > 1*). On completing all the tests the results are gathered and then formatted as CSV and written to a CSV file on the disk of the machine the automated test system is being run from.

Note that the CSV Results are “raw” in that the individual execution times only are written. A separate processor is used to generate basic metrics from the results, this is a standalone python script.

#### Verifying the Automated Test Framework

Before running the tests, the test framework itself was first verified. To do this, a native application has been created which simulates WAMR (the *iwasm* program). This program is then named such that the test framework will behave as if it is the real *iwasm* program.

This verifier program detects which collection the input test is by scanning its filename, and then generates an output in the appropriate format for that test type. The output includes a simulated execution time as a random number.

This verifier program is written in C++ and is built with LLVM for both Morello Hybrid and Morello Purecap. The source code for this application is supplied bundled with the *autorun\_benchmark* tool in the *tests/* folder of the Verifoxx CHERI-WAMR github repository.

#### Parsing Test Results

The raw CSV that is written by the automated test runner can be parsed by a *CSV Processor* tool which generates another CSV that can better show the differences between hybrid and purecap.

For every type of every individual test case this tool will generate a measure which is the *percentage speed-up* of hybrid when compared to purecap. The assumption is that, all being equal, purecap will be slower than hybrid and so a positive number is the percentage faster that hybrid is. A negative percentage results when purecap was faster.

The percentage speed-up of hybrid vs purecap is calculated as:

*Where:*

H = Time in seconds for the Hybrid version to run

P = Time in seconds for the Purecap version to run

Each of the AOT, Native and WASM tests then have a percentage speedup calculation applied.

The parser can be controlled via the command line to provide the input and output CSV files. It can be instructed to present the results as an individual result for every specific test, or *summarised*.

In the summarised version then all the results for each category are gathered together so that the *mean value* for each category is calculated. This will then output:

* Speed-up for Native (%); Speed-up for AOT (%); Speed-up for WASM (%)

For each of:

* Coremark
* Dhrystone
* Polybench
* Sightglass

The summarised version, will, lead to precisely 12 percentages.

#### Precision

Internally, assuming the scripts are running on a 64-bit processor, the execution times and all CSV processing calculation results are stored as a 64-bit floating point number equivalent to IEEE-754 (or a C *double*).

The generated CSV values are rounded to 6 decimal places in the case of execution times and 2 decimal places in the case of percentage speed-up times.

#### Dealing with Missing Results

In some cases the benchmark is not able to run, specifically this applies to the Coremark native executable when built for Morello purecap. We are not interested in porting such native applications to Morello purecap, we are interested in replacing them with WAMR and so we have to simply record that no timing result was generated in this case.

Equally, the user may choose not to run native or AOT applications as part of the test and therefore these results would also be missing in the output.

Missing results are recorded as having an execution time of zero. This was preferred as it would still allow statistical calculations to be performed on the dataset without invalidating the results.

## CHERI-WAMR Benchmark Results

The automated test system was run a number of times in order to compare the performance of purecap vs hybrid under a number of different conditions.

**Important Notes applicable for all Results**

* All executables are built as *Release* versions (no debugging information)
* All are *optimised for performance* (-O3 optimisation flag in clang and gcc)
* WAMR was built for speed, using *Fast Interpreter Mode* (as opposed to *Classic Mode*)
* **All execution times are in seconds apart from Dhrystone which is in microseconds (since time per dhrystone is so short). However speed-up times are still valid as a percentage.**

The below present the results for each of the different comparisons run. Note that it is infeasible in this document to tabulate the results for every single test – there are too many. Instead, the source CSV is attached as a file.

### Hybrid Best vs Purecap Best (non-Benchmark ABI)

The GNU toolchain was found to give the fastest results for Hybrid – this is unsurprising as gcc and glibc are known to be fast. For purecap, LLVM gave the best results. Again unsurprising as the GNU Toolchain is still in a state of development for Morello.

The below table shows the execution times for Hybrid-with-GNU and Purecap-with-LLVM and then the percentage speed-up of hybrid over purecap:

A table with numbers and text

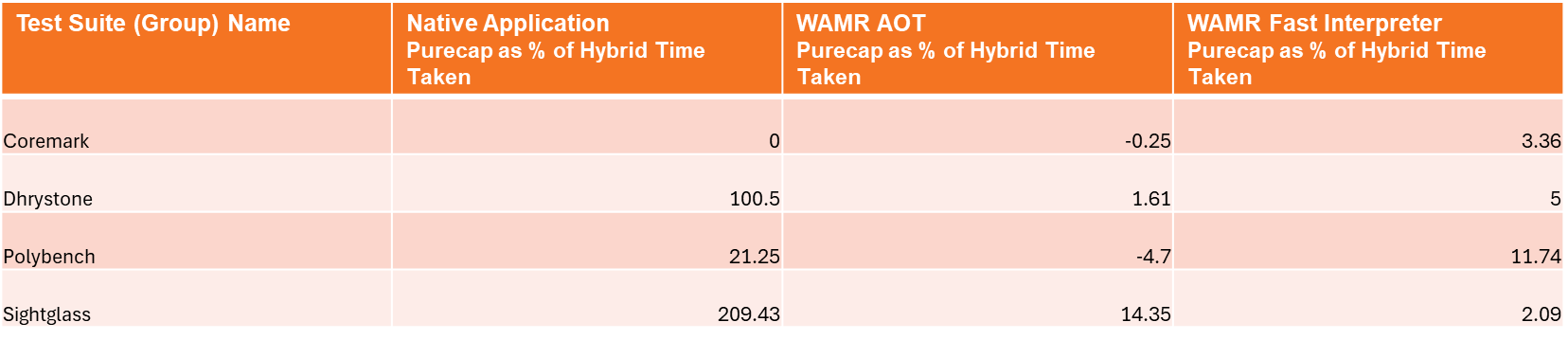
Description automatically generated

*Note native coremark application did not run on purecap*.

Full results attached:



Speed-up Percentages, summarised per category:



*Recalling the calculation metric: for example, 50% means that test takes 1.5 times as long on Purecap, 100% means it takes twice as long and -25% means purecap was actually only ¾ the hybrid time*

#### Analysis

These results showed that - pound-for-pound – hybrid would outperform purecap the majority of the time. In some cases purecap would be slightly quicker than hybrid, but in a significant number of cases purecap was significantly worse:

WASM

* Worst case: Purecap 24% slower than Hybrid
* Best case: Purecap 6% faster
* Average case: Purecap 6% slower
* Most frequent case, Purecap 10 – 14% slower

AOT

* Worse case: Purecap 63% slower than Hybrid
* Best case: Purecap 32% faster
* Average case: Purecap 16% slower
* Most frequent case: Purecap < 3% slower than Hybrid

Native (for observation purposes only)

* One native case (Coremark) crashed on Purecap
* Number of cases up to 8% faster on Purecap
* Majority of cases <20% slower on Purecap
* Significant number up to 100% slower (twice as slow)
* One case *(sightglass-memmove)* 1550% slower!

We would expect AOT to in general be better performing than WASM since it is predominantly object code, and therefore it would be similar to the native code.

We also noted that although in general the results were likely to be as expected, and similar to reported elsewhere for Morello (discussed more later in this document, with reference), there were a clear number of cases which were disappointingly and unexpectedly very poor.

### Testing the Benchmark ABI

The previous results suggested that there were some results which gave very poor purecap performance. We theorised that these were resulting from algorithms which were known to have issues on Morello, and therefore we ran the benchmarks again using the Benchmark ABI.

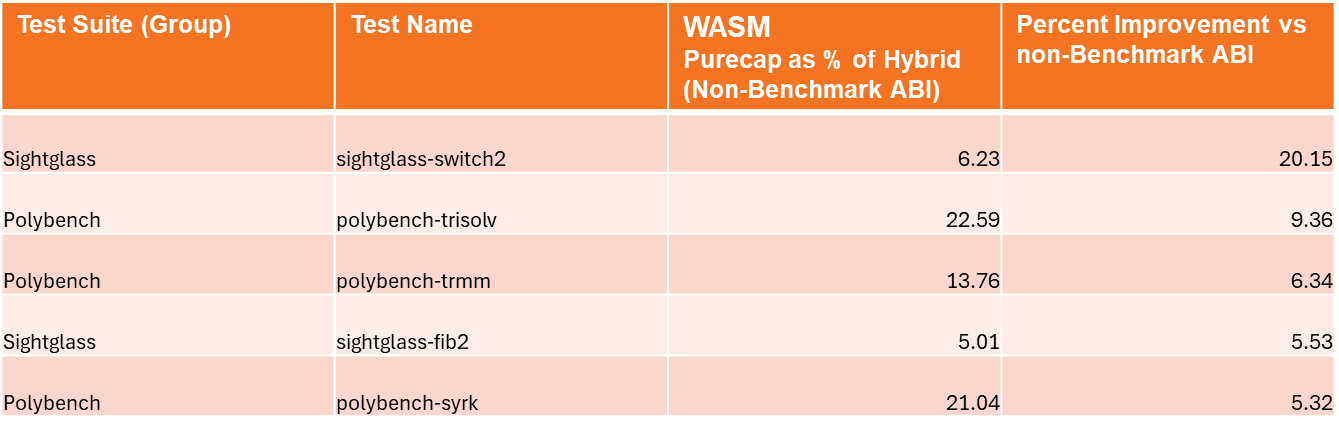
For these results we decided to ascertain how much better (or potentially worse) the benchmark ABI made our purecap tests.

**NOTE: As explained previously, it was not possible to compile AOT applications using the Benchmark ABI, hence any change in execution times would be solely due to improvements in the top-level parts of the WAMR application used to run AOT itself. This would give misleading data for AOT and therefore in the below discussion only WASM and Native applications are considered**.

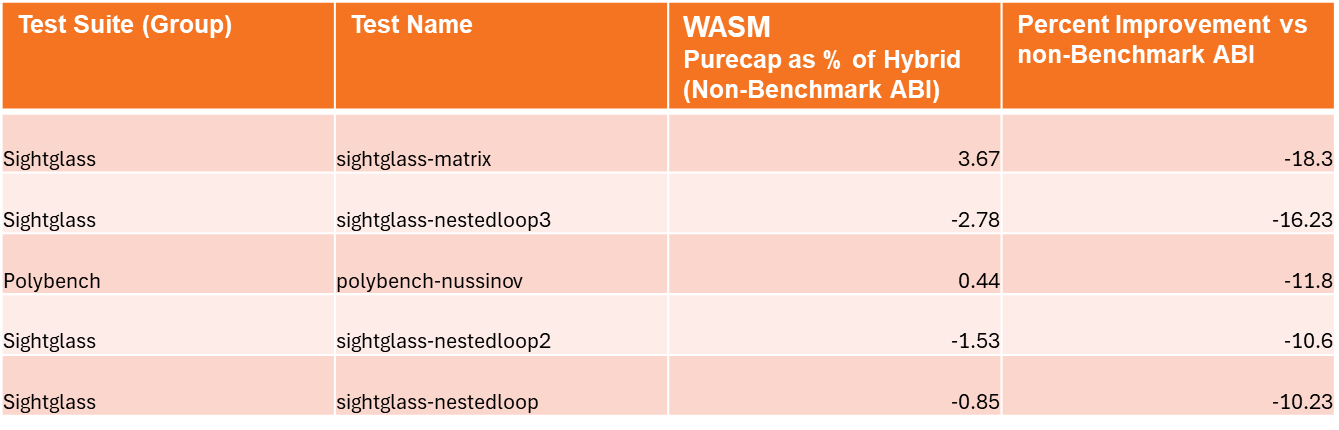
WASM

The tables below highlights the benchmark tests where the Benchmark ABI made the most difference, and where it made the least difference (or made things worse). Here, we show the hybrid-to-purecap speed-up as calculated *without* the Benchmark ABI and then show the improvement (or reduction) in the percentage speedup. For example, in the first example hybrid was about 6% faster than purecap but then improves by 20% such that purecap is now 14% faster than hybrid.

*Best Improving (Top-5):*



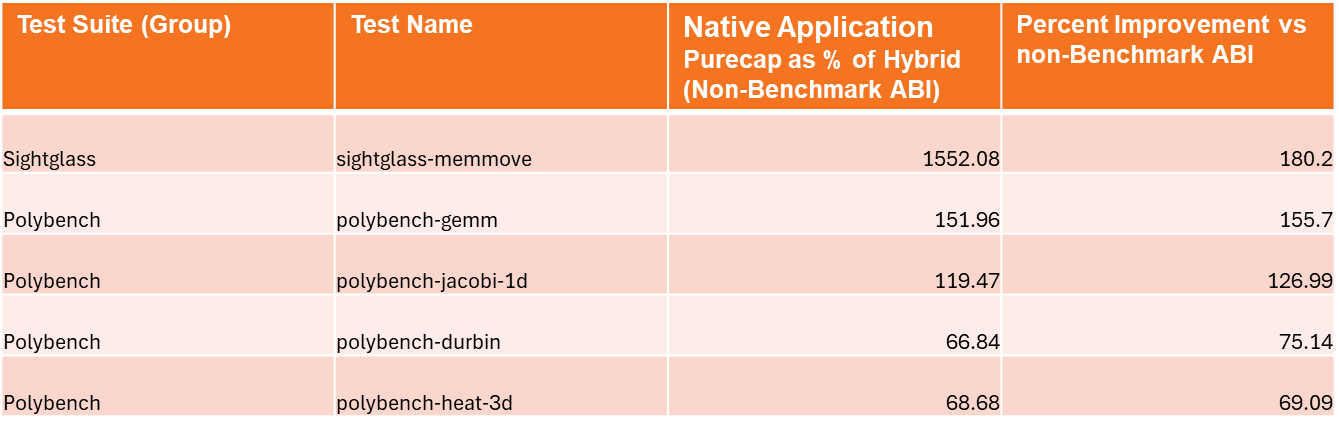
*Worst Improving (Bottom-5):*



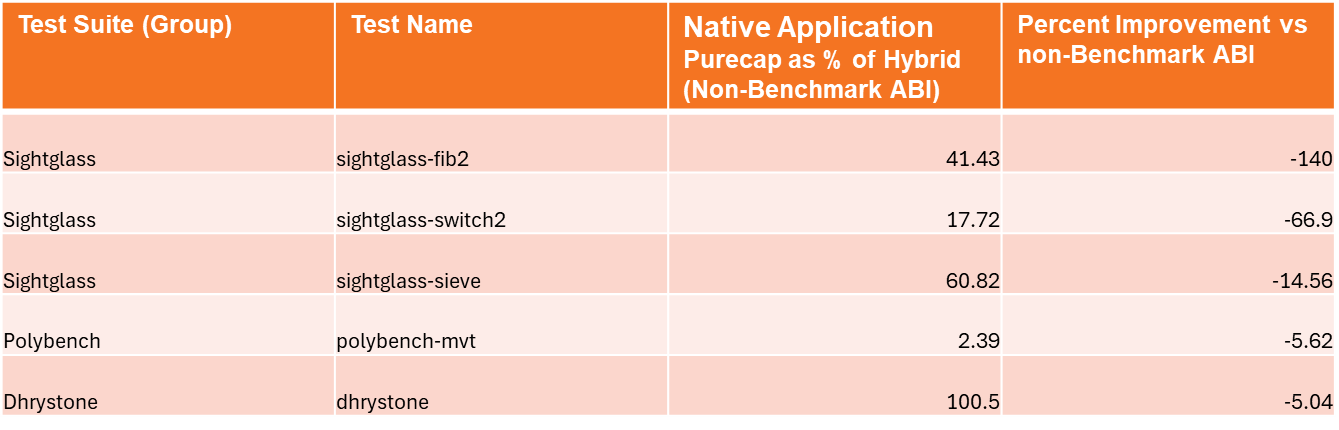
Native

The same thing is presented below for native applications:

*Best Improving (Top-5):*



*Worst Improving (Bottom-5):*



Full results for execution using the Benchmark ABI for purecap are attached below:



Here are speed-up percentage results for with and without the Benchmark ABI (all else being equal) and the improvement with the Benchmark ABI vs without. **NOTE:** AOT results do not have the AOT file compiled with the Benchmark ABI, therefore results are only a result of improvements in WAMR itself and may not be reflective of a true result:



#### Analysis

In general, the Benchmark ABI made improvements. We found that benchmark tests which were previously very poorly performing were, as a rule, made significantly better when using the Benchmark ABI (that is to say the difference between hybrid and purecap was reduced).

However, for tests that were already performing well (that is the purecap execution time was not much worse, or better, than hybrid) then there were negligible or no gains. In fact in some cases the results were actually worse with the benchmark ABI. This latter point was a little surprising.

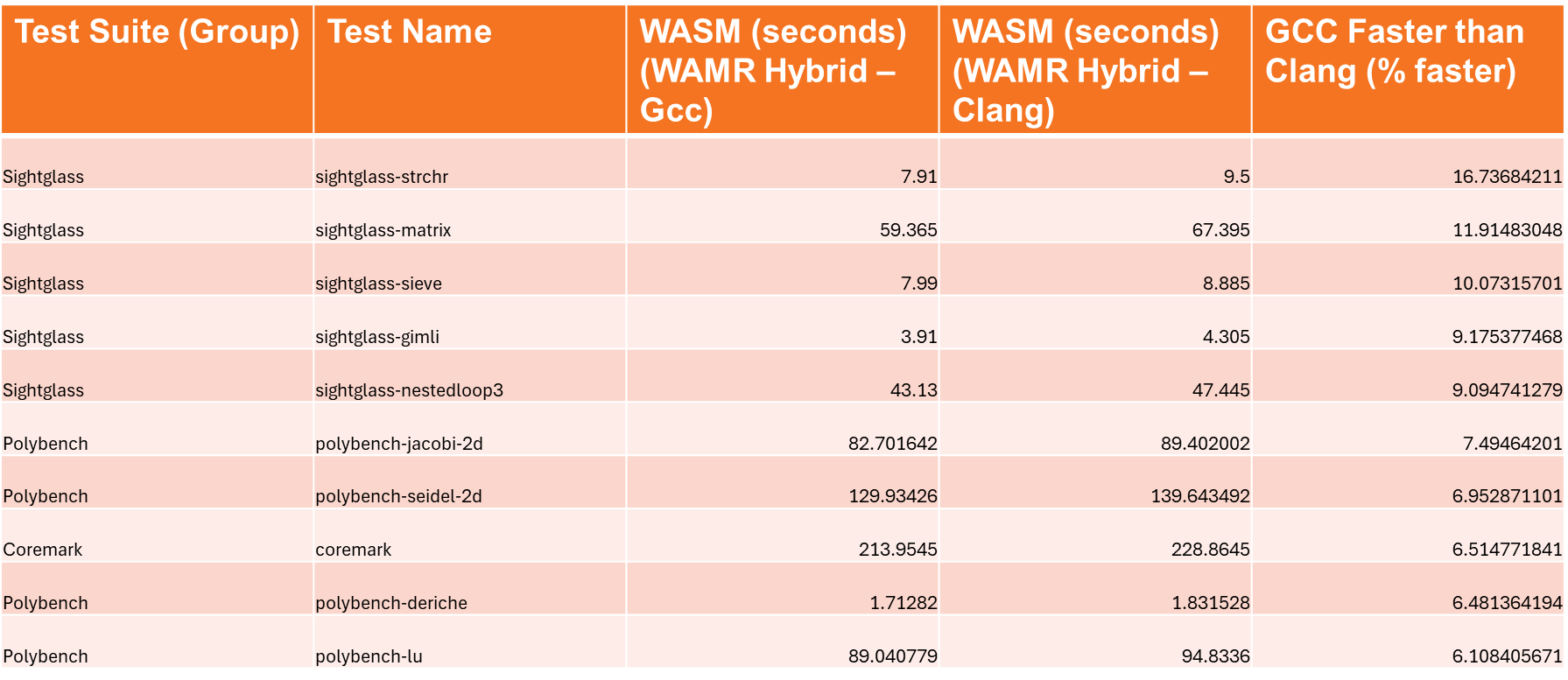
As would be expected, the Benchmark ABI gave most improvements on native executables and less of an improvement on WAMR. WAMR will be less optimal than native executables, is less impacted by libC changes and has been specifically engineered in the port to purecap so we would not expect overly significant gains. (Recall that no porting work was done on the native executables, they were simply compiled for purecap not hybrid).

### Toolchain Comparison: GCC vs Clang

As explained previously, the GNU Toolchain gave the best results for Hybrid and LLVM the best results for Purecap. In order to ascertain what affect the toolchain may be having, several different test runs were tried. The first of these, comparing GCC with clang on Hybrid, is described here.

We know that both GCC and Clang have long supported Aarch64, and therefore a like-for-like test with both running on Hybrid should be able to detect any significant and implicit differences between the two toolchains. It was suspected that GCC would be faster, and so we produced an analysis which showed how much faster GCC was than Clang as a percentage of execution time taken.

The table below gives the best ten tests for which GCC was faster than clang by the highest percentage:



#### Analysis

We can see from the above that for the majority of tests there was not a significant difference between the toolchains when both running on Hybrid. Given number 10 in the above list has GCC being 6% faster, then in the rest of the tests the improvement was an even lower percentage.

We do though note that in three tests GCC was over 10% faster than clang. These three sightglass tests are also those which happened to give particularly bad performance on purecap, so it suggests that the choice of toolchain is relevant but not the only factor.

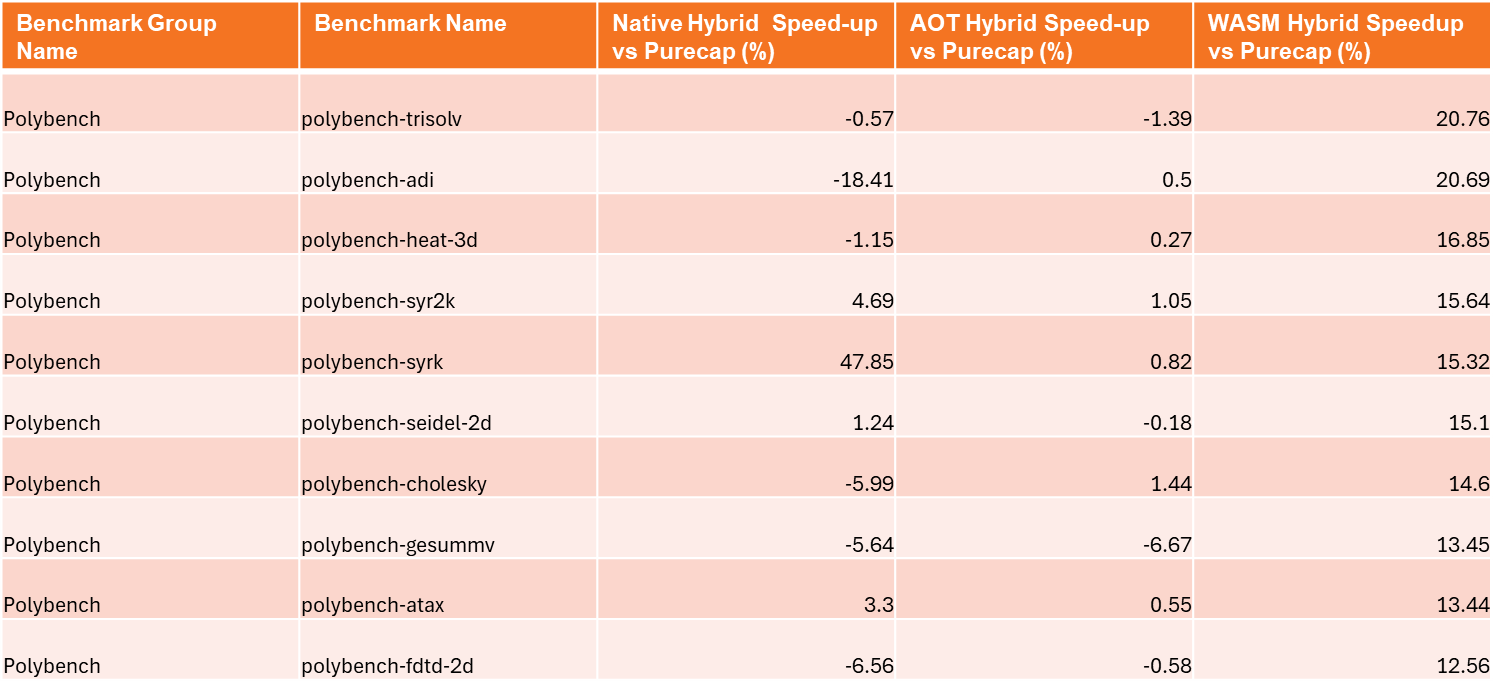
### Toolchain Comparison: LLVM Like-for-Like

For a second toolchain test, to compare “like for like” it was decided to use LLVM for both hybrid and purecap, and to use the Benchmark ABI for best results. Since the Benchmark ABI is only supported on LLVM this meant LLVM had to be used - in any case, LLVM is the main target toolchain port for Morello and GNU is much less supported (at the time of writing).

We were here also particularly interested in whether the few extremely poor results we had for purecap performance on native executables could be entirely explained by not using GCC: if both hybrid and purecap give similar results with LLVM then this would explain the anomalies.

WASM

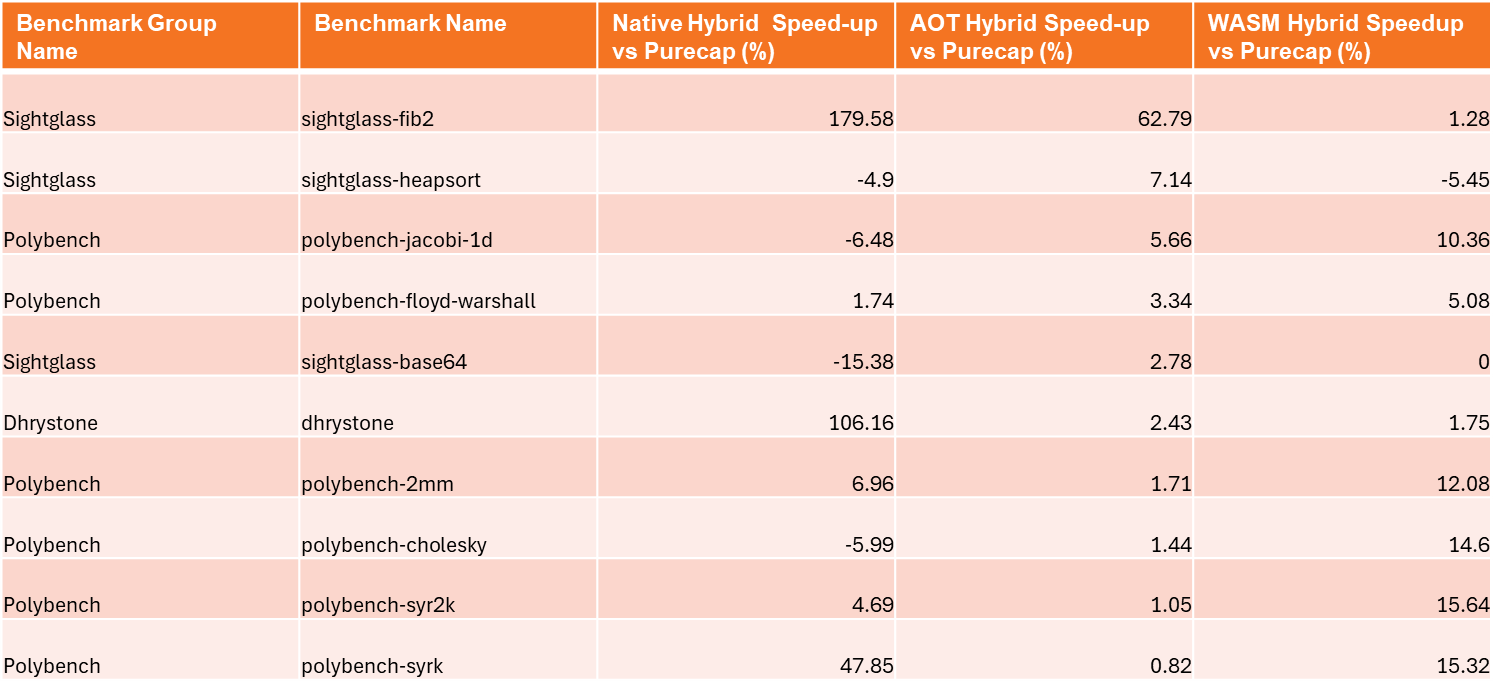
The table below shows the 10 worst results for WASM Applications; this is where Purecap was most reduced in performance compared to Hybrid:



It can be seen from the above that the majority of tests were less than 12% faster when using hybrid, and the worst were no more than 21% fast. In fact for most cases the difference was well into single figures and in some cases up to 20% better when using purecap.

AOT

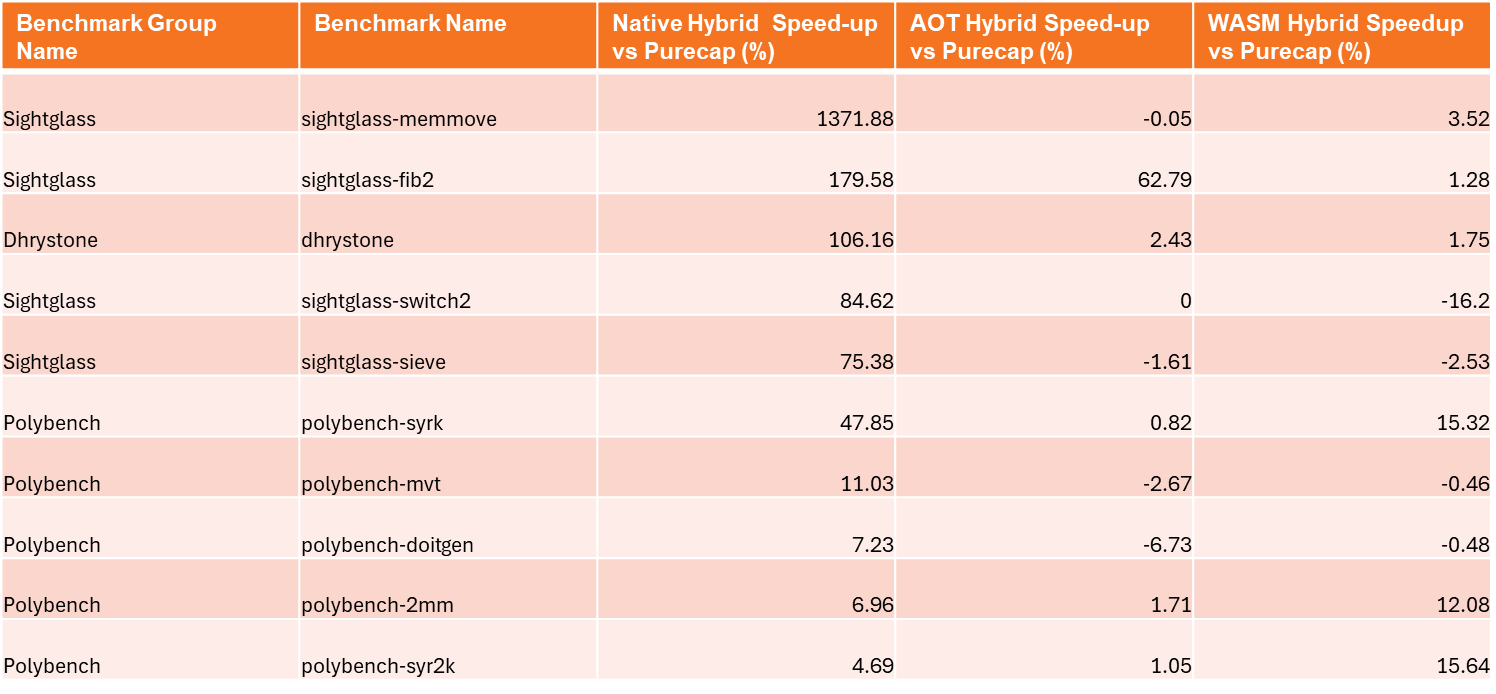
The same is shown for AOT, the 10 worst results are shown:



In the AOT case the majority of tests performed very similarly for hybrid vs purecap. As can be seen above, all but 4 tests had a difference of less than 3%. There was one outlier, *sightglass-fib2*, which was previously identified as one of the significantly badly performing tests on purecap.

*Note: With AOT the impact of the Benchmark ABI is less significant due to the inability to compile AOT files with the benchmark ABI.*

Native



As for the AOT case, we see that in the native case the majority of the tests have minimal differences between hybrid and purecap in terms of execution performance. However when there is a significant difference, it is much worse in the native case.

We can see the “usual suspects” are showing as being vastly worse performing in purecap than hybrid.

*Full Results:*

Attached are the full results for the LLVM only test:



And corresponding *CSV Processor* output giving the percentage speed-ups of hybrid vs purecap:



#### Analysis

The objective of these set of results was to ascertain if fundamental differences in the toolchain were explaining the performance of purecap being worse than hybrid, and to see if it can explain why we have some “outliers” with such awful performance on purecap.

Taking into account our comparison of LLVM with GCC on Hybrid, and comparing the above results with those for the benchmark ABI shown previously, then we see that there is minimal impact of using LLVM for both hybrid and purecap. LLVM does have a slight affect, as we might expect from earlier results, but it is negligible in all but the WASM application case. In the WASM case, where we are dealing solely with the building of WAMR, LLVM only contributes about a fifth of the speed-up percentage in the worst cases.

In the outlier cases for native and AOT, eliminating GCC has negligible effect.

## Benchmarking Conclusions

We have presented results of various benchmark tests and have analysed each of them individually. In presenting conclusions we need to consider the following aspects:

1. In general, do we consider the performance of WAMR to agree with other results from the ecosystem or if not why not?
2. Can we explain the “outliers” which are performing significantly worse on purecap than hybrid?
3. Can we make any recommendations for other projects in improving the performance of purecap applications vs hybrid?

Each of these is considered separately, below.

### Relative Performance of WAMR

In general we find that the performance of WAMR is what we would expect when comparing with other projects within the Morello ecosystem.

If we define the main dataset as lying within two standard-deviations, then we see that:

* For native applications: purecap < 10% worse than hybrid
* For AOT: purecap < 7% worse than hybrid
* For WASM applications: purecap < 12% worse than hybrid

*NOTE: This is provided the Benchmark ABI is used and (therefore) LLVM is used for building purecap executables.*

This is in line with the typical results of 10-15% that have been communicated to us by Imperial College and also identified within reports <https://dl.acm.org/doi/pdf/10.1145/3642974.3652282> and <https://ctsrd-cheri.github.io/morello-early-performance-results/performance-analysis-spec/initial-results.html>, amongst others.

Of most interest is the WAMR application itself; we are pleased to note that the port in general does give this good performance when running WASM applications. We also note that we have much less “outliers” for WASM applications and a much reduced range.

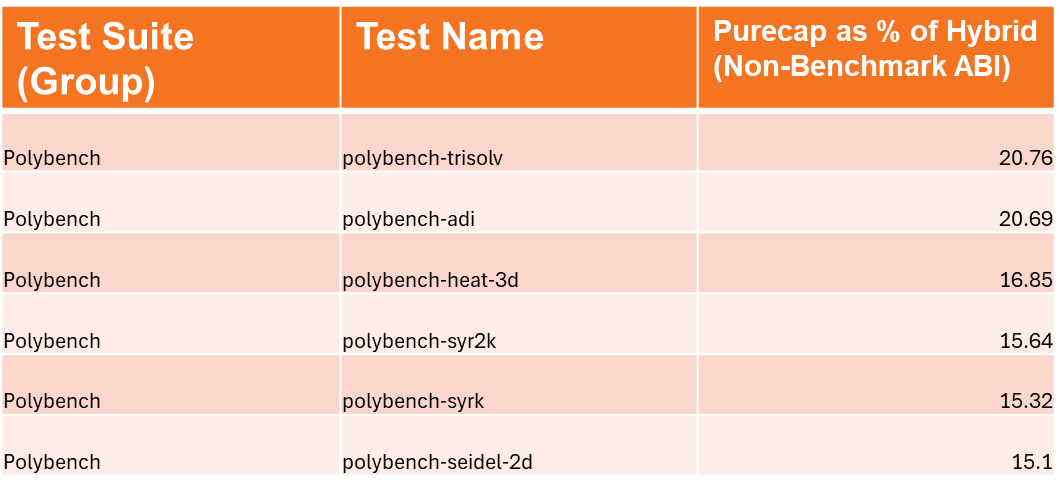
### Explanation for Outlying Poor Performance Results

We have identified that some of the benchmarks are particularly bad on purecap, especially so for the native executables case.

To understand why this is we need to identify the individual algorithms that are performing poorly and see if we can explain specific reasons for these based on what the code may be doing.

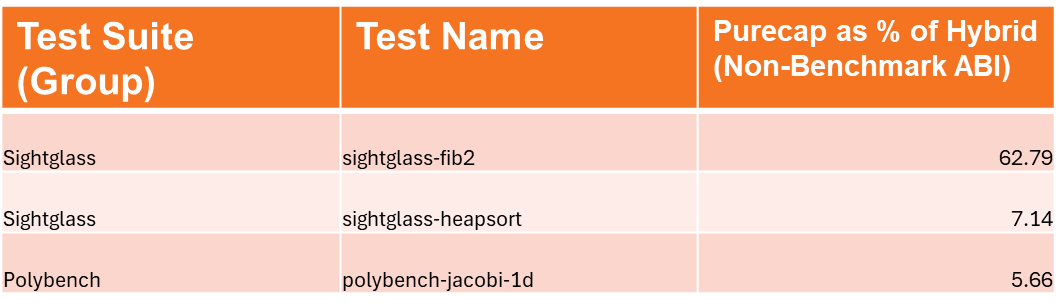
Below are the very bad performers lying outside the main dataset. For these results we use both hybrid and purecap built with LLVM and we continue to use the Benchmark ABI.

WASM



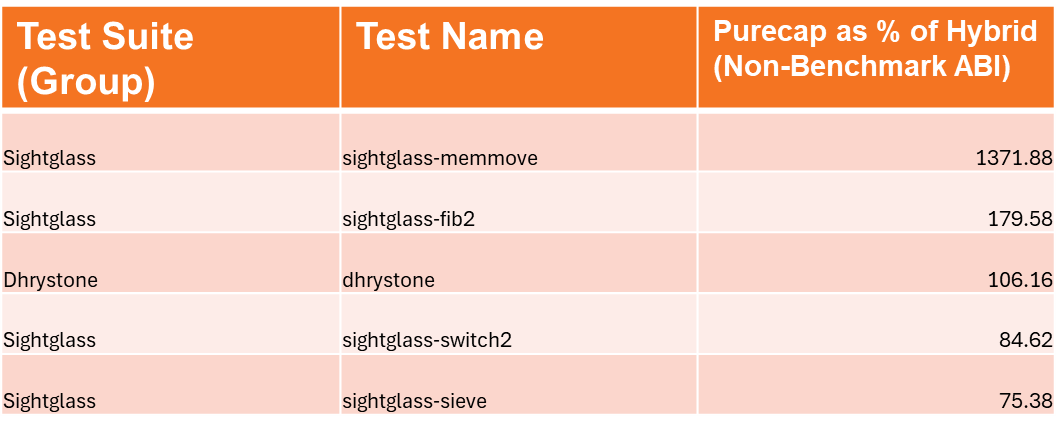
As stated above, the “poor performers” in the WASM case are really not particularly bad. The worst 6 are shown above, but we see none only 3 are outside the “10-15%” and none worse than 21%.

AOT



In the AOT case, we really have only one “outlier” – *sighglass\_fib2* as shown above. Nonetheless, the worst three are shown above.

Native



We are less concerned about the native case as it does not involve the WAMR program or any tools which have been the focus of this project. However to explain why the results of such outliers is so bad would be useful for other projects and for more detailed analysis and future work on WAMR. In particular, the *sightglass-memmove* result is so awful there must be a special case which deserves an attempt at explanation.

The reasons are described below.

#### Issues with Memmove() and MemCpy() Functions

An interrogation of the code for the poorly-performing sightglass tests reveals that they make extensive use of *memmove()* and *memcpy()* functions. This is because the algorithms are testing moving and copy a large amount of data in memory, many times over.

However these tests use a bespoke implementation of the functions that has been supplied with the benchmark test code. The problem is that these are not optimal for purecap whereby capabilities must be used for source and destination pointers – in fact they are particularly poor for a CHERI enabled system.

The *libC* implementations (in this case supplied with the port of MUSL) do specifically take into account that they are running on a CHERI-capable core. If we modify the benchmark code to force the use of the library versions then we see a significant performance improvement. In fact, the hybrid vs purecap speedup returns to being within the 10-15% limit than we would consider normal.

We note that this is only affecting the native applications. WAMR provides a layer of abstraction away from the WASM and AOT applications, and WAMR does use the C standard library functions.

#### Lack of Morello Purecap Multiply-and-Add Instruction

It is noted from <https://ctsrd-cheri.github.io/morello-early-performance-results/performance-analysis-spec/initial-results.html> that Morello purecap lacks a MADD (Multiply-and-Add) single instruction and that Arm expect this to cause a MADD operation to be 15-20% worse in performance on Purecap.

From analysing the algorithms used in the benchmark tests we observe that our worst performing WASM applications implement tight loops running many MADD operations. For example, for *polybench-heat-3d* a snippet of the C code within the test is shown below:

A white background with text

Description automatically generated

Each of the array indexing will require a capability to retrieve the element, and there are very many multiply and add operations occurring inside the inner loop.

We can therefore conclude that without this issue, all of our WASM benchmarks would fall within expected performance limits.

### Development Recommendations for Optimal Purecap Performance

The benchmarking of different combinations of toolchains, and analysis of some of the reasons for poor performance, has enabled us to put together a set of observations and guidelines that should be considered in porting any software to support Morello CHERI. Note these are specifically from a relative performance perspective.

**CHERI Security comes at a Cost**

It must be accepted that to add the security of CHERI there is a performance and resource usage cost. We discovered that, with work, this can be as low as 10% performance penalty but nonetheless this must be appreciated before starting any porting work.

**Early Research Platforms will likely have Issues**

We note that Morello has known issues in the core which is not unexpected in an early research version. This again must be appreciated and when developing and testing on such a platform engineers should be sure to consult the latest literature to understand any performance bottlenecks that have been detected and to apply any suggested workarounds (e.g use of Benchmark ABI).

**Some Hand-Crafting to be Expected**

It is unrealistic to assume that any software can simply be compiled for CHERI and will need no hand-crafting, even if it seems to run without fault. We have seen that some of the benchmarks have need a close analysis to understand why the performance is so poor – if we were interested in optimising them we would have to manually rework the source code. In one case the benchmark would not even run on Morello purecap, but in a number of cases the test ran fine but the performance was found to be poor. It is therefore important to validate any CHERI port against a non-CHERI and be prepared to rework the software as necessary to make it more robust or achieve acceptable execution speed.

**Choice and Maturity of Toolchain will make a Big Difference**

We have seen that the GNU Toolchain, which was in an early alpha release, performed poorly on Morello purecap which suggested there is some scope yet for improving the optimisation steps. Although the LLVM toolchain was a lot better it seems even here there is room for improvement.

The toolchain and its optimisation steps have a significant impact on performance, more so than any other single factor when considered across a whole software program. Toolchain developers will be more likely to work closely with silicon vendors to ensure the benefits and weaknesses of the hardware are reflected in the generation of object code from the toolchain intermediate language. It is clear that all Morello toolchains are still to an extent a work in progress and in the research phase, and this fact is contributory to our results.

**Don’t “Re-invent the Wheel”: Use Standard Libraries Where Possible**

Despite the comments on the toolchain (above), it will always be better to use a dedicated library function that has been written specifically with the executing processor in mind rather than creating a bespoke version. This was demonstrated by the *memmove()* implementation in the polybench benchmarks which performed extremely poorly until the MUSL libC version was used.

In general software developers will not reimplement what already exists, but sometimes this is done for a good reason when it appears to offer a benefit. However CHERI does represent quite a fundamental change in the way pointers work, and as such any expected benefit may well be lost. Of course if the other guidelines mentioned above are followed - in terms of analysing the software, being prepared to hand-tweak areas and run performance tests – then it is far more likely that such issues will be discovered and rectified.

# Summary and Conclusions

This section summaries the problems faced and the “lessons learnt” from the porting work. It also outlines a set of key points to be looked out for when porting other complex, low-level legacy SW applications to a CHERI research architecture.

***NOTE: Conclusions to the experimental WAMR Compartmentalisation work are presented separately – see Section 11.8.***

## CHERI Porting Specific Issues

Here we review & summarise all the different problems which had to be solved to make WAMR + *wamrc* work for Morello pure-cap, with references to elsewhere in the document where the detail is explained.

### Structure Alignment

WAMR typically uses a byte buffer allocated at the end of a structure to service as a placeholder for a heap array that is then dynamically filled. These structures are the backbone of common WAMR runtime functionality, storing the WASM Module data and execution environment internal structures.

***Solution:*** Force byte arrays within structures to be aligned to a capability boundary.

***Reference***: Section 4.2.2.

### Capability Pointer Alignment and Size

CHERI capability pointers are fat, and there they require storage size and alignment which are not the same as the machine’s default word size and alignment. This causes numerous problems in WAMR which assumes that a pointer can be freely treated as a 64-bit integer and vice versa.

The most common fault found during WAMR porting was a *Bus Error* due to a pointer not being aligned in memory to a 128-bit capability boundary.

The specific reasons this affects WAMR so much more than other, higher-level, SW programs is because:

* WAMR dynamically uses byte buffers and within them assigns misaligned data structures which contain pointers
* WAMR casts data that does not contain pointers (and which is therefore suitably aligned) to now contain a capability pointer, and wants to use it as such
* It attempts to perform pointer dereference on pointer-pointer types, with only 64-bit alignment considerations (i.e only native word size alignment considerations)

***Solution***: Solution in all cases is in the form of:

* Allocate or define enough space to be able to pad to alignment (so 15 bytes max pad for a Morello capability pointer)
* *cheri\_align\_up()* to the nearest boundary

***Reference***: Section 4.2.3 for the general problem / solution overview, 4.2.6 + 5.1.1.1 + 5.2.2.1 for the effect on the Interpreter Frame Stack and 5.2.2.2 for the fast interpreter address patching issue. Additionally, for the changes in aligning storage blocks in WAMR’s internal WASM Heap implementation, see Section 4.2.1.3.

#### Pointer Size + Alignment Considerations

Pointers being larger in size has less impact one the resulting misalignment issues are corrected, as this applies to all pointers. There are a few exceptions to this where WAMR assumes a pointer size is fixed to be *sizeof(uint64)*.

There was one case which was particularly difficult to resolve, and involved an innovative solution, and that was in fast interpreter label address / patch handling.

***Solution***: An intricate solution is needed to minimise WAMR code changes and keep them CHERI-specific, this was to provide a larger region for WAMR to store the pointer and also any pad needed to keep it aligned (so 32 bytes to allow for 16-byte pointer + 15-byte maximum pad + 1 byte to guarantee necessary WAMR even byte alignment).

***Reference:*** See 5.2.2.2 for a full explanation of the problem and solution.

### Loss of Pointer Provenance

There are a number of instances of this within the codebase, although most are resolved by fixes implemented for the alignment issues (for example the Fast Pointer issue in See 5.2.2.2). Elsewhere the fix is typically simple, for example using a *uintptr\_t* to store the pointer when needed as integer instead of using *uint64\_t* that WAMR would have been using originally.

Specific cases of loss of provenance can be found when dealing with native code. WAMR converts a linear memory offset (as used in WASM) to an address by adding the offset to the native pointer to the base of the memory. However WAMR then attempts to check for range violations by the difference between two pointers which can have different capabilities.

The other far more complex case was in passing arguments to native functions. WAMR converts pointers to a *uint64* in order to pass it to a native function, formatting the data to match the ABI calling convention and packing all values as 64-bit integers. This of course loses provenance on CHERI and requires a change.

***Solution:*** Checking the pointer difference between two capabilities that mark the start and end of a buffer is a simple problem to solve, by extracting just the offset (to check for equality) and length values (to check the buffer size).

To avoid loss of provenance when passing pointers to native functions it is necessary to handle the pointers separately, and also take into account alignment issues. The solution is to treat all values as full 128-bit, but when packing into a structure which becomes the stack we must carefully align because the capability pointers on the stack need 128-bit alignment.

***Reference:*** Section 4.3 covers all of the significant changes needed to the WASM to native interface, including the new assembly code that was required. Section 4.3.4.4 explains the WASM-to-native address conversion issues, and Section 4.3.4.1 provides details of the modified native calling mechanism needed for purecap.

### Mmap() Issues

WAMR uses *mmap()* to map large buffers for the WASM heap + linear memory, and also for code and data read from an AOT file. The well-documented change to *mmap()* is to apply full access to the initial map (to obtain needed future permissions in the capability), and then modify as needed with *mprotect()*,and this had to be made to WAMR to support CHERI.

Fortunately this is largely specific to the *mmap()* wrapper function, *os\_mmap(),* as implemented for the CHERI platform.

***Reference:*** See Section 4.2.7.

### Out-of-Bounds Access Violation

An intricate problem was found when attempting to run AOT code, which concerned the fact that the object code uses PC-relative addressing to access an initialized data area but these are *mmap()*d separately and so the capability obtained from *mmap()* for the code area cannot be used for the data areas, else an out of bounds *segmentation violation* occurs.

***Solution:*** The solution was complex, involving a lot of WAMR changes, to be able to *mmap()* a single area for both the code + data. For this it is necessary to know the data sizes earlier in the AOT parsing process than WAMR implements, hence an AOT file format change was needed.

***Reference***: Section 7.4.1 covers the changes in detail.

### Morello ISA and ABI Extensions

Morello extends the AArch64 architecture, there are modified A64 instructions and new instruction extensions for C64.

WAMR codes directly in assembler in several instances:

* To marshal calls from WASM to native, by converting buffers of arguments to the correct registers and stack to meet the C calling convention
* In implementing entries in a Procedure Linkage Table, to support AOT object code calling into native library functions at runtime

Additionally, Morello introduces some new symbol relocation types which affect AOT.

***Solution***: The Morello pure-cap handling to call WASM to native is so different that a new assembler file was added (it was impractical to modify the aarch64 one). The WAMR PLT (which does not use a GOT) is implemented with Morello C64 instructions to achieve the same result as existing code. The additional symbol relocations are handled by extending existing processing for aarch64.

***References:*** Section 4.3 explains the WASM to native calling, Section 7.3.2 covers the PLT and Section 7.3.3 goes through the new symbol types.

### AOT Compiler

The AOT compiler runs on x86\_64 but generates AOT for any of the supported targets, one of which is now Morello pure-cap (and hybrid is also supported).

***NOTE:*** For the avoidance of doubt, this is NOT just cross-compilation to a different target than the host. WAMR compiles object code but handles all data, including WASM data and initialized data sections used by the object code, separate – data is processed at runtime. This means that by default the compiler will build object code assuming a 64-bit pointer size/alignment, which would not correlate with the 128-bit pointer size/alignment that occurs at runtime. The reason for this is the AOT compiler is compiling WASM to IR using a parser implemented and running on the host, therefore it works with an x86\_64 architecture.

The following had to be solved for the AOT Compiler:

1. Setting the new target-triple for Morello within the AOT compiler (it does not take the same arguments as clang) – see Section 7.2.2
2. Handling the fact Morello uses a non-default LLVM address space for capabilities (Section 7.2.3 and Section 7.2.4)
3. Dealing with the different layout of WASM structures on the host machine vs the Morello machine (Section 7.4.2)

## WAMR Complexity and Legacy SW Issues

Porting WAMR has been a difficult and complex process. WAMR has required extensive, bespoke modifications to support CHERI in way that minimises changes to the WAMR core code and which does not affect other build targets.

It is noted the University of Cambridge studies suggest the vast majority of software will run with zero or very few modifications to support CHERI. For our project, this description isn’t entirely accurate so it is therefore worth assessing why this has not been the case for WAMR. Suggested reasons are discussed here.

### “Real-world” Software

As anyone with experience of real-world product development in industry knows, project deadlines often force engineers to write software which is not as well structured as would be desired and which may contain “hacks”. Engineers will typically do this with the best of intentions, to fix it later, but then end up working on other projects for other deadlines. What then happens is people leave the project and new people do not understand the SW properly and so people typically add to the existing poor codebase. This creates what we then often call “legacy SW”, where documentation is typically missing or incomplete and there are functions that nobody understands.

WAMR has this feel about it. Although this may not actually be the case it does come across as a bloated, legacy SW project with a number of contributors who may not fully understand how the code operates. It further appears to have a lot of re-use from other previous SW projects and programs, which may also have suffered a similar fate.

When porting to something as different as CHERI, this fundamental nature of the SW makes it difficult to understand operations and make needed changes. Further, if there is an issue then it tends to be compounded because there is no encapsulation of different functionality within the code.

### Embedded Software

WAMR is intended by design to run on an embedded platform – and clearly it is meant for platforms with limited system resources.

Embedded SW engineers work at a low-level and are used to treating pointers as if they are also integers, with plenty of interchanging between the two. Certain techniques which appear “horrible” from a SW Engineering perspective are often used on an embedded platform, particularly when the specific quirks of the end processor are known and utilized.

WAMR features numerous places where pointers are freely cast to integers and vice versa, and no consideration for a platform such as CHERI whereby a pointer contains meta-information and cannot simply be treated as a machine word.

The nature of WAMR, being quite low-level with a lot of target-specific platform code (including some assembler), means it is inevitable that it is so much affected by the impact of CHERI.

### Use of Structured Language

WAMR (including the AOT compiler) is written predominantly in C code, with just a little C++ (and that is mainly in the AOT compiler where a C++ API is mandated to LLVM).

WAMR uses pre-processor macros heavily, as well as language constructs such as goto. This all serves to make the code hard to follow, and more prone to error. Additionally, it means there is no data encapsulation which means certain changes affect more of the code than would otherwise be needed if an object-oriented language had been used.

WAMR would benefit tremendously from being written in an object-oriented language (effectively that would be C++). It is possible C was used intentionally to reduce the memory footprint and improve performance, although if one uses C++ correctly then it is unlikely there will be any real benefit in preferring C. It is more likely that WAMR was put together from legacy code that was already written solely in C, and so this approach was continued as it gained functionality. Instead of C++ classes, then, we get complex C structures which themselves contain sub-structures all of which use pointers ad-hoc throughout the data model.

### Build-time support for Multiple Platforms

WAMR supports a number of build-time targets and makes heavy use of preprocessor macros to compile different sections of code for different targets. However this makes it difficult to add support for a new target or make large modifications to the codebase (as many different parts of the code must change). This inevitably increases the complexity of porting certain parts of the code to CHERI, since if there is something which needs to change then multiple different functions are affected.

### Lack of Documentation

WAMR documentation is largely focussed around how to build and use it. There is almost zero documentation on how it works internally, and this includes there being few useful comments in the code.

Given the other issues - legacy, structured C code with no consideration for any data type being larger than 8 bytes in size - it is essential to understand how WAMR works to determine where the impact of CHERI will be most significant. The lack of any documentation, specification or description of any API makes this increasingly difficult.

### Minimal Support for Development

Often on large SW projects, tools will be produced to aid developers. Such things are not provided with WAMR. There is some support, for example debugging mode or memory profiling support, but much more could be added. For example, there is no external tool (or specification) to parse / analyse an AOT file (e.g like a *readelf* tool).

Debugging WAMR is tricky. This is because of the excessive use of macros, goto instructions and nested tree of structures. This is even more true for AOT, when attempting to find problems in executing AOT code, since this is running as native object code (with no debugging symbols and which bears little resemblance to the WASM from which it was generated). WAMR’s AOT compiler does provide the ability to generate LLVM IR and object code from WASM but this is of limited use.

The difficulty in debugging makes it hard to track down faults that occur when attempting to run under a CHERI architecture.

## CHERI and Morello Issues

It is recognised that CHERI and Morello are still research platforms, and a work in progress.

However, that being said, the Morello board HW was found to work admirably – no issues were found and behaviour was exactly as expected. In terms of the software stack and board setup, although this has improved since the project started it was still found to be a good standard with minimal faults.

Where issues have been found with Morello it has been related to the toolchain. At the time the project began the LLVM toolchain was having fundamental problems compiling C++ for Linux Morello pure-cap, and also was complex to use. The GNU toolchain was therefore preferred; this did have some bugs and confusing disassembly dumps but in the main it has worked well.

Later in the project the LLVM toolchain did increase in maturity and Verifoxx were involved in aiding in the testing of this from Arm’s git *dev* branch. Although LLVM is now supported in the WAMR port repository, GNU remains the toolchain of choice as fewer problems overall have been detected with it.

For the AOT compiler, LLVM must be used (the core and target libraries are needed). This worked better than expected, although this was running on an x86\_64 machine (C64 LLVM libs are not supported on Morello yet, and in any case the port of the *wamrc* AOT compiler to Morello would be too much complexity for that part of the overall WAMR porting work). There were difficulties due to the nature of the Morello using fat-pointers, but it is understood that the Arm LLVM developers are working with the LLVM core project to introduce support for the fat pointer concept and so it is to be expected that such new architectural constructs would inevitably lead to complications for users of the LLVM API.

### Documentation and Examples

It is our view that the level of documentation and examples could be improved on CHERI and Morello.

The CHERI Programmer’s guide reads like a research paper, not a specification, and is therefore hard to follow and not suitable as a quick reference manual. It is also too focussed on *CHERIBsd* and not Morello.

The Morello Armv8-A Extension document is good, and thorough - as has come be expected from Arm reference manuals. However there is a lack of background material on Morello, particularly as Morello has a number of enhancements to CHERI that are not covered well enough anywhere in the literature (so it would appear). Arm’s *CapInfo* web tool is also very helpful, in fact invaluable in tracking down problems.

There appears to be too few examples of CHERI source code, and in particular Morello-specific source code, in the public domain. Although not yet applicable for the WAMR porting, it was noted there is no pure-cap example of compartmentalisation or making use of the executive / restricted mode changes (the latter being Morello specific).

#### Documentation Repositories

It would be useful if there was a better single documentation repository for CHERI and Morello. This could point to other sources of public documentation, could also allow users to notify when a particular document is out-of-date (as this has been a problem we found) and be a record of all known issues with Morello toolchains and firmware. With regard to the latter, several times we reported an issue had been found only to then hear it was a known issue.

#### Forum Resources

Arm have a sub-section on their forum for Morello, which is useful for general Q&A but it does seem to be underused as a community resource. It may be useful if there was another forum set up for Morello to enable people to start threads and post issues they have are having.

As a note for Arm, the Arm forum in general does not present the best User Experience to make posts. This is not Morello specific, but if Arm have available resources then perhaps this could be improved by the tools support team.

## Key points for Future CHERI Porting

The fact that the vast majority, and most complex parts, of WAMR has been ported to work with CHERI Morello pure-cap shows that the CHERI technology is viable for complex, embedded applications such as WAMR. It also demonstrates that Morello is a mature enough platform to consider using with real-world projects, albeit it is clearly not yet at the stage where the SoC is ready for a commercial release.

As the take-up becomes more widespread then organisations will look to port more and more of their legacy, embedded systems to run on a CHERI architecture which will probably be Morello. This section is therefore added as an aid to other porting projects.

#### Windows and Visual Studio are Valid for Development

It is unrealistic to expect that all businesses (such as startups, or those where employees work solely remotely) can support all developers having a dedicated Linux machine which is networked via a wired LAN.

Fortunately, the project work demonstrated that – despite what is listed by Arm as the requirements - a Windows PC with WSL2 can be used as the host, Morello can be connected to an ethernet-to-wifi bridge and even that Visual Studio can be used as the IDE (at the time of writing, Arm development studio only supports Morello on Linux).

#### Pointer Alignment will be the biggest Problem

The WAMR porting work showed that, by far, the biggest problem was caused by pointers requiring an alignment of twice machine word. Therefore:

* + Check structures containing arrays and other structures which *may* be used for pointers, and add suitable alignment directives
  + Use the *cheri\_align\_up()* built-in readily, with suitable padding as needed
  + When debugging, note it is the pointer itself and not the “pointed to” location which needs to be aligned

In the WAMR case, often the “culprit” for a pointer misalignment was a completely different place in the code to where the fault occurred. Therefore a good understanding of the codebase is essential to track down problems.

#### Loss of Pointer Provenance will have Minimal Impact

We found that there were hundreds of warnings about loss of pointer provenance, but in the vast majority of cases the default behaviour was exactly what was required.

Pointer provenance issues had minimum impact on the porting and in only a few occasions were the issues complicated to resolve (in general, using a *uintptr\_t*) was the best solution.

#### Out-of-Bounds Issues will be few, but Difficult to Resolve

Few occurrences of an out-of-bounds condition have occurred in the WAMR port, as in general capabilities are “created” by system calls and then happily used for all future operations (with new capability pointers derived from the same capability).

However if it does occur that one capability needs to access a region of memory only accessible to another capability, then there will be significant code changes required because a capability cannot just be “magically” created to join the two.

#### Badly Written Code will make Porting Worse

As is often the case with such exercises, any code known to be poorly written or which is doing suspect operations should be addressed before attempting a port to CHERI, as CHERI’s fat-pointers will make these things much worse. It is suggested that, if using a structured language, use of (possibly inline) functions to wrap low-level operations that manipulate memory may be useful as this will isolate the areas that need changes for CHERI.

#### CapInfo and Other Tools are your Friend!

It was frequently noted that during the debugging phase if a capability tag was suddenly cleared then it was needed to break-down the capability to an extent greater than that displayed in the debugger. The Arm *CapInfo* web tool allows a quick display of the component parts of a capability, which was extremely useful and this will prove useful to other porting projects as well.

Although static analysis wouldn’t have been particularly helpful in the WAMR porting case, as CHERI and Morello mature it is expected there will be even more static and dynamic code analysis tools made available which will be invaluable in porting future projects.

#### Extensive Testing is a Must

The WAMR codebase is too large, complex and unknown to be able to guarantee all potential run-time CHERI issues have been resolved through modifying the codebase alone. Only by extensive testing can we have a high confidence that the port is successful. This is likely to be the case for porting other legacy systems, and therefore it is recommended that investment in an automated test system with a large suite of tests is pursued as part of the porting project.

# WAMR Compartmentalisation and Capability Manager

CHERI compartmentalisation offers an opportunity for the “double-sandboxing” of a WASM application. WAMR provides a sandboxed environment for the application to run, and a further sandbox can be achieved by running WAMR within a compartment.

An experimental system was developed in order to achieve this, using the Morello platform. On Morello, Arm have realised the CHERI compartment model with the addition of an *Executive* permission such that the PE can either be in *executive* or *restricted* state. The different states allow for use of different system control registers, and state switching from executive to restricted can only be performed using special instructions.

For the experimental system developed as part of this project, we consider a compartment as always running in the restricted state. The concept of a *Capability Manager* is introduced, which runs in the executive state and which serves as the entry point of the WAMR application. The capability manager is in charge of interaction with system functions and any user-defined native or WASI functions, as well as marshalling calls into the compartment. The compartment hosts the WAMR native or AOT runtime and any interactions with the WASM application itself. A well-defined interface allows the capability manager to call into the compartment, and the compartment to call into the capability manager in order to perform *services* (such as invoking a native function).

## Concept

WAMR comprises a number of different components, all of which are callable from native code via the exported API or via a front-end program such as WAMR’s own *iwasm* executable. These are as follows:

1. Module Loader
   * Create module data structures, parse the WASM or AOT file
2. Module Instantiation
   * Create the WASM Execution Environment including linear memory, WASM heap, WASM stack
3. Execute WASM Function
   * Executes the Interpreter or AOT runtime

The execution of a WASM function may be the entry point (*start* or *main* function) or a named function. During execution of the WASM function, other WASM functions may be called and/or native functions (which may themselves call other functions in the WASM application).

Furthermore, in the case that WAMR supports multi-modules, it may be necessary to load several modules and then functions can all into other functions in different modules.

The compartmentalisation of the above is handled as follows:

* Module loader & instantiation operations can run inside a compartment
* Execution of a WASM function runs inside a compartment
* Subsequent execution of a further WASM, or native, function takes place in a compartment

This concept is shown below:

A diagram of a company

Description automatically generated

In this diagram the system supports multi-modules, and two modules are loaded. The WASM entry point is *main()* and this function is called first. It subsequently calls another function, which then calls a native function and a function in a different WASM compartment. Conceptually, calls to other WASM functions happen seamlessly within the WASM application and WAMR but in practice these route through the capability manager.

Native calls are invoked directly by the capability manager; calls to other WASM function re-enter a compartment.

### WAMR Compartment Capabilities and Data Structures

#### Executable Code

WAMR comprises a complicated set of C and C++ functions, with many interactions and calls between them. When considering code which needs to run inside a compartment it is not feasible to isolate individual functions to individual compartments. Therefore WAMR code will be split into two parts:

1. Functions which run inside the capability manager (executive)
   * These comprise WAMR API entry point functions, native code function, system call wrappers, low-level OS handlers etc.
2. Functions which run inside a compartment (restricted)
   * All other WAMR functions which build data structures or execute a WASM application function

Any WAMR code (execution) capability therefore has access to any code which is bound within the compartment, but none outside of it (other than specific and well-defined compartment exit points, provided by dedicated capabilities).

Although this is not as secure (because there is an attack vector whereby the compartment could execute any other code within the compartment in an erroneous way) it provides the needed security to sandbox WAMR. With the complexity of WAMR and the high-level Linux framework and build structure involved then it is simply not feasible to provide a finer degree of code separation.

#### Data Structures

Like any C or C++ program, WAMR requires a native heap and native stack.

Additionally, there is a WASM heap and stack which are provided as part WAMR data structures that are created when the module is first loaded. WASM Linear Memory is also provided (in WAMR this is appended to the WASM heap).

There are numerous data structures in WAMR which collectively comprise all the data needed in order to execute a WASM function:

1. WASM code (from the .wasm file – not to be confused with WAMR SW code)
2. Module data
3. Includes tables, globals,
4. Execution environment
5. Pointers to other allocated memory, current execution frame
6. Function signature info
7. Function arguments (parameters for the function)
8. WASM Frame stack (and space for locals, function return value etc.)
9. Linear memory
10. WASM Heap

As this is CHERI each of these will be accessed by a capability. These capabilities could be individually managed by the capability manager, but where there are complex data structures which contain capability pointers to each of the individual structures then it is sufficient (at least for an initial development phase) to provide the compartment with a single capability for each usable data structure in WAMR.

By utilising WAMR functions to create the data structures (within a compartment), and controlling allocation of heap memory, we can provide the following individual capabilities for any operation within a compartment:

* Module Data
* Execution Environment
* Native Stack
* Native Heap

***NOTE:*** The “native heap” is managed by the capability manager and a service provide to allocate space onto it. WAMR’s WASM stack, WASM Heap and Linear Memory involve an allocation in this way.

### Same or Multiple Compartments

The compartment is an abstract concept. In practice it means that all code and data is isolated from other parts of the system, by means of suitable restricted capabilities.

It is therefore not realistic to say that we are using the “same” or “different” compartment as it depends on what code and data capabilities are set up ahead of transitioning to the compartment (and restricted state). In the WAMR case we can say that:

* The “same” compartment can be considered to be the case when the same native stack and/or native heap is applied for an operation that runs inside a compartment
* The WAMR data structures for a particular module will be different, and so functions in different modules always run in a “different” compartment

Initially for the work the “same” compartment is always used, although a future phase can look at better isolation by having a dedicated native stack for each WASM function. Note that running a different module should always involve a different compartment.

### Calling into a Compartment

The general concept of capability manager calling a compartment is as follows:

*Code:*

1. There is a *compartment entry* executable capability which has
   * Bounds set to the range of WAMR compartmentalised code only
   * Executive permission cleared (i.e it is restricted)
   * A specific, sealed, entry point (likely to always be the same marshalling function)
2. There is a *operation to execute capability*; this is the actual WAMR function to be run
   * This is also bound & restricted accordingly
   * The marshalling code at the common entry point uses this to call the actual operation intended
3. There is an *exit point capability* which allows the compartment to return
   * This is a sealed entry (sentry) capability which allows transfer back to executive
   * It has the executive permission and is ideally tightly bound to a small part of the capability manager code, as it is used only in the transition from restricted -> executive

*Data:*

1. The compartment has a dedicated, memory mapped stack
   * This is switched in automatically by Morrello on an Executive -> Restricted transition, as the *RCSP* register is set up to point to this dedicated stack
2. There is *either* a dedicated capability for a native heap pool, *or* a *service* is provided in the capability manager to allow the compartment to allocate heap memory (discussed further, below)
3. There is a *sealed* capability (restricted and bound for the compartment) which provides access to a capability table for the compartment that comprises individual capabilities for:
   * Function arguments to be passed to the operation
   * WAMR Module and execution environment data structures
   * Other WAMR memory areas such as WASM heap and frame stack

***NOTES:***

* There is no capability for returning arguments; WAMR operations follow the C calling standard so any return value is an argument to the function call that processes the compartment exit function
* Function argument capabilities by be reference types (pointers to pointers) allowing argument return by reference

The compartment calling concept is shown below. Note that a single entry point “unwraps” all needed arguments and data structures and then calls the actual WAMR function that is the entry point for the needed operation:

A screenshot of a computer

Description automatically generated

### Capability Manager Callback Services

The flow as described so far will allow a WAMR operation to run in a compartment by calling a C (or *extern C* C++) function with provided arguments and which returns a value. This WAMR operation function entry point may itself call many other functions within the compartment.

However it is necessary to extend this simple flow by providing *services* within the capability manager which a compartment function can use as part of its processing. A simple example of this would be to log information to *stdout*. This involves a *printf()* type call, which is a native function and which therefore cannot be made from the compartment without a transfer back to the capability manager in the executive state. As has been discussed, even calling other WASM functions should cause a call through the capability manager – again, this is considered a *capability manager callback service*.

***NOTE:*** *At the time of writing, Arm have not yet implemented a security policy for this and so on Linux it* ***is*** *still**currently possible to call system functions from the restricted PE state, although this is not intended behaviour*.

In order to allow the compartment to make calls back into the capability manager to process a service we need to extend the model and provide additional capabilities to the compartment upon entry. This is because, without a dedicated capability, the compartment cannot “magically” just call into the capability manager as it likes. Therefore the following additional capabilities are needed:

*Code:*

* Service callback entry point (generic entry point)
  + Executive permission, ideally tightly bound, for calling the service wrapper
* Table of available services
  + Allows compartment operation to choose the service
  + Can be table of capability function pointers or other secure mechanism to identify the service callback required

*Data:*

* Arguments needed by the service callback function

Note that as the service callback is handled in the capability manager then the native stack / heap etc. are not needed, as these will be the main executive stack and heap.

The generalised flow for a service callback is shown below:

A diagram of a company's company

Description automatically generated

In the diagram we can see the call into the compartment operation (in this case to run a WASM function), which itself involves a service in the capability manager before it finally returns.

## Realisation – Phased Approach

Realising the successful compartmentalisation of WAMR would clearly be a large undertaking, and so a phased approach to the work was devised. The aim was to first of all produce a compartmentalisation framework and achieve the compartmentalisation of a single WASM function. Then the system could be extended by adding the service callback framework, compartmentalising additional parts of the WAMR functionality before adding complex concepts like switch between compartments for each WASM function call and handling multi-module.

Unfortunately, and at the time of writing, it was not possible to complete all the planned development phases due to project timescales.

The table below lists the different phases and which ones have been completed (to date):

| Phase | Summary | Details | Finished? |
| --- | --- | --- | --- |
| 1 | Compartment Framework | * Set up compartment * Provide basic front-end * Execute a single WASM function in compartment (run WASM app in a single compartment operation) | **YES** |
| 2 | Functional WAMR | * All WAMR operations in a compartment * *iwasm* Front-End ported to cap mgr * Still WASM app in single compartment operation | **YES** |
| 3 | Service Callbacks | * Add service callbacks * Implement basic callbacks (for heap allocation etc.) | **YES** |
| 4 | Advanced Compartments | * Support AOT Mode * Support user native functions (service callbacks from WASM) | *NO* |
| 5 | Full Compartments | * Split WAMR to capability manager (native handling) & WASM (compartments) * All system, os, log point, WASI calls involve service callback to capability manager * All WASM calls involve service callback to capability manager | *NO* |
| 6 | Improvements | * Better sealing when switching into compartment * More efficient compartment setup * Higher security (more tightly bound caps) * Use of WASM heap memory pools; better control of heap resources * Support both LLVM and GNU Toolchains | *NO* |
| 7 | Multi-Module | * Enable multi-module feature * Call between modules invokes compartment switch, marshal via capability manager | *NO* |

***NOTE:*** To date, the experimental system:

* + Does not support AOT mode
  + Does not support loading user native code as shared objects
  + Does not implement the full WAMR API calls from the front-end

## Compartmentalising WAMR Code

WAMR’s *iwasm* is a Linux executable which is either statically or dynamically linked, this is configured to be a PIE. WAMR can also be loaded as a dynamic shared object library, with bespoke native code that calls API functions to access WAMR operations.

Neither of these are suitable for compartmentalisation because we require a subset of WAMR code that can be isolated to its own restricted virtual address space. In the current WAMR linking, code which needs to be part of the executable manager is intermixed with code that will need to be part of the compartment and so it is not possible to set up suitable bounds.

Given the previous discussion, WAMR will only be split into capability manager (executive) and compartment (restricted) code, as it is unfeasibly difficult to have a finer grain of granularity. It is also not feasible to attempt to do this “manually” by specifying every symbol name that needs to go in the compartment (either in a link script or at runtime).

Therefore the approach taken is to build WAMR as a dynamic shared object library – which is conveniently already supported in WAMR. For initial phases of the work, all WAMR code will be built to this library. In later phases native / WASI code can be moved to the capability manager executable.

If this library is dynamically loaded, at runtime, then its contents are placed into dedicated memory sections by the loader and we can maintain capabilities bound to the library (and hence WAMR) code but not the capability manager. Furthermore, loading the library in this way allows it to be PIC which means we have full flexibility and portability that would not be achieved if we attempted to bind to a specific address.

### Initialized Capability Problem

There is though still a problem with the library approach, which concerns initialized capabilities.

Within WAMR there are obviously a number of execution code symbols, but also initialized global data and constant tables symbols (and there are many of these). These symbols cannot be resolved at link time because the target destination address is unknown and in fact the symbol may be in a different shared object library. Instead, they are resolved at runtime by the loader.

On CHERI, these particular symbols will be *initialized capabilities* and a full capability is added by the loader.

The problem here is that the loader will use a *root capability* which is provided by the system to the executable program (i.e the capability manager) – but that root will have Executive permission enabled. But in our case the compartment code must run in the Restricted state.

This problem applies to not only the dynamic library loaded for WAMR compartment code, but also any dependent dynamic libraries. At runtime, a symbol may need to be resolved to an actual piece of code in another library by means of a *Procedure Linkage Table*.

Example

Consider the example that the capability manager has obtained the symbol address of a particular function in WAMR. It will use a suitably restricted and bound capability to call this piece of code.

However this will actually invoke a function in a PLT which then looks up the actual symbol (which maybe in a child dynamic library). The symbol address is resolved using a full capability which is initialized at runtime to the actual destination address of the symbol. The problem as stated is this full capability is not restricted, and potentially not correctly bound either.

What then happens at runtime is we make an unexpected switch from restricted to executive state, and so when we then return (via a normal *RET* not a special state-switch return) this is an invalid operation as it switches PE state back to restricted, and there is a HW fault which then occurs.

#### Potential Solutions

Arm have produced two examples which offer potential solutions to this problem.

In the first case, code to run in a compartment is linked to an executable which is then loaded to an absolute address (i.e not relocatable) and the “relocation” actions that would normally be performed by the loader are performed manually using a modified root capability. This is made possible because the loading of the executable in memory is performed using bespoke code.

In the second case, individual functions are compartmentalised at runtime by mapping a dedicated memory area and then copying the code at runtime with manual set up of suitable capabilities.

Unfortunately neither of these approaches are practical for WAMR. It is unrealistic for WAMR to statically link compartment code; it would be too inflexible not least given WAMR upstream is constantly in development. Further, WAMR comprises complex sections so implementing a manual loader is a large overhead.

Regarding the second approach, it is not feasible to manually copy and set up symbols to hundreds of code and data symbols that would have to be dealt with.

#### Actual Solution

The chosen solution is to use the dynamic shared object library loading mechanism “as is”, and then to manually patch all of the resolved symbols in order to remove the executive permissions.

### Relocation Symbol Patching

The WAMR dynamic shared object library is loaded via the POSIX system function *dlopen()*. This is a standard technique to load a library at runtime.

As part of this process, the loader library (*ld.so*) is used to resolve any link-time symbols whose address could only be known at runtime. The library is (on Linux) an ELF file, which is a data structure that includes a list of *relocation tables*. These inform:

* Where is the symbol within the file (as an offset) which needs resolving
* What sort of symbol is it (how should the resolution be performed)
* What is the target address that it needs resolving to

Morello ISA adds a number of new resolution types to deal with the fact that any addresses are now capabilities and not pointers (or number of bytes from start of memory). Therefore if there is a target address to be used for the resolution, in Morello it will also have bounds and permissions for example.

When the loader performs the resolutions, using the now-known actual addresses, on Morello these will be a full capability and will have the executive permission. This is because the loader is running in the executive state and is using a root capability with an executive permission. However in our case the restricted state is required for the symbol.

We must therefore “patch” all of the symbols resolved by the loader, after it is finished the loading process. This is effectively undoing a lot of what the loader is doing – therefore it could be considered that the loader code should be changed instead. However this would require rebuilding the standard C library which would presumably mean a rebuild of the Linux kernel, which is outside the scope of this work. Instead it is hoped and expected that Arm or others will implement a system function to load a library into a compartment in due course.

The process of patching the symbols is shown below (a very simplified example):

A diagram of a computer system

Description automatically generated

It can be seen that the loader actually maps memory and copies the relevant blocks from the shared object ELF file into that memory. At this point it knows the actual target address of all symbols, so a symbol offset from top of ELF file can now be converted into an actual address (e.g somewhere within a block of executable code).

It will then resolve the required address value to write into that location (e.g the target of a branch instruction) which may well be the address of a PLT placeholder in a PLT for one of the other libraries. This step is repeated for all the symbol tables, and the loader completes (and *dlopen()* returns).

At this point, we must now patch all the symbols which use full capabilities. Once this is done, we can successfully call into the compartment with the knowledge that we will remain in restricted mode.

#### Symbol Patching Caveats

Before explaining the patch-up process, some issues must be explained.

*Disable Lazy Binding*

In order for the patch-up to work, all symbols must have been resolved by the loader that is performed from the *dlopen()* call. Normally, PLT symbol lookup and resolution would only occur upon the first access to the symbol (e.g the first branch instruction) which means initial loading is quicker. This is known as *lazy binding*, but we need to tell the loader to not do this in our case.

*Namespaces*

If the executable is dynamically linked and happens to use a symbol in a dependent dynamic shared object that is also used by the WAMR library then there would only be one copy of the library loaded and the same bit of code would be used in both cases.

However in our case this would cause a problem because the capability manager executable requires a library loaded into the executive state but for the WAMR library it must be restricted.

The solution in this case is to load the WAMR library into its own *namespace* which means that any shared libraries are loaded into memory twice – and the new namespace is isolated from any others.

This requires use of the *dlmopen()* instead of *dlopen()* function. Note that if the executable is linked statically, then there is no issue because all symbols are resolved at runtime. Currently, the capability manager is statically linked but in the future it will depend on libraries which do not have a static variant and therefore it will need to be dynamically linked.

To facilitate this the framework enables the code to be built for both statically and dynamically linked executable.

*Library Cleanup*

This is only an issue when the executable is dynamically linked, i.e *dlmopen()* is used instead of *dlopen()*.

In this case, ahead of application exit any library termination functions are called by the system using hidden functions within the loader. The problem here is that this will invoke an exit function for the WAMR library which was loaded (and its namespace) but this is a problem because we will then invoke restricted code directly from executive.

The solution taken to this is to, just ahead of application exit, repeat the symbol-patching process but this time reverse all of the patching and hence restore the executive permission to all symbols.

Due to the need to do this, a root capability is needed to be used for all symbol patching – we cannot use the symbol directly as we cannot “create” new permissions from this.

*Toolchain Differences*

The LLVM and GNU toolchains before differently with regard to symbol relocation during dynamic library loading. The LLVM toolchain is likely much simpler because it includes a dedicated *.capinit* section which allows initialized capabilities to be modified *en masse*.

At the time of writing only building with the Gnu Toolchain is supported – the LLVM toolchain would need some code modifications to work.

#### Process

The symbol patching is complex. In essence it involves parsing the ELF structures which can be obtained from the *dl{m}open()* call and locating the relocation tables. We then iterate through each entry in each relocation table looking for types which would need fixing. Types which need fixing are those which involve a capability (i.e a Morello relocation type), which are in a section we care about and which are valid (which have the tag set).

Note that it is not necessary to resolve the address the symbol references – this was already done by the loader. We just need to modify the capability already set for that address.

In order to perform the relocation, a base capability is used which is itself derived from the executable’s RW auxiliary vector capability (passed into the executable startup by the shell). As stated already, this is to deal with the need to revert capability patchups in some instances and because you cannot set permissions to a capability which does not have them already, so it is necessary to source the fixups from a root capability in this case.

Apart from anything we want to restrict, all other settings of the capability to fix-up are derived from its existing value. This includes making the capability a sealed entry capability if it is a function call.

##### Finding Relocation Tables

Relocation tables are parsed to identify symbols to be fixed up, but first we need to locate the relocation tables in the dynamic library structure.

This is done by parsing the program header structures obtained from the *dlopen()* load of the shared object. We find the dynamic data area (PT\_DYNAMIC) which itself is a table of addresses and sizes of different dynamic data sections.

This enables us to build a map of *dynamic data IDs → [address | size]*, and cherry-pick those we are interested in.

From this we can determine the start address and end address where the relocation tables within the shared object have been loaded into memory, namely the *.rela.dyn*, *.rel.dyn* and *.plt.rel* sections.

##### Parsing Relocation Tables

Parsing the relocation tables is a relatively easy process given we don’t need to actually calculate the fix up address and just need to modify the capability permissions and bounds as required.

Only certain relocation types need to be patched, which are defined in the Morello ISA spec and are as follows:

* R\_MORELLO\_CAPINIT
* R\_MORELLO\_GLOB\_DAT
* R\_MORELLO\_JUMP\_SLOT
* R\_MORELLO\_RELATIVE
* R\_MORELLO\_TLSDESC

Standard *Aarch64* types which do not involve a capability can safely be ignored.

Additionally, there are certain addresses which should not be patched. These are those which are called intentionally from the executive which are related to the loading and teardown of the library. We check for those addresses by loading information from the *PT\_DYNAMIC* area of the shared object ELF structure.

##### Allowing Address Writes

In order to patch up an address capability then it needs to be writable. But if it is code or a constant value it will not be. To deal with this, before the patch up process we again consult the program headers for the loaded shared object and for any *PT\_LOAD* regions we use *mprotect()* to allow write access (this is permitted as it runs from the executive capability manager and we use a root capability to access the memory).

At the end of the patch up process we restore the *PT\_LOAD* region to read and execute permissions only.

##### Repeating for all Shared Objects

As the WAMR shared object library may depend on other shared objects then the fixup is needed for these, too.

This is handled by iterating all the shared objects in the namespace - apart from *ld.so*, the loader, which is never called from the compartment.

Overall, the approach yields a set of data structures, which is shown below:

A diagram of a computer program

Description automatically generated with medium confidence

Internally, in the capability manager code, the above data structures are loaded into a class hierarchy using the *link map* obtained from *dl{m}open()* and *program headers* for each loaded shared object.

The single *CompartmentLibs* object enumerates all loaded shared objects, and for each of then a *Shared Object* class contains:

* List of *Program Headers:* this is a map, header ID -> header info (which includes start/end addresses)
* A *Dynamic Section object* which is a parsed *PT\_DYNAMIC* program header
  + This object holds a map of all section symbol IDs -> address of symbol
* A list of *Relocation Tables* which are generated by parsing the dynamic section map
  + These are stored as a vector of shared pointers to the relocation table data

These structures allow the self-describing ELF headers to be built into ever more intricate structures, resulting in a relocation table structure for each of the relocation tables we need to know about.

The relocation structure comprises a start and end address, resolved from an ELF offset by converting the offset base to the base address of the library in memory. The start and end address themselves are obtained by looking up the symbols in the *Dynamic Section* map. For example, here is the code to get the *.rel.dyn* relocation table start address, end address and element size:

**Range CDynamicSection::GetRelRel(size\_t& elem\_size) const**

**{**

**uintptr\_t addr = reinterpret\_cast<uintptr\_t>(m\_base + GetEntry(DT\_REL));**

**size\_t sz = reinterpret\_cast<size\_t>(GetEntry(DT\_RELSZ));**

**elem\_size = reinterpret\_cast<size\_t>(GetEntry(DT\_RELENT));**

**return Range(addr, sz);**

**}**

Here we lookup three symbols in the map, DT\_REL which is the start of the relocation table, DT\_RELSZ which is the length and DT\_RELENT which is the size of each entry (so we know how many bytes to skip in passing each table entry). We return these as a Range, which is a utility class to represent an address block.

All the relocation tables can be handled in the same way, hence use of a base class. The only difference is how their addresses and size can be resolved, hence the use of a base class virtual function to call into the derived class for each relocation table type.

**Patching**

The actual capability fixup patching code performs a number of checks before then modifying the capability to using a base capability. The checks are:

1. Relocation type must be one of the Morello types which need patching
2. Target address must not be within a *skip range*
   * Skip ranges are read from the Dynamic section symbols
   * They are things like FINI code which is part of executive state library cleanup only
3. Capability tag must be valid

The fix-up value is derived by taking the bounds and permissions from the current capability and then using a base root to set or remove executive permissions and reseal *sentry* as needed. The code snippet below shows the derivation of a fix up value:

**uintptr\_t DeriveFixupValue(uintptr\_t val\_to\_fixup, bool makeRestricted=true) const**

**{**

**auto cap = Capability(m\_fixup\_cap)**

**.DeriveFromCap(reinterpret\_cast<void\*>(val\_to\_fixup),**

**makeRestricted ? 0 : ARM\_CAP\_PERMISSION\_EXECUTIVE, // Perms to add**

**makeRestricted ? ARM\_CAP\_PERMISSION\_EXECUTIVE : 0 // Perms to remove**

**);**

**return cap;**

**}**

The *DeriveFromCap* method of the *Capability* utility class loads the base *m\_fixup\_cap* which is a root and copies the *val\_to\_fixup* but adds in and removes permissions as shown. Effectively, if we switch from executive -> restricted we explicitly remove the Executive permission and if not we are reverting the fixups on exit so we explicitly add the Executive permission.

*DeriveFromCap* uses intrinsic methods from *cheriintrin.h*:

**Capability& DeriveFromCap(void \*p, size\_t set\_perms=0, size\_t clear\_perms=0)**

**{**

**// If requested, make sure p is within range of existing**

**if (cheri\_base\_get(p) >= cheri\_base\_get(m\_cap) &&**

**(cheri\_length\_get(p) + cheri\_base\_get(p))**

**<= (cheri\_length\_get(m\_cap) + cheri\_base\_get(m\_cap)))**

**{**

**// Restrict lower bound to same as p**

**m\_cap = cheri\_address\_set(m\_cap, cheri\_base\_get(p));**

**// Restrict upper bound same as p**

**m\_cap = cheri\_bounds\_set(m\_cap, cheri\_length\_get(p));**

**// Current offset (like address) same as p**

**m\_cap = cheri\_offset\_set(m\_cap, cheri\_offset\_get(p));**

**}**

**else**

**{**

**m\_cap = cheri\_address\_set(m\_cap, cheri\_address\_get(p));**

**}**

**size\_t desired\_perms = cheri\_perms\_get(p);**

**m\_cap = cheri\_perms\_and(m\_cap, desired\_perms | set\_perms);**

**m\_cap = cheri\_perms\_clear(m\_cap, clear\_perms);**

**if (cheri\_is\_sentry(p))**

**{**

**m\_cap = cheri\_sentry\_create(m\_cap);**

**}**

**return \*this;**

**}**

Note the application of *cheri\_sentry\_create()* if the source capability was a sealed entry one.

**Writable Area for Patch**

Before the patchup, the area containing the symbols which may be patched is switched to writable, and this is switched back after the patching completes.

The program headers for the library were previously created in a map, actually this is a *std::multimap* to permit duplicate keys because there may be multiple *PT\_LOAD* blocks. For each block the program header structures will inform the start & end addresses, along with the flags (read | write | execute).

Ahead of the capability patch, the map of program headers is searched to find PT\_LOAD entries and the address range for each is retrieved. *mprotect()* is then used to give the range writable permissions.

After the patch, the original values are written by checking the flags from the PT\_LOAD entry in the program headers map.

The function *ProtectAllBlocks()* handles this in the code, here is the call to make the blocks writable:

**ProtectAllBlocks(PT\_LOAD, false, PROT\_READ | PROT\_WRITE);**

To restore the settings according to flags (after all relocation table patchups done), the third argument defaults to true and flags are not needed:

**ProtectAllBlocks(PT\_LOAD)**

*ProtectAllBlocks()* just iterates all matching entries in the multimap:

**bool CSharedObject::ProtectAllBlocks(Elf64\_Word type, bool restore\_original**

**int64\_t prot\_required) const**

**{**

**// Find matches**

**auto itrs = m\_phdrs.equal\_range(type);**

**bool result = true;**

**for (auto itr = itrs.first; itr != itrs.second; ++itr)**

**{**

**// Keep going on error**

**result &= ProtectBlock(itr->second, restore\_original, prot\_required);**

**}**

**return result;**

**}**

*ProtectBlock* is given an *Elf64\_Phdr* structure which contains address and flags, and looks like this:

**bool CSharedObject::ProtectBlock(const Elf64\_Phdr& phdr, bool restore\_original,**

**int64\_t prot\_required) const**

**{**

**// Get the address of the start of the block and the size**

**void\* base = m\_base;**

**uint8\_t\* block\_start = &(reinterpret\_cast<uint8\_t\*>(base))[phdr.p\_vaddr];**

**// Align so we are always working with values aligned to an OS page size**

**uint8\_t\* block\_aligned = cheri\_align\_down(block\_start, m\_page\_size);**

**size\_t sz = phdr.p\_memsz +**

**reinterpret\_cast<ptrdiff\_t>((block\_start - block\_aligned));**

**// Figure out the perms we need**

**if (restore\_original)**

**{**

**// Derive from the supplied flags, otherwise is supplied**

**// Need to map Program header constants to mprotect constants**

**prot\_required = 0;**

**if (phdr.p\_flags & PF\_X)**

**prot\_required |= PROT\_EXEC;**

**if (phdr.p\_flags & PF\_W)**

**prot\_required |= PROT\_WRITE;**

**if (phdr.p\_flags & PF\_R)**

**prot\_required |= PROT\_READ;**

**}**

**if (0 != mprotect(block\_aligned, sz, prot\_required))**

**{**

**return false; // mprotect failed**

**}**

**return true;**

**}**

## Calling WAMR Operation in Compartment

As part of the Phase 1 work it was necessary to produce an initial framework which could call a single WAMR operation within a compartment. For this, the *wasm\_runtime\_call\_function\_a()* call would be modified to invoke a switch to the compartment for execution – this is a basic top-level “run a WASM function” API call that passes command-line arguments, so it seemed a reasonable first choice.

At this point all other WAMR API calls were taking place in the capability manager. To facilitate this, two version of WAMR needed to be built and data structures had to be manipulated to switch them from executive to restricted, as well as a bespoke and simple front-end program being needed.

This initial framework was then modified to make it easier to call other WAMR API functions in the compartment. Essentially this involved creating a lot of “boiler plate” code with the result that it became very easy to call a WAMR function in the capability manager using a “proxy”, and the framework would then transit this across the compartment boundary, unwrap, and call the actual WAMR function inside the compartment.

Note that a number of complexities were faced when trying to implement a modern-C++ STL framework to handle this, and in the end simpler constructs were used. The problems were down to attempting to move complex, virtual class instances between the executive and restricted states but this caused problems with the virtual function tables and other hidden aspects within library code. This prevented aspects of modern C++ such as *std::bind()* and RTTI from functioning properly. Instead, simple derived class structures were used which relied on static casting based on a passed in type identifier (similar to how Microsoft COM was implemented originally, without RTTI support).

### Compartment Setup

The capability manager sets up a compartment by building a set of data structures for it. This process involves setting the compartment entry and exit capabilities and allocating the compartment stack by mapping a memory area. At the time of writing the compartment has a large stack size for development purposes only, but this should be reduced to whatever WAMR requires currently.

The compartment setup results in a number of capabilities being created to represent the compartment, which are as follows:

* Compartment Data Structure, itself comprising:
  + Compartment CSP: The compartment stack, which is *mmap’d()* to a block of available memory suitable for a stack
  + The permissions for this are set to *Restricted* and *Non-executable* (data only)
* Compartment DDC: Set to null capability
  + This is because DDC is only applicable for Hybrid-cap, and Arm are phasing it out
* Compartment CTPIDR: Set to same as the capability manager CTPIDR
  + This is used for application use and WAMR uses it for a thread ID
* A sealer capability, which is used to seal the WAMR function argument data structure that will be created later
  + The sealer is created from the auxiliary vector *AT\_CHERI\_SEAL\_CAP*
* A read+exec capability which points to the compartment exit function in the capability manager
  + This is the known, bound point which the compartment will use to return
* A capability which is the common compartment entry point, used to unwrap and route the actual function call
  + This is derived from the auxiliary vector *AT\_CHERI\_EXEC\_RX\_CAP*
  + The address is determined by looking up the function in the loaded shared object namespace by a hard-coded given name, specifically this function is called *CompartmentUnwrap* and is implemented in any WAMR compartment

The remaining information is constructed when a specific WAMR function is called as at this point the data structures passed in function arguments can be sealed.

### Compartment Call WAMR Function

When a WAMR function is called, the remaining data structures are built and then the compartment can be entered. All arguments for the WAMR function need to be passed to the compartment as a data structure accessed through a single capability.

Additionally, the WAMR function within the compartment to call needs to be passed as a capability. This is tightly controlled from the capability manager, so only the intended top-level WAMR function can be called.When the compartment is entered, the “unwrap” function will retrieve the data arguments and call the WAMR function itself, so as far as the WAMR function is concerned it was called directly. The WAMR function in question is resolved, by name, by looking up the symbol in the library.

This whole process behaves like an IPC mechanism, whereby the appearance is given of calling the WAMR function directly from the capability manager but in reality the call involves bundling up data, crossing the compartment boundary and then unpacking the data.

The final data structure is built at the point the WAMR function to call is actually known. This is then sealed. The data structure is shown below, along with the behaviour of the unwrapping function in the compartment.

A diagram of a computer system

Description automatically generated

With reference to the above, we have:

* Compartment CSP, DDC and CTPIDR as mentioned previously
* Sealer capability, as mentioned previously
* Compartment entry point, as mentioned previously
* Sealed entry Capability which is the compartment exit function in the capability manager (as set previously in compartment setup)
* Sealed entry Capability which is the actual WAMR function to call inside the compartment
  + This is obtained by looking up the function name in the loaded shared object library, and building a capability with restricted permissions and bounds using the auxiliary vector *AT\_CHERI\_EXEC\_RX\_CAP* as a base
* An identifier of the WAMR function being called, this is needed by the “unwrap” function in the compartment
  + This is required to be able to resolve the WAMR function pointer and argument data to the correct type
  + This is necessary because RTTI is not really possible as discussed previously, hence we perform a static lookup to access the derived class
* All data arguments, of varying type, for the WAMR function in question

A capability is taken to the above structure, which is itself then sealed. This can then be passed to the assembly code which switches into the compartment.

#### WAMR Function Arguments

Each WAMR function takes a different number of arguments, of varying types. All WAMR entry functions are in C, not C++.

To pass function arguments across the compartment boundary a class instance is used. There is a different class for each WAMR function, with the function arguments being stored as member data within the class.

Each of these classes derives from a base class (named *CCompartmentData*)which itself has attributes that are common to all WAMR functions, namely the capability for the WAMR function to call, the capability for the exit function, and an identifier to know which WAMR function is being called.

When a WAMR function proxy is called in the capability manager it causes an instance of the correct class to be constructed and a capability pointer (actually a *std::shared\_ptr*) to be created for it. The entry function is set based on resolving the WAMR function to its name and looking up the address of the function in the loaded shared object library.

Inside the compartment unwrap function, the exit function pointer capability and the WAMR-function-to-call capability are extracted from passed in data. The identifier reveals how the WAMR function pointer should be cast, and which arguments are available in the passed-in data. This then enables the function to be called directly using passed in arguments.

The return value from the function is always passed back as a *uintptr\_t* in order to be large enough for any type (including any capability type). This can then be cast in the exit function in the capability manager.

The exit function is called directly, which is actually again in assembler to marshal the return back from the proxy function.

The compartment data structure is internally handled as a class object called *CCompartment.* The class structures for *CCompartment, CCompartmentData* and derived classes are shown below (example of calling the WAMR function to create the execution environment is shown):

A diagram of a computer flowchart

Description automatically generated

### Crossing the Compartment Boundary

The capability manager calls functions in the compartment, and receives a return value, across a well-defined boundary. This boundary switches from the PE *Executive* to *Restricted* state and involves a call into a function implemented in the dynamically loaded library. This function is called the *compartment entry point* and its job is to marshal the call into a specific WAMR function that is being called.

“Crossing the boundary” must be done in Morello assembler and involves using the *BRR* instruction to branch to a function with a switch to restricted state.

The process is as follows:

1. The compartment builds the final data structures and calls a plain C function, *CompartmentCaller(),* passing the following arguments:
   1. The assembly function to call to actually switch to the compartment
   2. The data structure containing:
      1. Compartment CSP
      2. Compartment DDC
      3. Compartment CTPIDR
   3. The sealed entry capability which is the *CompartmentUnwrap* entry point function inside the compartment
   4. The sealed capability which points to the class instance that holds:
      1. The WAMR function pointer capability, and an identifier for the function
      2. The capability for the compartment exit function in the capability manager
      3. The WAMR function argument data
   5. The sealer capability used to unseal the above
2. The assembly function passed in is then called from C, passing the remaining arguments
3. The assembly code saves the current values of the restricted CSP, CTPIDR and DDC along with the link register (return address) onto the stack and then prepares the compartment stack using the values passed in for compartment CSP etc.
4. The sealed capability is then unsealed and set as *C0* i.e the first argument to a C function a direct branch to the compartment entry point (the “unwrap” function is performed) which switches to the restricted state
5. We are now in the compartment. The Unwrap function is passed a capability which points to the class instance containing needed arguments. This is first treated as the base class *CCompartmentData*, in order to retrieve the WAMR function ID and the underlying WAMR function pointer capability.
6. As the WAMR function ID is known, we can now static cast to the actual derived class instance, and directly call the WAMR function and provide all function data arguments correctly. The WAMR function runs, and may return a value which is cast to a *uintptr\_t*.
7. It is now time to return from the compartment. This is done by calling the compartment exit function via the passed in capability and giving it the WAMR function return value as an argument.
8. This calls us back into the capability manager in the executive state. The exit function is implemented in assembler. As we are in executive state:
   1. We are now switched back to the main stack - no longer using compartment’s
   2. We can therefore directly retrieve the link register from before, i.e the return address within the capability manager to go back to
9. We finally then return, ensuring *C0* register correctly contains the return value from the WAMR function. This is then made available as a return value from the proxy, giving the appearance the WAMR function was called and returned directly.

The diagram below illustrates the process:

A diagram of a system

Description automatically generated

#### Compartment Entry Point

The compartment entry point as explained previously is resolved from looking for a hard-coded symbol name in the WAMR shared object library. This entry function is key, as it is the only function which must always exist in the library and which provides a hard link between the capability manager (executable) and compartment (dynamic library). The compartment exit point is handled by capabilities passed to the compartment, and individually named WAMR functions are dynamically resolved via a proxy mechanism (so it is up to the using class to know the desired name) – but the entry point is effectively a hard coded constant.

It is therefore necessary that it never changes, but this could be problematic if the library needs to change or a new functionality is added in the future (and we don’t want to break backwards compatibility). It is therefore convenient to provide a *trampoline* function which simply branches to the real entry point – this branch code is implemented as an ASM branch. This has the additional benefit of the address of the capability (which does not have write permissions) being elsewhere to a function which actually does anything useful. Therefore it is a slight security improvement.

A further improvement is to restrict the capability to be bound only to the address range of the library itself. If the PCC was used, this would include the range of the entire application which would mean the compartment could in theory access anywhere in the capability manager. By restricting to only the address range of the compartment’s loaded shared object library in memory it means that there is this added security benefit.

***NOTE:*** *It is not possible to set the bounds to only the trampoline function. This is because the next step in the calling process is to resolve a capability from the PLT of the loaded shared object. The accessor for this PLT capability uses the ADRP instruction which derives an address from PCC. If PCC at that point did not include the PLT in its addressing range then a segmentation fault would occur due to an invalid capability tag.*

*In theory it could be possible to resolve a capability bounds to be limited to the library’s PLT, but this is a future work item that would be non-trivial to achieve.*

The concept of the trampoline is shown below (PLT redirect shown also):

A diagram of a flowchart

Description automatically generated

### WAMR Proxy Functions

The intent of the initial phases of the work was to provide a framework that can call a single WAMR function within a compartment. However this should also make it easy to call other WAMR functions within a compartment, essentially it should be fairly boiler plate code to make this work.

The objective is that a proxy WAMR function will be called in the capability manager, with the same function declaration as the real WAMR function, which to the capability manager SW appears to invoke directly. However under the hood it will actually transit to the compartment and then call the WAMR function for real. The framework should then handle everything in the middle layers, and it should be made as simple as possible to support a new WAMR function.

Given that there are in excess of 20 WAMR functions in the exported API, and many of these are needed for a basic *iwasm* program, then clearly there should be minimal effort to support the full API. Additionally it is quite feasible that functions will change and more will be added to the API in the future (from upstream WAMR) – therefore the framework should be flexible enough to support this.

To this end, a proxy class – *CwamrProxy* – is provided in the capability manager. Any WAMR function can then be called by calling a member of this proxy, for example:

**exec\_env = m\_proxy.wasm\_runtime\_create\_exec\_env(m\_module\_inst, STACK\_SIZE)**

Will end up, in the compartment, calling the existing WAMR function:

**wasm\_exec\_env\_t wasm\_runtime\_create\_exec\_env(wasm\_module\_inst\_t module\_inst,**

**uint32\_t stack\_size);**

The names are the same, and the signatures are the same.

In a high-level programming language, this would of course be easy to do and it is no different in modern C++. Unfortunately when we cross the boundary between capability manager to compartment, and change PE state, things become much more complicated and so a hybrid solution is created which needs a little additional work to add a new WAMR function to the mechanism.

#### Implementation of Solution

The *CCompartmentData* class was mentioned previously, along with the need to have a derived class which provides the specific argument data for the WAMR function being called. This allows the handling of capabilities and functionality needed for the general compartment framework that unwraps arguments and calls the underlying WAMR function to be in the base class, whilst the actual arguments (of varying type and quantity) is I the derived *CCompartmentData* class.

To add support for a new WAMR function, a derived *CCompartmentData* class is added which simply stores arguments passed to its constructor in the object instance. For example, here is the class for the *wasm\_runtime\_create\_exec\_env()* function shown above:

**class CWasmCallRuntimeCreateExecEnvData : public CCompartmentData**

**{**

**public:**

**wasm\_module\_inst\_t module\_inst;**

**uint32\_t stack\_size;**

**public:**

**CWasmCallRuntimeCreateExecEnvData(**

**wasm\_module\_inst\_t module\_inst\_,**

**uint32\_t stack\_size\_**

**) : CCompartmentData(WAMRCall\_callCreateExecEnv),**

**module\_inst(module\_inst\_),**

**stack\_size(stack\_size\_)**

**{}**

**};**

Note that due to the static casting in the compartment it is necessary to pass some ID to define the underlying data type and WAMR function being called (in this case *WAMRCall\_callCreateExecEnv*, which is just an enum (not a class enum) value).

This class is used by creating a new instance of it, passing needed arguments, and obtaining a *std::shared\_ptr* to this. This is then passed to a function in the capability manager which expects a *CCompartmentData* pointer, i.e a base class pointer. This is the function which is responsible for finishing the setup of the data structure and transferring the object to the compartment.

Inside the compartment the standard unwrapping is done, and we cast the base class pointer to get the actual derived class type, and then extract arguments to call into the actual underlying WAMR function.

The final step is to add a little more boilerplate in the capability manager in order to avoid lots of copy-pasting of code, as the *CWamrProxy* function will essentially just need to forward arguments to the *CCompartmentData* class function. Therefore we use a *variadic template function* for the proxy function. Again using the example from above, the code looks like this:

**template <typename T, typename... Args>**

**uintptr\_t CallWamrFn(const std::string& fn\_name, Args&&... args)**

**{**

**return m\_compartment.CallCompartmentFunction(fn\_name,**

**std::make\_shared<T>(std::forward<Args>(args)...)**

**);**

**}**

**template <typename... Args>**

**wasm\_exec\_env\_t wasm\_runtime\_create\_exec\_env(Args&&... args)**

**{**

**return (wasm\_exec\_env\_t)CallWamrFn<CWasmCallRuntimeCreateExecEnvData>**

**(\_\_func\_\_, std::forward<Args>(args)...);**

**}**

Here, the *wasm\_runtime\_create\_exec\_env()* passes its function name (i.e the string “wasm\_runtime\_create\_exec\_env”, resolved from \_\_func\_\_) and passed in arguments to *CallWamrFn().* *CallWamrFn()* creates an instance of a *CCompartmentData* child class (in this case, *CWasmCallRuntimeCreateExecEnvData*) and calls a function called *CallCompartmentFunction()* to finish building the data structures and transfer control to the compartment.

As part of this step, we will resolve the symbol named “wasm\_runtime\_create\_exec\_env” inside the WAMR shared object library, and then pass a capability to this symbol along with all argument data to the compartment, where casting will give the correct *CCompartmentData* derived object pointer from which the actual arguments can be obtained.

When creating the proxy function it is only necessary to ensure the function name matches that of the plain C WAMR function, and that the correct *CCompartmentData* class has been used.

Note also that arguments are always passed back as *uintptr\_t* to ensure any size of value can be returned. Only at the top-level is the cast performed to the actual WAMR function return type.

An additional point to note, demonstrated by the above example, is the creation of data structures within the compartment which are then passed back to the capability manager. In this case the data structure is only needed by the compartment, so is essentially an opaque handle within the capability manager – it will be passed as an input argument to other functions that run in the compartment. This suggests the data could be sealed, however as the capability manager is the “overseer” there is not really an added benefit in doing this.

In future, these data structures will be needed in the capability manager as well.

#### Supported WAMR Proxy Functions

At the time of writing, all functions needed to implement the *iwasm* front-end from the WAMR exported API have been added to this mechanism. They are:

* bh\_log\_set\_verbose\_level()
* wasm\_runtime\_set\_exception()
* wasm\_runtime\_get\_exception()
* wasm\_runtime\_lookup\_wasi\_start\_function()
* wasm\_runtime\_set\_wasi\_args()
* wasm\_runtime\_set\_wasi\_addr\_pool()
* wasm\_runtime\_set\_wasi\_ns\_lookup\_pool()
* wasm\_runtime\_destroy\_exec\_env()
* wasm\_runtime\_deinstantiate()
* wasm\_runtime\_destroy()
* wasm\_runtime\_unload()
* wasm\_runtime\_full\_init()
* wasm\_runtime\_load()
* wasm\_runtime\_instantiate()
* wasm\_runtime\_create\_exec\_env()
* wasm\_runtime\_lookup\_function()
* wasm\_runtime\_call\_wasm\_a()
* wasm\_application\_execute\_main()
* wasm\_runtime\_get\_version()
* wasm\_runtime\_is\_xip\_file()
* wasm\_runtime\_get\_wasi\_exit\_code()
* wasm\_runtime\_register\_native()
* wasm\_runtime\_unregister\_native()

The WAMR API supports additional functions that have not yet been added.

## Compartmentalisation with iwasm Front End

For the initial phase of work a bespoke front-end was created which only used the framework to call a single WAMR function inside the compartment – this required a complicated build as WAMR was added to both the front-end executable and a dynamically loaded library.

The second phase of the work was to utilise the WAMR proxy function framework to allow all needed WAMR API functions to be called in the compartment. This meant the *iwasm* front-end needed modification.

The *iwasm* program is implemented as *main(),* with some support functions. This is C not C++ code, but the proxy function framework is C++ only. To handle this, the main file was compiled as C++ and slight changes made to allow C code to be treated as C++.

At this stage WAMR had not been split into library compartment code and capability manager code, therefore it was not possible to use WAMR OS utility functions for e.g reading the WASM file and logging. Instead, C++ functions and classes were added to support these in the capability manager front-end.

### New WAMR Product

WAMR supports numerous platforms, and implements these by providing a “product-mini” directory containing *main.c* and a CMake build file for that platform. It then defines *WAMR\_BUILD\_PLATFORM* CMake flag to the name of the platform in question, and our top-level *CMakeLists* will pick up the correct folder according to this flag that is set in *CMakePresets.json*.

The decision was made to create a new product for the CHERI-WAMR version with capability manager / compartment support, in order to avoid corrupting the existing CHERI-WAMR product. This enables the new experimental CHERI-WAMR version to be merged to the same branch as the main product without interfering with it.

The new product is:

***linux-cheri-purecap-capmgr***

For reference, the existing and main port of CHERI-WAMR is:

*linux-cheri-purecap*

#### Removal of Hybrid Support

The compartmentalisation / capability manager implemented cannot be supported in hybrid mode. Therefore for the *linux-cheri-purecap-capmgr* product, CMake flag *ENABLE\_CHERI\_PURECAP=0* must **not** be defined else the build will fail.

#### Product Targets

The new product CMake will cause the generation of:

* *iwasm* : The standard WAMR front-end, modified for capability manager
* *libiwasm.so* :The compartment dynamic shared object library, which contains all the WAMR code

#### Build Configuration

As explained, the capability manager (so in this case the *iwasm* executable) can be built statically or dynamically, which affects how the WAMR library is loaded.

The existing flag *CHERI\_STATIC\_BUILD* was repurposed for this. *CHERI\_STATIC\_BUILD=1* is the default, and builds a static executable. *CHERI\_STATIC\_BUILD=0* builds dynamically.

#### Restrictions on WAMR Flags

The experimental capability manager / compartment version of WAMR does not yet support all WAMR functionality. To avoid build or runtime errors, this means certain WAMR features must be disabled with the following flags set:

* WAMR\_BUILD\_AOT=0 → AOT mode not yet supported
* WAMR\_EXTERNREF\_APP=0 → Calling through WAMR API from native not yet fully supported
* WAMR\_BUILD\_NATIVE\_TEST\_LIB=0 → Calling native functions not yet supported

#### New Compiler Flags

The new product will define two new compiler flags:

* ENABLE\_CHERI\_COMPARTMENT → The compartment library is being built
* ENABLE\_CHERI\_CAPMGR → The capability manager is being built

These enable different functionality within the same file to be included for the capability manager or for the compartment. They also make it possible to check certain files are only being built into the correct target. For example, this code is present at the top of a file which is compartment only, and which must not be built into the capability manager:

**#if !ENABLE\_CHERI\_PURECAP || !ENABLE\_CHERI\_COMPARTMENT || ENABLE\_CHERI\_CAPMGR**

**#error "Bad make definitions; compartment must have ENABLE\_CHERI\_PURECAP=1,**

**ENABLE\_CHERI\_COMPARTMENT=1 and ENABLE\_CHERI\_CAPMGR=0"**

**#endif**

### Modifications to iwasm Executable

To support the *linux-cheri-purecap-capmgr* product a bespoke *iwasm* executable was needed, as this now needs to be a capability manager. This introduces some additional arguments that are not found in the standard iwasm, and also contains necessary source code to call WAMR functions via the proxy class and handle relocation symbol patch-ups.

#### Command Line Option Changes

*iwasm* must now always load the WAMR compartment library, therefore a new argument is added to specify the pathname for this.

Additionally, a new logging level is added which is identical to the existing *VERBOSE* level but which will also dump the symbol relocation tables to *stdout* (in a sense, then, it is “verbose++”). The new arguments are as follows (other arguments omitted for clarity):

**iwasm [--wamr-lib=/path/to/libiwasm.so] [-v=0|1|2|3|4|5|6] [other options]**

**wasm\_file [args...]**

Where:

* *--wamr-lib* is a new optional argument which specifies the name of the WAMR compartment shared object library to load at runtime. The default value is *./libiwasm.so* i.e libiwasm.so in the current directory.
* *-v=6* is a new debug level which is identical to the WAMR verbose level of 5, but additionally causes the capability manager to dump all symbol relocation tables parsed from the libiwasm.so ELF

#### Source Code Changes

The product’s main.c, containing main(), is changed to use the *CWamrProxy* object instance for member functions, which itself will create the compartment as needed. A single instance is created in *main()* and passed to all utility functions which require it.*.* This proxy is then used to call WAMR functions in the compartment.

The capability manager uses its own C++ logger, rather than having to include all of the WAMR utility functions. To this end, logging calls in the *iwasm* program are replaced to use this logger and the logging level is set by converting the *-v=<verbosity level>* to a C++ logger level equivalent.

### Changes to Common CHERI-WAMR Code

Most of the WAMR code is shared amongst all products - this code is now built into the *libiwasm.so* compartment library which contains all WAMR functionality.

However WAMR also supports providing platform specific code. This has been done previously for the *linux-cheri-purecap* product, but a version of this code is now needed for *linux-cheri-purecap-capmgr*.

Given that most of the *linux-cheri-purecap* code is identical to what is needed for *linux-cheri-purecap-capmgr,* and a DRY methodology, then a common folder was needed. Therefore the following has been created:

* A common folder for both *linux-cheri-purecap* and *linux-cheri-purecap-capmgr*
  + This contains code common to both CHERI platforms and also common API headers for code which has a different implementation
* The *linux-cheri-purecap* folder now contains only implementations of functions which are only applicable to the standard CHERI-WAMR product
* A new folder is added for implementations which are specific to the newly created experimental *linux-cheri-purecap-capmgr* product

Each of the product-specific platform folders contains a CMake file for that product. Both of these will include files in the *cheri-purecap* common folder, as well as the files for the specific product implementation.

Additionally, the *linux-cheri-purecap-capmgr* CMake file adds files which are needed for the capability manager build, but not for the compartment.

### Capability Manager Service Callback Support

The service callback framework has been briefly introduced, but is discussed in full in the next section.

From the perspective of this section, it needs to be appreciated that the proxy function framework is applied for service callbacks as well, therefore there is a proxy in the compartment which calls actual function in the capability manager.

As these functions have the same declaration, and use the same header, dedicated support is needed.

This is handled by the *linux-cheri-purecap-capmgr* shared folder containing one implementation for the compartment and a separate implementation for the capability manager. When building the WAMR library, the compartment version is used and when building the *iwasm* executable the capability manager version is used. As the filelist is determined at CMake makefile generation time, separate lists are provided in the CMake file in the product’s platform folder.

The compartment version - built with the “standard” WASM CMake file structure - calls member functions of a proxy class, which will transfer execution through to the capability manager. The capability manager version is then the “real” function which actually implements what is required, e.g heap memory allocation.

## Compartment Callback to Capability Manager Services

The compartment needs to be able to call services in the capability manager as part of its normal flow, for example to allocate memory on its heap or print a message to *stdout*. Anything which involves a system call or native code cannot be performed in the compartment, as it runs in *Restricted* state.

A generalised mechanism is therefore required to make this possible.

This can be achieved using the same ASM marshalling code that is used to switch from the capability manager executable into a compartment, however a modification is needed because there is no compartment CSP/DDC/CTPIDR that needs to be switched in - there is only one capability manager executable, and so only one main stack etc.

***NOTE:*** *The same mechanism can allow a transit from one compartment to another, for example a future phase of work is to allow one compartment to process a WASM function and then call into another compartment to process a further sub-function in WASM. For this WAMR “double-sandboxing” work it is expected that all calls are “overseen” by the Capability Manager, and therefore the Capability Manager is always directly involved in switching between compartments. This is not the same as Arm demos where, for example, an inter-compartment call involves nothing more than basic Asm instructions running in the Executive state.*

### **Service Callback Flow**

The overall flow of how an operation in a compartment can then callback to services in the capability manager, as part of its processing before it completes, is illustrated below. Here we have all the phases to setup a WASM module and execute a WASM function shown:

A diagram of a company

Description automatically generated

If we examine the an individual operation in more detail, the sequence occurs like this (shown for the example of allocating heap memory):

A diagram of a company's company

Description automatically generated

The compartment entry and exit points are shown, which would be abstracted to a function call and then return. But *within* this processing we have an exit back to the capability manager to call a specific identified service, which itself then returns. This indicates that two more capabilities are needed to be passed to the Compartment during the initial entry:

* Exit point to callback a service
* Specific function to call which implements a service in the capability manager

The service callback does not interfere with the original compartment call because the calling mechanism in the capability manager saves stack states, so we return to the original compartment processing as if any other function were being called.

The mechanism implements a *proxy* within the compartment that will actually call into the capability manager to carry out the real implementation of the function.

This is the same mechanism used to call functions inside the compartment, and the classes and data structures for service callbacks are analogous to the compartment calls themselves.

### Service Callback Processing

The same assembly function inside the Capability Manager that is called when entering the compartment is used for calling a service callback function – this entry point is called *CompartmentSwitchEntry()*. This is entered via the same C calling mechanism - a C Compartment Caller function is implemented in the compartment in the same way as it is implemented in the capability manager.

To this end, the same data structures are used. However, the call from the Compartment to the Capability Manager does not need to switch in restricted parameters for stack etc. (RCSP/RDDC/ RCTPIDR) because the capability manager must use the executive (main) stack. Therefore, these are set to Null by the compartment’s calling code as they will be unused.

When the capability manager switches to a compartment, as we have seen already a *CCompartmentData* derived data structure object is used to provide function-specific arguments along with various capabilities needed such as the compartment exit point back to capability manager. The same concepts are used for the service callback, however the base data class is slightly different and so we instead have *CCapMgrServiceData* in lieu of *CCompartmentData*. (In theory there could have been a common base class for both of these).

The differences for the service callback vs the compartment call are as follows:

* No exit function pointer for the service callback exit function is required. This is because it is itself in the capability manager, and therefore known to the capability manager
* This exit function is not the same as used for the compartment → capability manager returning, the reason for this is explained further below
* The “function to call” is a function within the capability manager, instead of a (WAMR) function within the compartment
  + This is still supplied by the compartment, the reasons for which are explained further in the remainder of this section
* The ID for static function pointer casting is that of a service callback function, and is analogous to the WAMR function ID used in calling capability manager → compartment function call

We can now present the data structures used in service callback processing. Previously when calling the compartment was described some of the additional capabilities used to enable service callbacks were omitted, to avoid overcomplicating the explanation.

Structures are shown below:

A screenshot of a computer

Description automatically generated

The compartment structure is that passed *from* the compartment *to* the capability manager assembly code. We have the values to be used as the new CSP, DDC and CTPIDR but note these are all Null. This is because they are not needed as there is no restricted state to switch in – the result is to set restricted versions of the register to Null. This is a good security enhancement as if for some reason we did accidentally attempt to switch to the restricted state then a fault would occur.

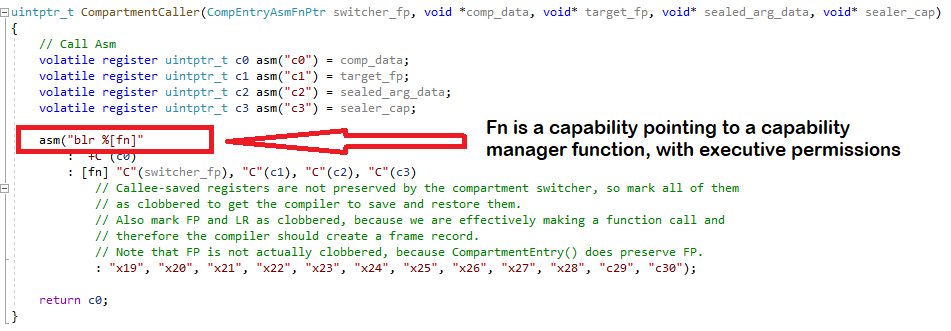
The sealer capability is the same one used when calling into the compartment – it seals the callback function data arguments, static function ID and the individual function’s entry point capability. The main entry point is the unwrapper for the service callback mechanism, in this case *CompartmentServiceHandler*. This *CompartmentServiceHandler* function determines which underlying “do something” function needs to actually be called, and uses the passed in capability to call it.

Note that no capability needs to be passed to provide the exit point back to the compartment. This is already known to the capability manager, and it is more secure to let the capability manager decide this for itself.

#### PE State Switching

The compartment runs in restricted state, therefore there is a switch to the executive state to execute the service callback function, and a switch back to restricted during the return.

Switching from restricted -> executive state in Morello does not require a special instruction, unlike executive -> restricted. It actually occurs during common C code inside the compartment, when a direct branch (not a function call) is made to the call back entry point (*CompartmentSwitchEntry*). The snippet of code which does this is shown below:



This same code is compiled in both the capability manager and the compartment. In the compartment, the branch will perform a PE state switch since the target capability has executive permissions and source does not; in the capability manager the state switch is not necessary.

### Service Callback Return

When the compartment returns back to the capability manager it calls an (executive) assembly code function in the capability manager which returns back to the point at which the compartment was called.

The same method is used to return from a capability manager service function; an assembly function is called. However as execution is already in the capability manager, the function can be referenced directly. That function in assembly code is called *CompartmentServiceCallbackSwitchReturn()* and the argument to that function is the return value, which is always a full sized capability and passed as a *uintptr\_t.*

It should be noted that here there is a slight difference in the assembly code between returning *from* a compartment and returning *to* a compartment from a service callback. The reason relates to the way stack frames work in C and the fact that service callback functions utilise the main stack within the capability manager.

In the case of the compartment -> capability manager return (not related to service callbacks), there is a function call to an exit function from within an existing C function that itself never actually returns. This means the stack frame for this function is never unwound.

This though is not a problem, because at this point we are using the *restricted* stack. Calling into the capability manager immediately switches to the executive PE state which then switches back to the main executive stack- this main stack was not used while we processed the compartment and therefore the main (executive) stack pointer is set up correctly to return at this point. Before the next compartment entry the original top-of-stack is restored and so the correct stack frame hierarchy is restored automatically. Essentially, after we are finished with the compartment its stack can be left dangling.

#### Service Callback Stack Frame

For the capability manager service function things are different because the main (executive) stack is being used throughout. Therefore we cannot leave this “dangling” when we switch back to the compartment, as we’ll need it again when the overall compartment operation returns (in fact we need to retrieve the return address from the main stack). We therefore need to fix this stack when we make the service callback exit call, otherwise it will be corrupted.

One approach to this problem would be to use a separate, bespoke, stack for the capability manager service functions which would then isolate the main stack. However this is difficult to achieve because unlike the restricted stack we would not be able to cleanly switch between the two when the PE state changes, and also we would have to manage the stack pointer for our “extra” stack.

A simpler solution is to manually unwind the stack frames during the capability manager service function exit processing. We mandate that the assembly code *CompartmentServiceCallbackSwitchReturn()* function is only ever called from the entry point (the *CompartmentServiceHandler()* function) and therefore the unwinding comprises only and always:

1. Precisely one C code frame
2. A “pseudo-frame” saved by the Assembly code *CompartmentSwitchEntry()* function when we first invoked the service callback

The unwinding needed is shown below:

A diagram of a phone

Description automatically generated

At the point when *CompartmentServiceCallbackSwitchReturn()* is called the stack is shown as above. The first frame to be unwound is that placed by the compiler when *CompartmentServiceHandler()* was called. It is unwound in the standard way, which on Morello Aarch64 is to retrieve the last frame’s pointer from the location pointed to by the current frame pointer (CFP on Morello).

However, we cannot then do *new CSP = previous CFP* because the frame structure used in our assembly code is different. We need to retrieve the stack pointer at the point we called *CompartmentServiceHandler()*. From the diagram above, it can be seen that unlike the compiler, we have placed the frame data *after* the location of the FP on the stack. Fortunately the size of that data is known since we implemented it ourselves and therefore we simply need subtract this size from the location of the frame pointer to get correct address. (It is a subtract, not an add, because on Aarch64 - like many architectures - the stack grows downwards in memory).

Having now got the correct address, we have restored the stack state to how it was when we called the capability manager service handler function. From then on, the rest of the processing is exactly identical to the way we exit the compartment operation back to the capability manager and so we already have some code to do that.

The excerpt from the assembly code shows this additional processing needed in *CompartmentServiceCallbackSwitchReturn()* before the “exit from compartment to capability manager” common bit in *CompartmentSwitchReturn()* is run:

**.globl CompartmentServiceCallbackSwitchReturn**

**CompartmentServiceCallbackSwitchReturn:**

**// Capability manager service return to compartment**

**// Remove the last stack Frame**

**// Note our "frame" is arranged differently to the compiler's**

**ldr frame\_ptr, [frame\_ptr]**

**sub stack\_ptr, frame\_ptr, #COMPDATA\_STRUCT\_SIZE**

**// Everything else is same as CompartmentExit**

**b CompartmentSwitchReturn**

***Note:*** *“frame\_ptr” is an alias for CFP, i.e C29 register, and “stack\_ptr” is an alias for the CSP register.*

### Additional Compartment Entry Capabilities

Clearly some capabilities needed to manage the service callback mechanism need to be passed to the compartment from the capability manager in the first place. This is done at compartment entry time.

Previously we omitted these from architecture diagrams for reasons of brevity, but the below diagram shows in more detail what is needed:

A diagram of a company

Description automatically generated

On the left hand side we see the compartment making a service callback, with the red ellipse showing the crossing of the boundary back to the capability manager and the transfer of the “compartment structure” block.

The following fields are sourced from the capability manager, passed in the call to process the compartmentalised operation in the first place:

1. Sealer capability
2. Capability for Callback entry function in the capability manager (*CompartmentSwitchEntry())*
3. Capability for Service handler function to call *(CompartmentServiceHandler())*
4. Lookup table of capabilities for individual service call functions (e.g *service\_callback\_XXXX()* in the diagram)
   * This allows the compartment operation to select the service callback function it needs to make, and grab the capability for it from the table

More detail on each of these is now provided.

#### Sealing Capability

The sealing capability is the same as the one used by the capability manager to seal data to the compartment. It is provided as an attribute to the compartment, so it can be used for data being sent to a service callback.

The concept is that each compartment will be given its own sealer capability, which is unique to each compartment, for maximum security.

**NOTE:** Future work should remove passing this sealer capability back from the compartment during a service callback; instead the capability manager should resolve it by looking up compartment data when given the ID of a compartment. This requires additional implementation to add all compartment data to a <compartment\_ID> keyed map, which can then be examined when the service callback is made. This work has not yet been implemented due to project timescale constraints.

#### CompartmentSwitchEntry Calling Point

This is the top-level entry point into the capability manager’s assembly code involved in switching PE state. It is exactly the same as used when calling the compartment.

#### Service Handler Calling Point

This *CompartmentServiceHandler* function is the unwrapping function that marshals any service callback. It is called from the assembly code in the compartment manager which handled the state switch.

This function could be resolved directly by the capability manager as it is known to the capability manager. However passing it across the boundary and back again:

1. Permits the same assembly code to be used as calling into a compartment
2. Gives the framework maximum flexibility (e.g in future a different entry point could be used for different compartment types)
3. Makes it easier to support compartment -> compartment transitions in future (should the need arise, although this is not expected to be required)

#### Individual Service Function to Call

The actual service function to be called in the capability manager is determined by the compartment: the compartment calls whatever service it needs from the available services. For example, currently implemented are service functions to allocate general memory, free memory and allocate a linear memory buffer.

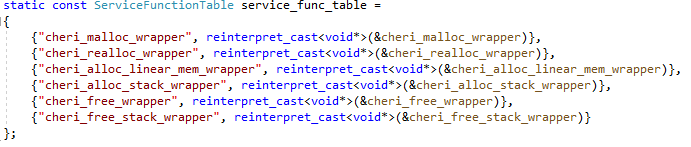
The actual service function is implemented in the capability manager and therefore the compartment either needs a capability for the function or a mechanism to identify the function. In the latter case, the resolution must then be done in the capability manager.

The implementation as written provides the compartment with the capabilities themselves so that resolution to which capability function pointer is used is done within the compartment. Resolution is done automatically using the name of the function as a key to a map to derive the correct function pointer capability. This is more automated, because the name of the function can be derived from the function call at compile time and then used to lookup the actual function pointer (this is a level of reflection that is not readily available due to the fact we are calling a function in a different PE state).

The mechanism is handled as follows:

1. Along with the rest of the *CompartmentData* attributes, the capability manager passes the compartment a reference to a map of string -> capability function pointers, for each service callback function
2. When a service callback function proxy is called in the compartment then the name of the function is looked up in the map to resolve the function pointer capability
3. This is passed to the capability manager and during the *CompartmentServiceHandler* unwrapping operation it is then called directly
   1. Note: A corresponding identifier for the function is also resolved by the proxy because static casting is used for service callback unwrapping as well (RTTI will not easily work due to the switch of PE state)

A snapshot of the function pointer map is shown below, as defined in the capability manager:



At the time of writing, only memory allocation service callbacks are available, as shown.

##### Justification for Implementation

It should be explained why the compartment is provided with a capability function pointer map of service callbacks – allowing the compartment to provide the capability function pointer itself – instead of an identifier to keep all capabilities inside the capability manager. It may seem a security risk to do it this way, but because the capabilities are sealed entry (sentry) with restricted bounds the risk is minimal, and the benefits outweigh the risk. They are as follows:

Localisation of Error Handling

If the function by name cannot be resolved to a capability, then this is handled in the compartment *using* information sourced from the capability manager. This avoids an unnecessary transition to the capability manager and a bespoke error code, which would anyway need handling in the compartment

Greater Flexibility for potential Future Code Improvements

Ideally, a mechanism such as *std::bind()* would be implemented to bind all service function callback arguments and a function pointer during the compartment itself. This would then require all the capability function pointers to be provided to the compartment. The framework has, therefore, been implemented in a way that makes it easier to extend in the future.

***NOTE:*** *At this time, it has not been possible to implement std::bind() because the transfer across the compartment boundary causes a problem with the internal class object data and the virtual function tables. However with more study and support in the standard C++ library then this may be achievable in future.*

### Invoking a Service Callback Function

Like calling compartmentalised WAMR operations service callbacks are invoked by calling a proxy function in a compartment service callback proxy class. However the WAMR code is largely implemented in C and therefore a mechanism is needed to provide a C API which can retrieve an instance of the proxy class to be called (the proxy is only available in C++).

This is achieved by instantiating a single instance of a proxy function class *CServiceCallProxy* when the compartment is first entered via starting a compartmentalised WAMR operation. This instance remains active until the compartment finally exits and so its lifetime is the time the compartment function is running for. All service callbacks then use this proxy instance.

To provide access to the single proxy instance from any function, a static *getter* is provided, the code snippet is shown below:

**// Global CServiceCallProxy ptr which is set to the single CServiceCallProxy on each compartment call**

**static std::unique\_ptr<CServiceCallProxy> g\_service\_call\_proxy;**

**// Accessor for user classes**

**CServiceCallProxy \*CServiceCallProxy::GetInstance()**

**{**

**return g\_service\_call\_proxy.get();**

**}**

Using the service function can then be done as follows (example shown) - and note this is callable from C:

**extern "C" void\* cheri\_malloc\_wrapper(size\_t size)**

**{**

**return CServiceCallProxy::GetInstance()->cheri\_malloc\_wrapper(size);**

**}**

#### Variadic Templates

Like the WAMR operation proxies for the compartment, the service callback proxy functions also utilise C++ variadic templates to provide a level of boilerplate code.

Considering the *cheri\_malloc\_wrapper()* function as an example, the following proxy is implemented in the Compartment’s proxy function (*CServiceCallProxy::cheri\_malloc\_wrapper())*:

**template <typename... Args>**

**void \*cheri\_malloc\_wrapper(Args&&... args)**

**{**

**return (void\*)CallServiceFn<CCheriMallocCapMgrServiceData>**

**(\_\_func\_\_, std::forward<Args>(args)...);**

**}**

This variadic template resolves arguments as supplied to the proxy, and calls a common function supplying the string “cheri\_malloc\_wrapper” as the first argument:

**template <typename T, typename... Args>**

**uintptr\_t CallServiceFn(const std::string& fn\_name, Args&&... args)**

**{**

**return CompartmentServiceCallback(fn\_name,**

**std::make\_shared<T>(std::forward<Args>(args)...)**

**);**

**}**

This function creates the service callback data structure (in this case an instance of *CCheriMallocCapMgrServiceData*) and calls *CompartmentServiceCallback()* to actually transition into the capability manager*.* Before doing that, *CompartmentServiceCallback()*:

* Resolves the string “cheri\_malloc\_wrapper” to the corresponding capability function pointer (assuming it could be found in the map)
* Completes the *CCheriMallocCapMgrServiceData* object instance by adding the service handler capability function pointer and providing the other needed arguments to the Asm function call, namely:
  + The sealer capability
  + The map-looked-up service function pointer to call, from above
  + The RCSP/RDDC/RCTPIDR structure (all fields set to NULL as not required)

The *CCheriMallocCapMgrServiceData* object instance pointer is itself sealed and passed as a function argument.

The final call which actually switches into the capability manager’s assembly function (and the executive state) is *CompartmentCaller()*, a C function implemented in all compartments and capability manager. The arguments to this *CompartmentCaller()* is then as follows:

**auto ret = CompartmentCaller(**

**m\_compartment\_data->service\_callback\_entry\_fp, /\* CapMgr entry fn in Asm \*/**

**&comp\_data\_nulls, /\* Stack etc. to use (n/a) \*/**

**reinterpret\_cast<void\*>(**

**m\_compartment\_data->capmgr\_service\_fp**

**), /\* CapMgr service handler called from Asm \*/**

**sealed, /\* Sealed argument data including the underlying function to call \*/**

**m\_compartment\_data->sealer\_cap /\* Capability used to seal \*/**

**);**

#### Service Callback Implementation and Build

The framework provides a means for a service callback proxy in a compartment to call through to the capability manager and actually call the service function, therefore the function is effectively implemented twice:

* The proxy, which calls through to the capability manager
* The real implementation, in the capability manager

Clearly, these functions have the same name and arguments (and hence same declaration) and therefore:

* There is a common header file for the function
* An implementation is built into the compartment, which is the proxy function
* Another implementation is built into the capability manager, which is the real function

The build can either be done with separate files for the two implementations, or the same file can be used with compiler flags switching in the correct implementation for the build.

When the compartment is built, the compiler macro *ENABLE\_CHERI\_COMPARTMENT=1*is defined and when building the capability manager then *ENABLE\_CHERI\_CAPMGR=1* is set.

If capability manager / compartments are not in use, i.e we are building the CHERI-WAMR version that does not support these new features, then neither of the above is set.

Therefore a *cheri\_malloc\_wrapper()* is implemented as follows, and this works for the existing *linux-cheri-purecap* product and our new *linux-cheri-purecap-capmgr* product:

**extern "C" void\* cheri\_malloc\_wrapper(size\_t size)**

**{**

**#if ENABLE\_CHERI\_COMPARTMENT**

**/\* Provide a proxy implementation for the compartment \*/**

**return CServiceCallProxy::GetInstance()->cheri\_malloc\_wrapper(size);**

**#elif ENABLE\_CHERI\_CAPMGR**

**/\* Provide the real implementation for capability manager service \*/**

**void \*ptr = std::malloc(size);**

**return cheri\_perms\_clear(ptr, ARM\_CAP\_PERMISSION\_EXECUTIVE);**

**// Set suitable perms for restricted usage in compartment**

**#else**

**/\* Provide a WAMR implementation that does not use compartments at all \*/**

**return std::malloc(size);**

**#endif**

**}**

#### Service Callback Functional Flow

The MSC below shows the flow through the system for the *cheri\_malloc\_wrapper()* example. This demonstrates the flow through the service callback framework and notes the points where the PE state switches:

A diagram of a project

Description automatically generated with medium confidence

### Implemented Service Callback Functions

As part of the Phase 3 work service callbacks were added for all WAMR heap allocation and free functions.

WAMR freely allocates needed memory on the native heap, and does so throughout all its operations. The WASM frame stack and WASM linear memory are also allocated in this way.

Previous work during the porting of WAMR to CHERI split out the allocation functions for the WASM stack and linear memory, so these were already available as separate C functions at the point the experimental compartmentalisation work was started.

Therefore the following table lists all the functions which were moved to the capability manager and made a proxy function inside the compartment. Note that the underlying functionality of these is unchanged (they still allocate off the native heap and additionally log metrics):

| Function | Details |
| --- | --- |
| *cheri\_malloc\_wrapper* | Allocates general purpose memory for WAMR (wraps *malloc()*) |
| *cheri\_realloc\_wrapper* | Reallocates general purpose memory for WAMR (i.e a resize of an existing allocation) (wraps *realloc()*) |
| *cheri\_alloc\_linear\_mem\_wrapper* | Allocates WASM linear memory block (wraps *malloc()*) |
| *cheri\_alloc\_stack\_wrapper* | Allocates WASM frame stack (wraps *malloc()*) and stack data structures |
| *cheri\_free\_stack\_wrapper* | Frees WASM frame stack (wraps *free()*) and stack data structures |
| *cheri\_free\_wrapper* | Frees allocated memory (wraps *free()*) |

## Testing and Results

This section describes the testing that was performed on the experimental compartmentalised version of WAMR and the results which were obtained. As this is not the final product the level of testing was not as rigorous as for the WAMR port.

### WASI Test Suite Updates

For compartmentalised WAMR it is necessary to load the dynamic shared object library that contains the WAMR core code. The *iwasm* program has the new argument *–wamr-lib=<library>* added to support this.

Due to the way the WASI test suite, when running remotely on Morello, copies files then it was not possible for the default argument to be used for the library to load. This meant a new WASI test suite adapter file was needed, which would be able to read the *–wamr-lib* argument from JSON settings config file and pass it to the command line.

A screenshot of a config file is shown below with the new parameter shown in red. This specifies the file which needs to be loaded for the WAMR library:



### Untestable Features

AOT is not yet supported in the compartmentalised WAMR and therefore WASI test suite configs which test AOT as well as interpreter modes would not work, and manual testing of AOT files is not possible at this time.

Loading of user native code is not supported yet in compartmentalized WAMR and therefore any tests which require loading a native code library cannot be executed at this time. Note that there are no automated WASI test suite tests which use native code libraries, as WASI test suite does not have support to load such a library.

Testing of the external references feature with the external references bespoke application and/or native library is also not currently possible because of the inability to load native libraries.

Hybrid-cap mode cannot be supported with proper compartmentalisation and therefore all tests can only be performed in pure-capability mode. In any case, hybrid-cap is being deprecated by Arm.

Note that system calls from the compartment should not work, but they do work because Arm have not yet added security policies for their use. Ahead of this time, WAMR will need splitting such that any Linux system calls will need to be handled via capability manager service callbacks.

### Results

Results were identical to the non-compartmentalised product testing, i.e the compartmentalisation introduced no defects, other than when test could not be run as described due to incomplete functionality.

The results then are:

*Manual*

2 of 5 manual tests could be run, both passed.

*WASI Test Suite - Automated*

| Target | WAMR Mode | Passed | Failed | Notes |
| --- | --- | --- | --- | --- |
| Morello pure-cap | Interp | 45 of 48 | 3 of 48 | See below re. failures |

*\*Failing tests are the same as those for the non-compartmentalised product, none of these issues relates to CHERI modifications*

**OVERALL RESULTS**

| Total Tests | Run (Automated) | Pass | Fail | Coverage (%) | Pass Rate (%) |
| --- | --- | --- | --- | --- | --- |
| 50 | 48 | **47** | 3 | **96%** | **94%** |

***Note: “****Total tests” includes a potential 5 x 6 = 30 tests which cannot yet be automated.*

## Compartmentalisation Conclusions and Future Work

The initial compartmentalisation of WAMR has, so far, proven to be successful. WAMR has proven a good candidate for compartmentalisation and the “double-sandboxing” works well for WAMR.

The frameworks put in place are a good step in order to make it possible to achieve the further enhancements needed to make WAMR fully compartmentalised. Additionally the complicated problem of dealing with global symbol relocations was solved in a way that allowed work to continue.

The remainder of this section discusses future work suggestions and system deficiencies which need addressing.

### Future Work Proposals

#### Support the LLVM Toolchain

This experimental WAMR has so far used the Gnu Toolchain (gcc) only, due to earlier issues with LLVM. The two toolchains have a different method to implementing *dlload()* for a shared object runtime load and Morello LLVM ELF files support the *.capinit* section which may make it easier to perform the symbol relocation patchups.

It is essential that the system can build and work with LLVM and so this should be investigated, and is unlikely to be large task.

#### Split WAMR into Capability Manager and Compartment

WAMR is a complex, optimised software application. A large part of WAMR is related to actually parsing and executing WASM, but it also includes numerous low-level parts for example:

* + WASI operations which use native code
  + Invoker for calling user and system native functions
  + Memory and heap management utilities
  + System function wrappers for logging, file access, stdio and threading functions

These low-level parts of WAMR need to be moved to the capability manager and made available as capability manager service callback services. This is a non-trivial task as WAMR is not well partitioned into these different areas of functionality.

#### WASM Function calling through Capability Manager Services

All WASM calls to a sub-WASM function or native function should cause an invocation into a different compartment with a switch through the capability manager. Whilst not simple, this task should not be excessive as the framework already provides most of these mechanisms.

#### Support for Multiple Compartments

The capability manager currently only supports a single compartment, but it should be reasonably simple to modify it to manage a list of compartments.

Multiple compartments (each with their own stack) would provide greater security and also open the route to multi-modules. It also improves security by allowing individual sealer capabilities for compartment -> capability manager calls.

#### Improve the Capability Symbol Patching Mechanism

At the time of writing the Arm lib C does not seem to support a mechanism to load a library entirely into the restricted state, and no examples have been provided which deal with the problem of loading large numbers of library functions *en masse* into the same restricted state compartment.

When these aspects are better supported then the symbol patching process that happens at the beginning of the compartmentalised WAMR flow could be improved, or hopefully removed. This is currently a not insignificant delay in processing before the WASM application starts, and is undoing steps that did not need to be done in the first place.

Although there seems to be no official support for this, Verifoxx has started working on an experimental port of the GNU LibC (glibc) which will perform all the capability relocations once as part of standard dynamic library loading. This is explained more fully in Section 13.

#### Improved Compartment Model

Once multiple compartments are implemented then the mechanism for crossing between capability manager and compartment can be improved and made more secure.

A simple way to achieve this can be to avoid passing sealer capabilities across the boundary and instead allow these to be stored in the compartment metadata within the capability manager.

More difficult tasks involve looking again at the use of partial function application techniques and possibly reflection in order to make it easier to extend the compartment function call and service callback function calling frameworks. Ideally this can all be done automatically within the codebase, but worse case C macros and / or external code generation tools could be used to avoid the need for any copy/pasting when writing the boilerplate code in the framework.

Ideally, adding a new function to be run in WAMR using a proxy should be as easy as just using it or at the least a few lines of boilerplate code that need to be written – and the framework should take care of the rest.

#### Native Code Support

Native code support will be required and should be simple once other steps are put in place. This is likely to need a dynamically linked capability manager executable, but this is already supported.

Once this is available then full testing of native user code modules and the external references feature would be possible.

#### Multi-Module Support

Multi-modules are not yet supported in the core *linux-cheri-purecap* WAMR product and so would need adding here first. It would then be convenient to have a dedicated set of compartment(s) for each module, and this should be reasonably easily achievable once other work as outlined above has been implemented.

#### Better Resource Management

The service callback mechanism has so far been used to support only the basic memory allocation functions for the WASM application. Once WAMR has been split into compartment and capability manager parts then the number of methods using this framework will be greatly increased.

However, more can be put into the capability manager in terms of managing the memory resources used by a compartment and different parts of WAMR. Each compartment can be given its own heap, to be managed e.g with a memory pool (WAMR core can support this), and WASM structures such as application linear memory can be tightly controlled as to which compartment has access.

There is a lot of scope in this area and an investigation would be needed to see what can be achieved, and so timescales are unknown currently for this aspect of potential future work.

#### Compartmentalised WAMR as the main WAMR CHERI Product

Currently we have the “mainline” CHERI product and an experimental compartmentalised version. In time it may be decided to make a single CHERI product using only compartmentalised WAMR. Additionally, in the long-term the CHERI changes should feedback into the main WAMR project.

This would probably be a late stage of the work, which would require buy-in from the whole WAMR community. As part of this step, the CHERI-WAMR codebase would need updating to the latest WAMR-upstream top-of-tree. Unfortunately issues with repository management have to date prevented regular pulls of mainstream WAMR, but now these are resolved it would be better to take this pain sooner rather than later.

### System Deficiencies

It would appear to be the case that the Morello Linux SW support for compartmentalisation is currently still quite immature, and not able to utilise what the firmware allows. This experimental phase of the project has made more difficult due to this; although these issues may have been mentioned elsewhere they are summarised here for completeness.

#### Lack of Support for Bulk Compartmentalisation

The “double-sandboxing” approach for WAMR relies on running almost all (and currently all) WAMR code inside a compartment. There is no implicit support in the Linux system for doing this; examples so far are either manually loading ELF file code or cherry-picking individual functions to remap.

The logical approach to follow would be to import all code in runtime via a shared object library, and this is what we have done. However it then requires patching all symbols from the relocation tables since these are loaded with executive permissions.

A policy is needed in the Linux system to control the loader processing: a recommendation, and simple way of doing this, would be to provide an additional argument to *dlopen()* to specify the root capability to use for all symbol loading. Currently the auxiliary vector root for the executable is always used, which is not acceptable.

Perhaps instead there could be a *dl****c****open* function, specified like this:

**void \*dlcopen(const char \****filename***, int** *flags,* **void \****rootcap***);**

#### Lack of System Call Security Policy

Linux System Calls are currently possible without consideration of the PE state. At the least, this should change so system calls are not possible in the restricted state. What would be better is to be able to apply a policy to control what is available in a non-executive state, but allow some calls to work in a meaningful way.

For example, WAMR attempts to find out if it has enough stack to run by using the *pthread\_attr\_getstacksize()* call. As it stands, if this is called while in PE restricted state it (unsurprisingly) reports the size of the main (executive) stack. One option is to have this call fail if in the wrong state, but what could be better is to allow it to report the size of the current (i.e restricted) stack, even though *pthread\_attr\_****set****stacksize()* should likely not be allowed by default.

#### Sealer Capabilities

Sealing capabilities also do not have a policy at the moment, although Arm are aware of this. The root sealer cap allows any sealer to be constructed as it is bound to the whole of the memory range.

# Experimental GNU C Library Port to Support Compartments

The experimental compartmentalisation framework described in the previous section made it possible to run WAMR in the *executive PE state* and then call through to compartments running in *restricted PE state*. In order for this to work it was necessary to patch the capability relocation tables so that all calls to take place in the compartment would load a restricted capability. The patch is necessary to remove the Executive Permission from any PCC relative capability in the relocation table.

As was pointed out, this is wasteful because as part of loading the dynamic library via *dlopen()* or *dlmopen(),* the GNU libC loader would be building the relocation tables in the first place. What would be more efficient would be for these compartment capabilities to be set correctly by the loader.

As this is not supported in the Morello port of the GNU C Library, so as a work package extension we started creating a port which would implement this. Unfortunately due to time constraints it was not possible to complete the work but progress so far and planned work is documented here for reference.

## Overview of Loader Operation

We are concerned only with loading a library, not launching an executable. As for the compartmentalisation “capability manager”, the intention is that an executive application will load a library (and hence all dependent libraries) into a restricted state for execution within a compartment.

The dynamic library / libraries (shared object(s)) are loaded using the *dlopen()* or *dlmopen()* functions. The difference is that *dlmopen()* will load into a new namespace; for our purposes *dlmopen()* must be used unless the loading executable is statically linked. This is because all dependent libraries need to also be loaded into a compartment and so there may be multiple versions (e.g a libC for the compartment and another for the executable). This has already been explained in the compartmentalisation section and will not be discussed further here.

Documentation on *dlopen()* and *dlmopen()* can be found here: <https://man7.org/linux/man-pages/man3/dlopen.3.html> (or any other Linux manual). It can be seen that the API functions take a *flags* argument. This would be a suitable place to request loading a library into a compartment; we can simply define a new flag to be used to request this operation.

A very top-level overview of what happens when *dlopen()* runs is given below (*dlmopen()* the same, other than it will declare a new namespace):

A diagram of a software program

Description automatically generated

Note that the implementation of the GLibC is such that there are machine-specific routines which get called to actually perform relocations. These are already implemented for Morello, as this is CHERI specific.

**ADDITIONAL NOTE:** Not shown in the diagram is what happens on library *unloading*. Calls to any *dl\_fini* symbols are made. These can be considered as the reverse to *dl\_init* symbols.

### Modifications for Compartment Loading

The first aspect is to detect if loading into a compartment is requested. This would be handled via an additional flag value passed in the *flags* argument to *dl[m]open()*. This would have to be a CHERI only flag, as it would be meaningless for other architectures. It is still mandatory to support loading which does *not* load into a compartment – in fact this is the default when any application first launches.

With reference the loader operation, the following aspects would need modifying in order to support loading into a compartment:

**Library Enumeration**

We need to ensure that if the loader itself is being loaded (*ld.so*) then this is not subject to being compartmentalised.

**Symbol Patching**

When iterating the relocation tables, if loading into a compartment then any PCC relative symbols need the Executive permission to be removed.

**Libc Early Init**

*Libc\_early\_init* is the process whereby a well-defined API call into the loaded libC library (if this is indeed loaded as a dependency) needs to be made by the loader, because it is needed early on in the lifecycle of the loaded shared object.

A mechanism is needed to handle this. Either this call needs to be made in the executive state, and then (re-)patch any symbols afterwards, or this call needs to involve a transition from executive to restricted state. Although the latter is complex it is preferred as it means no re-patching of symbols is necessary.

**Call Initialisation Functions**

The shared object can advertise two sorts of functions to be called as part of early init by the loader (the below is a simplification of what is actually supplied in the ELF data):

* + *DL\_INIT* : A single function to be called
  + *DL\_INITARRAY* : An array of function pointers which must be called

These functions will, like *libc\_early\_init*, need to be called with a transition from executive to restricted state.

*(On Unload)* **Call Finalisation Functions**

When the library is unloaded by the loader, the finalisation functions must be called. Like the INIT functions these can either be a single function or an array of functions.

These must also involve a transition from executive to restricted in our case – but only if the library was loaded as a “compartment” library. This is more complicated in our case because it means the loader must store in its link map (structures which represent the loaded library in memory) whether the library(s) were loaded into a compartment or not.

## Implementation

Verifoxx have created mirrored the glibc github repository. The Verifoxx mirror, which includes work-in-progress latest changes, can be found here: <https://github.com/Verifoxx-LTD/verifoxx-glibc-morello-compartment>.  
The default branch is cloned from the ARM Morello main branch, and is *arm/morello/verifoxx-main*

*(Note: explicit access may need to be requested, at the time of writing the repository is private).*

### Creating the Mirror: Obtaining and Building GLibC

The GNU C Library is made available at the official source of <https://sourceware.org/git/glibc.git> or one of the mirrors. For our experimental version we are using a fork of the *arm/morello/main* branch. This is obtainable easily from git, but for a web interface see the mirror on the Morello gitlab pages: <https://git.morello-project.org/morello/gnu-toolchain/glibc>. The specific commit we are using is *33cb9de5cba0e3b428a2bab4bd8368bf55806430* (committed 4/4/23) which as of 28th January 2025 is HEAD on the branch.

The codebase was first built for Morello in the normal way; first the *configure* script is run and then *make* is used in a build folder in order to build all the libraries. We chose to use a separate output folder and then linked against this folder in our installation directory, which we then used in our test code, which was itself a modified version of the experimental CHERI-WARM Compartmentalisation code.

The exact configure command used is given below:

cd build

../configure --prefix=<OUT\_FOLDER> --host=aarch64-none-linux-gnu --target=aarch64-none-linux-gnu --disable-werror --enable-obsolete-rpc --disable-profile --without-gd --without-cvs --without-selinux --enable-shared --with-tls --disable-profile --disable-omitfp --disable-bounded --disable-sanity-checks --includedir=<OUT\_FOLDER>/usr/include --with-headers=<INSTALL\_FOLDER>/libc/usr/include CC="aarch64-none-linux-gnu-gcc -march=morello+c64 -mabi=purecap" CXX="aarch64-none-linux-gnu-g++ -march=morello+c64 -mabi=purecap" CFLAGS="-O2 -g" CXXFLAGS="-O2 -g"

### API Modification

A new flag was added which gets included with *dlfcn.h*, but only if building for CHERI:

/\* Special flag for CHERI - load as compartment \*/ #ifdef \_\_CHERI\_PURE\_CAPABILITY\_\_ #define RTLD\_LOAD\_COMPARTMENT 0x00200 #endif

An example of this being used in the function call is given below:

dlmopen(LM\_ID\_NEWLM, so\_name.c\_str(), RTLD\_NOW | RTLD\_LOCAL | **RTLD\_LOAD\_COMPARTMENT**);

### Detecting the Loader

The loader can be detected by the fact that in the internal *link\_map* structure for the loaded library, the *link\_map* pointer is not the same as *link\_map->l\_real* pointer for *ld.so* if not in a new namespace. This is used to avoid removing executive permission in this case as it would result in problems with the loader being used by the (executive) application itself.

### Relocation Symbol Changes

Actually applying the removing of the executive permission is reasonably simple. At the point where the capability permissions are set, it is simply a case of modifying the permissions used for a PCC capability to ones with the executive permission removed.

The code snippet below shows an example of this being done: the flag *make\_symbol\_restricted* is true only if the flag to load a library to a compartment was applied, and it is not the loader:

/\* Compartment perm mask is same as data read + execute (with no executive perm) \*/

#define CAP\_PERM\_MASK\_RX\_COMP (CAP\_PERM\_MASK\_R | CAP\_PERM\_EXECUTE)

if (make\_symbol\_restricted && perm\_mask == CAP\_PERM\_MASK\_RX) {

perm\_mask = CAP\_PERM\_MASK\_RX\_COMP; /\* Restrict for a compartment. \*/

}

value = \_\_builtin\_cheri\_perms\_and(value, perm\_mask);

This change must be applied in all applicable places inside the *dl-machine.h* for the Morello system – the different symbol types are handled separately within the file. Fortunately, a separate function *morello\_relative()* is used for all relative relocations *(R\_MORELLO\_RELATIVE* and *R\_MORELLO\_IRELATIVE*) and the remaining ones are handled together *(R\_MORELLO\_CAPINIT, R\_MORELLO\_GLOB\_DAT* and *R\_MORELLO\_JUMP\_SLOT*). See the ABI specification for more detail on relocation symbols (<https://github.com/ARM-software/abi-aa/blob/main/aaelf64-morello/aaelf64-morello.rst#relocation-types>).

### Future Work

The work done so far in building, modifying and testing the updated glibc shows that symbols are being relocated correctly and compartments are restricted (executive permission removed) in the same way that was being done with our post-loader patches from the compartmentalisation CHERI-WAMR experiment. However, as would be expected, a fault occurs when attempting to call the necessary *libc\_early\_init()* or the *DL\_INIT* functions because there is a switch from executive to restricted which is not being applied as specified in the ISA.

Future work in this area would be to create a machine-specific function which can implement the assembly code needed to switch the PE state correctly. The code to this is already present, in a form, in the compartmentalisation demo code. The PE state must then be modified and correctly returned for all calls into the library after symbol relocation fixups.

The final stage would be modifying the finalisation function calling. Note that a flag has been added to the link map structure to record whether compartmentalisation was applied or not. This is *l\_compartment*, an integer set to 1 for “load into compartment” and 0 for all other cases. This should make it easier to determine if a PE state switch is required for finalisation function calls.

Another, perhaps nicer, way would be to read each finalisation function pointer to determine if it has the executive permission or not and if not then call across the compartment boundary. This would have the advantage of not depending on any previous setting and so be completely independent of the loading code.

## Conclusions

The modification of the GNU C Library to support compartmentalisation in CHERI is a useful exercise and although we have not been able to complete the work it does seems that it will be feasible.

GLibC is well structured into common code and system / machine specific code, however due to the need for the CHERI-specific flag it has been necessary to make some common code dependent on the CHERI compiler (*#if defined \_\_CHERI\_PURE\_CAPABILITY\_\_* ). However we note that in the Morello port baseline this has been done elsewhere by Arm / Linaro, so perhaps at least for now this is an acceptable technique.

It is clear, though, that at some point these changes would need to be adopted by the GLibC upstream project. Clearly the executive / restricted concept is a Morello-only technology, however the general concept of compartmentalisation loading is an overall CHERI concept. The exact implementation of what happens in symbol-patching for a compartment, or calling into a compartment, can be made system dependent and so each individual CHERI architecture can provide their own implementation.

From that perspective, it may not be unreasonable to have the API changes one day included in the main GNU C Library release.

# Issues Reference

Throughout this document various issues and bugs are described which have been found in CHERI / Morello Toolchains and associated SW, or the WAMR release from ByteCodeAlliance. These are summarised here together with a link to where the issue has been recorded.

***WAMR Bugs***

These bugs have been found in the original *ByteCodeAlliance* WAMR and are not related to the CHERI port. Currently recorded in Verifoxx’s internal issues repository, they need raising on WAMR.

| Reference Number and Type | Details | Link |
| --- | --- | --- |
| REG1-84  (Verifoxx Internal) | WAMR does not exit AOT on *wasi\_exit()* on all build targets. | <https://verifoxx.atlassian.net/browse/REG1-84> |
| REG1-98  (Verifoxx Internal) | Intermittent receive error attempting to receive from a socket (within WAMR’s WASI-sockets implementation) | <https://verifoxx.atlassian.net/issues/REG1-89> |
|  |  |  |
|  |  |  |

***WAST Test Suite Bugs***

These are found on the *WASI-Test-Suite / WASI* and need further analysis before the original developers are notified.

| Reference Number and Type | Details | Link |
| --- | --- | --- |
| REG1-76  (Verifoxx Internal) | Several supplied Rust WASI tests fail on all platforms (and *wasmtime*). | <https://verifoxx.atlassian.net/issues/REG1-76> |
|  |  |  |
|  |  |  |

***CHERI / Morello Bugs***

These issues have been found in CHERI / Morello documentation, tools or firmware.

| Reference Number and Type | Details + Component | Link |
| --- | --- | --- |
| #67  (Morello LLVM Project) | **LLVM LLDB Issue**  LLDB debugger won’t stop at breakpoint on Linux Morello pure-cap. | <https://git.morello-project.org/morello/llvm-project/-/issues/67> |
| N/A | **GNU GCC Issue**  *cheri\_pcc\_get()* builtin compiles incorrectly on GCC. | <https://community.arm.com/support-forums/f/morello-forum/54390/behaviour-of-cheri_pcc_get-on-linux-hybrid-cap-and-morello-compartments-demo> |
| N/A | **GNU GCC (C++) Issue**  Inline assembler in template function ignores registry declarations on Morello. | <https://community.arm.com/support-forums/f/morello-forum/54444/morello-gnu-g-compiler-bug---inline-assembler-in-template-function-ignores-register-declarations> |
| N/A | **LLVM Clang (LLD) (C++) Issue**  Excessive incorrect warnings when linking. | <https://community.arm.com/support-forums/f/morello-forum/54191/c-standard-library-support-in-morello-linux-pure-cap> |
| N/A  NOW RESOLVED | **LLVM Clang C++ Issue**  **BUG NOW RESOLVED – INCLUDED FOR POSTERITY**  *Exceptions in C++ do not work for static binaries.* | <https://community.arm.com/support-forums/f/morello-forum/54191/c-standard-library-support-in-morello-linux-pure-cap> |
| N/A | **GNU Objdump Tool**  Objdump is not interpreting resolved PC relative addresses correctly in some cases on Morello (llvm-objdump is ok) | <https://community.arm.com/support-forums/f/morello-forum/54769/objdump-not-interpreting-pcc-relative-address-page-correctly-for-gnu-toolchain-port> |
| Linaro Bug 6055 | **GCC Issue**  Taking address of a label, which yields a *void \**, created a capability which is not a sealed entry capability (sentry).  Clang correctly seals the capability. | <https://bugs.linaro.org/show_bug.cgi?id=6055> |

# Resources Reference

A list of resources referenced in this document and used to assist with the WAMR porting work.

| Resource Name | Reference / Link |
| --- | --- |
| Verifoxx-CHERI-WAMR Port Git Repository | <https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr> |
| WAMR Git Repository | <https://github.com/bytecodealliance/wasm-micro-runtime> |
| Morello Git Repository | <https://git.morello-project.org/morello> |
| Web Assembly Core Specification | <https://www.w3.org/TR/wasm-core-1/#runtime-structure%E2%91%A0> |
| WASI Documentation | <https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-documents.md> |
| WASI SDK | <https://github.com/WebAssembly/wasi-sdk> |
| WASI Test Suite | <https://github.com/WebAssembly/wasi-testsuite> |
| CHERI C/C++ Programming Guide | <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf> |
| Morello Architecture Extension (DDI0606) | <https://developer.arm.com/documentation/ddi0606/latest/> |
| Arm Morello System Development Platform Technical Manual | <https://developer.arm.com/documentation/102278/latest> |
| Morello Prototype Architecture Overview | <https://developer.arm.com/documentation/den0133/0100/Morello-prototype-architecture> |
| Armv8-A Instruction Set Architecture | <https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/Armv8-A%20Instruction%20Set%20Architecture.pdf?revision=ebf53406-04fd-4c67-a485-1b329febfb3e> |
| Morello Development Platform Getting Started Guide | <https://developer.arm.com/documentation/den0132/latest/> |
| CHERI Instruction Set Architecture | <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf> |
| CHERI HW Compartmentalization | <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-887.html> |
| Morello pure capability kernel user Linux ABI specification | <https://git.morello-project.org/morello/kernel/linux/-/wikis/Morello-pure-capability-kernel-user-Linux-ABI-specification#mapping-permissions-and-capability-permissions> |
| Morello in AArch64 Linux | <https://git.morello-project.org/morello/kernel/linux/-/blob/morello/master/Documentation/arm64/morello.rst> |
| Morello Extensions to ELF | <https://github.com/ARM-software/abi-aa/blob/main/aaelf64-morello/aaelf64-morello.rst> |
| ELF for AArch64 | <https://github.com/ARM-software/abi-aa/blob/main/aaelf64/aaelf64.rst> |
| Morello Extensions to the Procedure Call Standard | <https://github.com/ARM-software/abi-aa/blob/2982a9f3b512a5bfdc9e3fea5d3b298f9165c36b/aapcs64-morello/aapcs64-morello.rst> |
| Arm CapInfo Tool | <https://www.morello-project.org/capinfo> |
| Arm Morello Release 1.5 | <https://git.morello-project.org/morello/docs/-/blob/morello/release-1.5/user-guide.rst> |
| Ubuntu Distributions | <https://releases.ubuntu.com> |
| WSL2 X-Server | <https://sourceforge.net/projects/vcxsrv/> |
| Debian Distributions | <http://deb.debian.org/debian>/ |
| PuTTy Terminal Emulator | <https://www.putty.org/> |
| PuTTy Session Manager | <https://puttysm.sourceforge.io/> |
| GNU Toolchain Morello port | <https://developer.arm.com/downloads/-/arm-gnu-toolchain-for-morello-downloads> |
| LLVM Morello Port | <https://git.morello-project.org/morello/llvm-project> |
| MUSL-libC for Morello | <https://git.morello-project.org/morello/musl-libc/-/tree/morello-release-1.5.0> |
| CMake | <https://cmake.org/> |
| Visual Studio CMake Linux Project Information | <https://learn.microsoft.com/en-us/cpp/linux/cmake-linux-project?view=msvc-170> |
| Web Assembly Binary Toolkit | <https://github.com/WebAssembly/wabt> |
| WAMR Fast Interpreter Information | <https://www.intel.cn/content/www/cn/zh/developer/articles/technical/webassembly-interpreter-design-wasm-micro-runtime.html> |
| WAMR Documentation | <https://wamr.gitbook.io/document/> |
| LLVM Infrastructure | <https://llvm.org/> |
| LLVM API Reference | <https://llvm.org/doxygen/> |
| Morello GNU C Library | <https://git.morello-project.org/morello/gnu-toolchain/glibc> (mirror) |
| Morello Benchmark ABI | <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-986.pdf> |
| Morello Early Performance Results | <https://ctsrd-cheri.github.io/morello-early-performance-results/performance-analysis-spec/initial-results.html> |
| “Cherifying Linux: A practical view on using CHERI” | <https://dl.acm.org/doi/pdf/10.1145/3642974.3652282> |