Morello MicroPython: A Python Interpreter for CHERI

Duncan Lowther
University of Glasgow
Glasgow, United Kingdom
duncan.lowther@glasgow.ac.uk

Dejice Jacob
University of Glasgow
Glasgow, United Kingdom
dejice.jacob@glasgow.ac.uk

Jeremy Singer
University of Glasgow
Glasgow, United Kingdom
jeremy.singer@glasgow.ac.uk

Abstract

Arm Morello is a prototype system that supports CHERI hardware capabilities for improving runtime security. As Morello becomes more widely available, there is a growing effort to port open source code projects to this novel platform. Although high-level applications generally need minimal code refactoring for CHERI compatibility, low-level systems code bases require significant modification to comply with the stringent memory safety constraints that are dynamically enforced by Morello. In this paper, we describe our work on porting the MicroPython interpreter to Morello with the CheriBSD OS. Our key contribution is to present a set of generic lessons for adapting managed runtime execution environments to CHERI, including (1) a characterization of necessary source code changes, (2) an evaluation of runtime performance of the interpreter on Morello, and (3) a demonstration of pragmatic memory safety bug detection. Although MicroPython is a lightweight interpreter, mostly written in C, we believe that the changes we have implemented and the lessons we have learned are more widely applicable. To the best of our knowledge, this is the first published description of meaningful experience for scripting language runtime engineering with CHERI and Morello.

CCS Concepts: • **Software and its engineering** → **Interpreters**; • **Security and privacy** → *Virtualization and security*.

Keywords: CHERI, capabilities, software implementation

ACM Reference Format:
Duncan Lowther, Dejice Jacob, and Jeremy Singer. 2023. Morello MicroPython: A Python Interpreter for CHERI. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '23), October 22, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3617651.3622991

1 Introduction

In 2021, Arm released a prototype platform code-named 'Morello' [2, 9] which realizes the CHERI hardware capability concept [22, 25] in an industrial strength microprocessor. A *capability* is a double-width 'fat' pointer that includes metadata for address bounds and access permissions. Additionally, CHERI capabilities have an out-of-band *tag* to ensure pointer validity. The premise of hardware capabilities is that entire classes of memory vulnerabilities can be eliminated, including spatial bugs (i.e. out-of-bounds reads and writes) and temporal bugs (i.e. use-after-free bugs) [12].

In this paper, we describe our experience and lessons learned during a full port of the MicroPython framework to Morello. We modify the C source code of MicroPython in order to provide runtime awareness of CHERI capabilities. There were two logical stages to this work: firstly the MicroPython code was refactored to eliminate compiler errors and warnings, as described in Section 3; secondly memory safety enforcement of capabilities was leveraged to generate tight bounds on runtime allocations, as described in Section 4. In Section 5, we analyse test coverage for MicroPython on the Morello CheriBSD platform, which has almost identical results to the AArch64 FreeBSD platform. Section 6 characterizes the source code modifications required, showing that the code base alterations are small (0.18% of overall codebase). We note this is comparable with work reported from various other CHERI ports.

Porting MicroPython to Morello had the additional benefit of uncovering a number of latent memory safety bugs in the interpreter's code base. They were exposed by bounds checking of capabilities during execution of unit tests and benchmarks. The patches applied to fix these bugs have been merged into the upstream MicroPython repository and are discussed in Section 7. In this way, we demonstrate concretely that a CHERI port of an interpreter can also provide memory safety improvements for non-CHERI users.

Further, we provide a performance analysis of MicroPython running on Morello in Section 8. Taking into account that (1) Morello is a prototype platform [19, 20] and (2) MicroPython is not a high-performance runtime, we measure the overheads of capability support for MicroPython running a standard set of Python benchmarks. Our results show there is a geometric mean 73% slowdown in terms of execution time.

This work describes a complete first phase of enhancing security and resilience of MicroPython on Morello. CHERI capabilities enable runtime bounds checking for memory access in the MicroPython interpreter. Other possible applications of capabilities such as supporting software compartmentalization or sandboxing untrusted libraries will enhance the overall security properties of the interpreter. As we consider advanced use cases for capabilities, we outline a potential agenda for future development in Section 9.

2 Background

2.1 What is Morello?

CHERI [22, 25] specifies an abstract set of processor extensions to support hardware capabilities for accessing memory. The CHERI Concentrate compression technique [24] ensures both pointer data and metadata can be stored in a double machine word, i.e. 128 bits on a 64 bit architecture. A further single bit tag denoting the validity of a capability is stored out-of-band, referred to as the *129th bit*. This hardware tagging prevents capability values from being forged by untrusted code. Dynamic checks take place in hardware on each memory access, to ensure:

1. the capability is valid (tag check)
2. the capability has appropriate access permission (permission check)
3. the capability address is within bounds for this memory access (bounds check)

If any check fails, the hardware raises a capability violation exception. This is transmitted from kernel-space to user-space as a SIGPROT signal and execution of code is interrupted.

Morello [2, 9] is the Arm instantiation of the CHERI concept. It is a quad-core AArch64 server class system-on-chip, based on the Neoverse N1 core, enhanced to support 128-bit architectural capability values. There are appropriate new instructions, along with modifications to the register file and memory hierarchy. The CPU clock speed is 2.5GHz; the Arm instruction set architecture version is Arm v8.2-A with Morello extensions.

The Morello platform runs CheriBSD, a capability-aware variant of FreeBSD. Within CheriBSD, user code may run in *hybrid* or *purecap* mode. *Hybrid* mode involves executing standard AArch64 code, with support for executing capabilities. Only pointers explicitly annotated in source code are treated as capabilities. Effectively, this code runs in a process-level sandbox, with all raw pointers being converted to capabilities with the widest possible permissions and bounds by setting the global base capabilities in the Morello register set (e.g. the default data capability, DDC). The *purecap* mode involves executing Morello code with capability support. All references within purecap applications are represented as capabilities directly. This is the preferred execution mode for CheriBSD applications, since it provides finer-grained capability support.

While Morello is a server-class processor, there is ongoing work to create smaller, embedded variants of the CHERI concept, e.g. [1, 26].

The typical language for developing CHERI applications currently is C [23]. CHERI C is a variant of C, with built-in intrinsics to handle capabilities and additional semantic constraints around pointer accesses.

2.2 What is MicroPython?

MicroPython [5] is a lightweight interpretive Python runtime, typically for targeting microcontroller scale devices. Although such systems usually run applications that are compiled from low-level languages like C, MicroPython provides a feasible alternative paradigm for development and deployment. Scripting languages enable rapid prototyping; also, Python is highly accessible for educational usage [8, 17].

MicroPython supports almost all of the Python 3 language, with a few minor omissions [6]. Textual source code is parsed and compiled to a compact bytecode format. This bytecode is interpreted at runtime by the interpreter. MicroPython does not feature a JIT compiler.

Typical interpreter optimisations implemented by MicroPython include interned strings, small integers embedded directly within tagged pointers, optimised method calls, Python stack frames hosted on the C stack, garbage collection without reference counting, and exceptions using custom non-local returns implemented with inline assembly.

MicroPython is written in C; the project is 307 kSLOC. Ports are available for common microcontroller families including Arm CortexM and ESP32. In addition, there is a POSIX process-level port which means MicroPython can be compiled and run as a hosted Unix executable. With respect to Arm instruction sets, MicroPython currently has support for AArch64, as well as Arm32 and Thumb.

2.3 Related Work

This section briefly surveys related projects that also investigate the development of managed runtimes on capability platforms.

JavaScriptCore (JSC) is a mature and highly performant JavaScript engine used in popular applications such as the Safari browser. The JSC runtime has been partially ported to CHERI [10], including the interpreter and a baseline tier of the JIT compiler. It represents the fullest exploration to date of a managed runtime on Morello. All changes to JSC are grouped into one of the following categories:

1. support for JavaScript execution.
2. support for VM-specific value representation.
3. C++ pointer use.

This port adds hardware supported bounds checking to VM allocations for spatial safety in heap memory. However the analysis does not include any performance measurements.

Like our MicroPython port, the JSC work is restricted to a single logical compartment model of execution.

Morello-specific CHERI ports of the OpenJDK virtual machine [16] and the Wasm Micro Runtime [18] are in progress. For both projects, there are no published results so far.

The CPython interpreter compiles for CHERI, but we find it is missing key functionality when we try to use it in purecap mode. We are able to run a 'hello world' script but nothing more complex. However, CPython runs with full functionality in hybrid (i.e. legacy AArch64) mode. We rely on this hybrid executable for building the MicroPython interpreter using the Morello platform as build host.

The Boehm-Demers-Weiser standalone garbage collector has been ported to CHERI [11]. Like MicroPython, it implements a single generation, non-moving, mark/sweep algorithm. The use of Morello 'tags' to identify valid pointers transforms this conservative collector into a precise collector. Preliminary performance results with this framework are based on emulated execution.

A range of C runtime memory allocators have been ported to Morello, with accompanying analysis including a performance characterization [3]. They contrast hybrid and purecap execution, noting significant overheads for purecap execution in some circumstances.

3 Initial Porting Process

The initial port simply involved getting the MicroPython code to compile and run on Morello.

Of the various MicroPython configurations (also known as ports) the `minimal` configuration was initially modified for CheriBSD on Morello. The `minimal` configuration is a simple reference implementation of the interactive interpreter with no Python library support. Using the `minimal` build as a foundation, the `unix` build configuration with support for more features was ported.

The build process involved compiling MicroPython natively on a CHERIBSD/Morello system using the natively hosted LLVM Morello toolchain[1]. Both *hybrid* and *purecap* MicroPython interpreters were generated using build-tools that were *hybrid* binaries. This reduced the porting effort for MicroPython in `purecap` as the build process for MicroPython is dependent on CPython and the `purecap` variant of CPython is not yet mature (see Section 2.3).

3.1 Minimal Configuration (Hybrid)

The `minimal` build required two modifications to properly compile for the *hybrid* ABI, neither of which were specific to Morello.

1. The AArch64 non-local return (NLR) sequences required a small change due to compiler register allocation issues in the AArch64 code. A fix for this issue was submitted upstream[2] and the issue was later fixed and merged in slightly different manner[3].
2. The configuration header file needed to be modified for a BSD build. (By default, MicroPython minimal is set up to run on Linux.)

[1]clang version 13.0.0 `aarch64-unknown-freebsd14.0`

3.2 Minimal Configuration (Purecap)

3.2.1 Non-Local Returns.

MicroPython uses non-local returns (similar to `longjmp(3)`) to implement Python exception handling semantics. Its implementation uses inline assembly to save a number of (64-bit) registers to a buffer, and restores them when needed. Substitution of 64-bit X-registers for the corresponding 128-bit C-registers and doubling the memory offsets was sufficient to create the purecap version. (Since the part of the buffer holding registers is declared in the C code as an array of pointers, the compiler allocates a sufficiently large buffer for this without further changes being necessary.)

3.2.2 Capability Provenance Considerations.

CHERI requires every valid capability to be derived from exactly one other capability, so arithmetic operations on two provenance-carrying variables are ambiguous and generate compiler errors [23]. The compiler initially flagged several places where two variables of provenance-carrying types (`(u)intptr_t`) were combined in an arithmetic expression. All of these resulted from the fact that MicroPython uses `mp_int_t` and `mp_uint_t` as its standard integral types, and these are typedef'd as `intptr_t` and `uintptr_t` respectively. The vast majority of places that `mp_(u)int_t` are used do not involve pointers and should not carry provenance, so we redefined those types as 64-bit integral types.

In a few places, however, `mp_uint_t` was used to hold values that may be pointers, most notably the `mp_obj_t` type, which can either represent a pointer to an object struct or a small integer. We changed these sites to use `uintptr_t` instead to preserve provenance. There were also several places where a pointer was cast through the `mp_uint_t` type to another type: in these cases, the provenance-discarding cast was removed.

Both hybrid and purecap builds of the `minimal` configuration were successfully tested with simple interactive Python scripts. However, the full test suite included with MicroPython requires the ability to run a Python file specified as a command-line argument, which `minimal` does not support. For this reason, we shifted to a configuration with more advanced features.

3.3 Unix Configuration (Purecap)

In order to use the MicroPython test suite, we ported the `unix` configuration to Morello. This configuration adds features that are disabled in `minimal`, including support for

[2]https://github.com/micropython/micropython/pull/11716
[3]https://github.com/micropython/micropython/pull/11762

Table 1. MicroPython code changes required by category and symptom. (M) denotes symptoms discovered on the `minimal` build, while (U) denotes symptoms discovered on the `unix` build.

Symptom	Asm	UP	PL	Bnds	PS
Compiler error (M)	-	2	2	-	-
SIGPROT (M)	2	-	-	-	1
Compiler error (U)	-	-	1	-	-
SIGPROT (U)	-	-	11	2	-
Incorrect results	-	-	-	-	1
Total	2	2	14	2	2

reading and writing files. However, we disabled the FFI and SSL modules in the first instance to avoid issues with their dependencies; for instance, the FFI module depends on `libffi`. The purecap Morello port of `libffi` does not currently support closures, which MicroPython's FFI module requires in order to wrap Python functions as callbacks to be passed to native functions.

3.4 Required Modifications

Running the test suite uncovered several problems. Importantly, with one exception (an incorrect pointer-shape assumption in the `mp_classify_fp_as_int()` function that caused five floating-point tests to return incorrect results without a crash), all problems caused the relevant tests to crash with a SIGPROT signal, making them relatively easy to identify and debug.

Table 1 breaks down the changes required by category and symptom, counting each distinct source code location where a change was required. We explain the categories of changes, and map these onto the existing CheriABI classification (refer to Table 2 in [4]).

- *Assembly (Asm)* changes were those required to adapt existing inline assembly to use the capability registers, mostly corresponding to the calling convention (CC) patch of the CheriABI classification.
- *Unnecessary Provenance (UP)* refers to integer variables (not intended to store pointers) that are defined to have unnecessary provenance-carrying types, which cause compiler errors due to ambiguity in arithmetic expressions and takes up more space unnecessarily.
- *Provenance Loss (PL)* refers to casts through integer types which clear the tag on a capability that must be later used. Both of these categories belong to the integer provenance (IP) style of patch in the CheriABI classification.
- *Bounds (Bnds)* refers to attempts to use a capability to access an address outside its bounds, which breaks monotonicity (M) in the CheriABI scheme. Monotonicity refers to the CHERI principle that the only way to create a valid fresh capability is by restricting the bounds or permissions of an existing capability; it is thus not possible to widen capability bounds.
- *Pointer shape (PS)*, as in the CheriABI classification, refers to places where the existing code expects the size of a pointer to match that of another integer type.

4 Capability Bounds on Heap Allocations

So far, the changes we have made have been the least required to build and run MicroPython on Morello under the purecap CHERI ABI. The compiler automatically sets suitable bounds on pointers to local variables and other stack allocations, and the runtime linker similarly sets suitable bounds on pointers to global variables [23]. The C library also sets bounds on heap pointers allocated with `malloc()` and `mmap()`; however, MicroPython uses its own internal memory allocator and garbage collector (simple mark/sweep) for dynamic heap allocations. As this allocator is not CHERI-aware, the pointers it returns will have the same bounds as those from which they are derived, which cover an entire heap area.

The logical next step, then, was to set tight bounds on these heap allocations. To do this, we modified the internal interpreter `gc_alloc()` and `gc_realloc()` functions to use CHERI intrinsics to set tight bounds on all returned capabilities. Note that the `gc_realloc()` function also had to be modified to *widen* the bounds of the input capability, using the base heap area capability as the provenance source, when the allocation is expanded in place.

These changes caused bounds faults to be raised in some of the test cases. The underlying issues here fall into two categories: buffer overflows and issues with bounds on in-place reallocated pointers. The first category is covered in Section 7. The second category involves two areas: parsing and string interning. (In MicroPython an interned string is referred to as a uniQue STRing or `QSTR`.)

Parsing and string interning perform their own memory management, requesting block allocations from the garbage collector and breaking them down as needed. When one of these modules runs out of memory in its current block, it asks the garbage collector to reallocate the block in-place (i.e., to extend the allocation). Because it has restricted its call to an in-place reallocation only, the code here assumes that the return value of `m_realloc_maybe()` can (after checking it for `NULL` to determine success or failure) be safely discarded. However, with CHERI bounds on these allocations, the return value must be saved back into the block pointer in order to ensure the bounds are properly updated.

There are various vulnerabilities which are mitigated by this enforcement of tight capability bounds. For instance, the `uctypes` MicroPython module exposes several functions to manipulate native pointers in interpreted Python code. These pointers can be easily abused on a non-CHERI system to gain read/write access to a large portion of the heap. A simplified

```python
import uctypes as uct
tiny = bytearray(1)
ptr = uct.addressof(tiny)
unsafe = uct.bytearray_at(ptr, 16384)
unsafe[200] = 0x1f
```

Listing 1. A simple buffer overflow attack. MicroPython's `uctypes` arrays exposes a pointer to raw memory. In *purecap* mode access outside the bounds will trap with a `SIGPROT`.

example is given in Figure 1. Note that as Python functions are objects stored on the heap, this allows manipulation of code as well as data.

In our *purecap* build, this vulnerability does not exist, as the bounds of the initial allocation stay with the pointer and later arbitrary expansion without explicit re-allocation is prohibited, causing a `SIGPROT` error.

5 Full Coverage Tests

To round off our port, we reintegrated the FFI and SSL modules (disabling FFI callbacks to avoid the particular issue with closures). We also compiled for the `coverage` variant of the `unix` port, which includes a few more features for maximum test coverage, ran the *hybrid* and *purecap* builds through the MicroPython test suite, and compared the results with three reference builds of the same variant compiled and run on non-Morello targets. The test results are shown in Table 2.

Table 2. MicroPython test suite results (commit de98805c9504703f153f0a56a7af8ee78a84eea2, `unix` port, `coverage` variant)

OS/Architecture	Pass	Fail	Skip
Ubuntu 20.04/x86-64	920	0	8
Ubuntu 20.04/AArch64	874	1	53
FreeBSD 13.2/AArch64	875	0	53
CheriBSD 22.12/Morello (hybrid)	875	0	53
CheriBSD 22.12/Morello (purecap)	874	1	53

The single failed test in the *purecap* build is due to lack of support for FFI callbacks, and apart from this, our builds passed all of the tests that those for other AArch64 targets passed. There were 8 tests which all targets skipped and 45 tests which are skipped on the AArch64 targets but not on x86-64, the latter relating to the option to compile to native machine code rather than MPY bytecode, which does not yet have an AArch64 backend, and to the related Viper extensions for native types.

6 Total Modifications

In total we modified 533 lines of code (510 additions and 23 deletions) in the original MicroPython project. Of these, 5 additions and 5 deletions were upstreamed fixes, and the rest were Morello specific. This comes to just under 0.18% of the existing codebase, roughly seven times the 0.026% reported for the KDE port [21]. This might be expected, given we are dealing with more low-level systems code.

However, over half of our additions can be attributed to two source files that were added wholesale, accounting for 252 lines of code between them. The `nlrmorello.c` file (87 new lines) specifies the inline assembly for non-local returns, which is identical to that in the previous `nlraarch64.c` file except for its use of 128-bit C-registers in place of 64-bit X-registers. The `objcap.c` file (165 new lines) creates a new built-in Python type to wrap pointers when dealing with native structures. If these are discounted as anomalous, the 258 remaining modifications represent only 0.086% of the existing codebase.

By way of comparison, Bramley et al. [3] report numbers between 0.4% (jemalloc) and 6.7% (dlmalloc) for CHERI-induced codebase changes to real-world memory allocator libraries. These are libraries only, not full managed runtimes. Similarly, Gutstein [10] reports a figure of 0.34% for the proportion of lines of code modified in the Morello low-level interpreter for JavaScriptCore. We assess our headline figure of 0.18% as reasonably in line with similar CHERI porting efforts.

7 Upstream Fixes

One compelling use case for Morello is to detect latent memory bugs in legacy systems code. For instance, the original CHERI developers were able to uncover multiple buffer bounds violations as they ported FreeBSD to CHERI [4]. The dynamic checks enforced by hardware capabilities are very helpful for identifying spatial safety issues in C code. While other dynamic debugging tools like Valgrind [14] and ASan [15] can be used to find memory errors, their high overhead means they are often not included in continuous integration pipelines.

During our port of MicroPython to Morello, we discovered four such problems in the existing MicroPython repository. Two of these were *test cases* which failed to allocate large enough buffers for the operations they were performing because they did not take structure padding into account. The third issue was a library function that involved string handling which attempted to read one character beyond the null terminator. The final issue was another library function, where incorrect integer division led to a buffer length calculation error: in a base64 decoding function, the output buffer needed to be no less than three-quarters the size of the input, but the calculation used was `(input_len / 4) * 3 + 1`, where the truncation from the division occurs before the multiplication and results in the buffer being as many as 2 bytes too short (an input length of 7 results in an output buffer length of 4, where the actual required length would be 6). All of these latent bugs were simple to fix once discovered, and we submitted a pull request with the relevant patches which was accepted and merged into the upstream repository on 21 June 2023.[4]

8 Performance Evaluation

To evaluate our MicroPython interpreter, we use the set of Python benchmarks from the *performance* section of the MicroPython test suite. These are mostly small-scale, compute-intensive, real-world applications.

For all quantitative measurements, we run MicroPython on Morello with CheriBSD version 22.12, taking arithmetic means of 30 runs. When we take measurements of execution in MicroPython, we pre-compile each benchmark to bytecode format so we are only reporting performance data for bytecode execution, not including the overhead of source code compilation time.

8.1 Performance Results

We want to compare *hybrid* and *purecap* execution on Morello with CheriBSD. In hybrid mode, we are effectively running the upstream AArch64 MicroPython interpreter in a capability-oblivious way. In purecap mode, we are running MicroPython with all our capability-based extensions as reported above—taking advantage of the capability support provided by Morello.

It is only fair to note that the Morello system is experimental hardware [19]. We are reporting performance figures to give an indication of worst-case overheads for runtime capability support. We are aware of some performance issues with the prototype Morello system, which are subject to ongoing investigation at Arm with the CHERI team. We believe our performance overheads could be reduced with more sophisticated Morello benchmarking techniques [20].

Figure 1 shows the runtime behaviour of Python benchmarks on the purecap interpreter, with counts normalized against the baseline hybrid interpreter. For each benchmark, we report six metrics. The leftmost (blue) bar is the wall-clock *execution time*. In general, the purecap execution time is between 1.5x and 2x hybrid, i.e. an overhead of between 50–100%.

The other four bars represent CPU performance counter data for purecap execution, normalized against the same counters in hybrid execution mode. The *instructions retired* generally tracks total time. The *cache misses* at various levels in the memory hierarchy are increased for purecap, but generally not excessively.

Figure 2 records the dynamic instruction counts for each Python benchmark we executed on the MicroPython interpreter. This data is collected using QEMU-based emulation of Morello running MicroPython on CheriBSD. Again, we executed each benchmark in both hybrid and purecap execution modes. We divide the instruction set into two distinct classes: one is conventional AArch64 instructions, the other is new Morello instructions for handling architectural capabilities. We notice that, for most benchmarks, there are many additional Morello-based instructions in purecap mode. There are a few Morello instructions in hybrid mode but these are only caused by CheriBSD system and library calls.

8.2 Performance Fixes

For certain benchmarks, like `fannkuch` and `nqueens`, we initially saw a 100x execution time overhead. Further investigation showed that the overhead was almost entirely caused by an inefficiency in the heap allocator, created by the doubling in pointer size and triggered by the use of repetitive list-slicing operations (each list-slicing operation requires at least three heap allocations: one for the slice specifier object, one for the new list object, and one for the new list contents; and the slice object is then almost immediately freed resulting in fragmentation). We adjusted two compile-time constant values: (1) allocator minimum block size to account for the increased pointer size, and (2) stack frame heap-spill threshold (which was responsible for a 3.7x overhead in `pystone`), along with minor VM optimisations reduced the geometric-mean execution time overhead to 73%. These optimisations are detailed at length elsewhere [13]. The results shown in Figure 1 were measured after these performance tuning changes.

9 Future Work

At present, we have a fully functional MicroPython interpreter on Morello, running as a process virtual machine in CheriBSD, with spatial memory safety enforced by capabilities in the interpretive runtime. The key MicroPython module that is not yet fully CHERI-compatible is the FFI module. This is only partially implemented because foreign function interface (FFI) callbacks are not yet supported by the CHERI C library port of libffi. This section explores possible future developments that might enhance our port and leverage the CHERI capabilities in new ways.

First, we plan to leverage CHERI hardware bounds checking to eliminate some of the software bounds checking performed by MicroPython in the interpreter. This should reduce runtime overhead significantly.

One of the most promising features of Morello is its support for *lightweight software compartmentalization*, although there is no clear consensus on the best way to implement compartments in Morello. Capabilities enable us to divide an executing program into mutually distrusting compartments with managed interfaces between them. We will explore the Morello compartmentalization primitives and determine how best to split up the MicroPython runtime. We aim to compartmentalize libraries and driver code, since these are often supplied by third parties and may be untrusted [7]. Another

[4]https://github.com/micropython/micropython/pull/11786

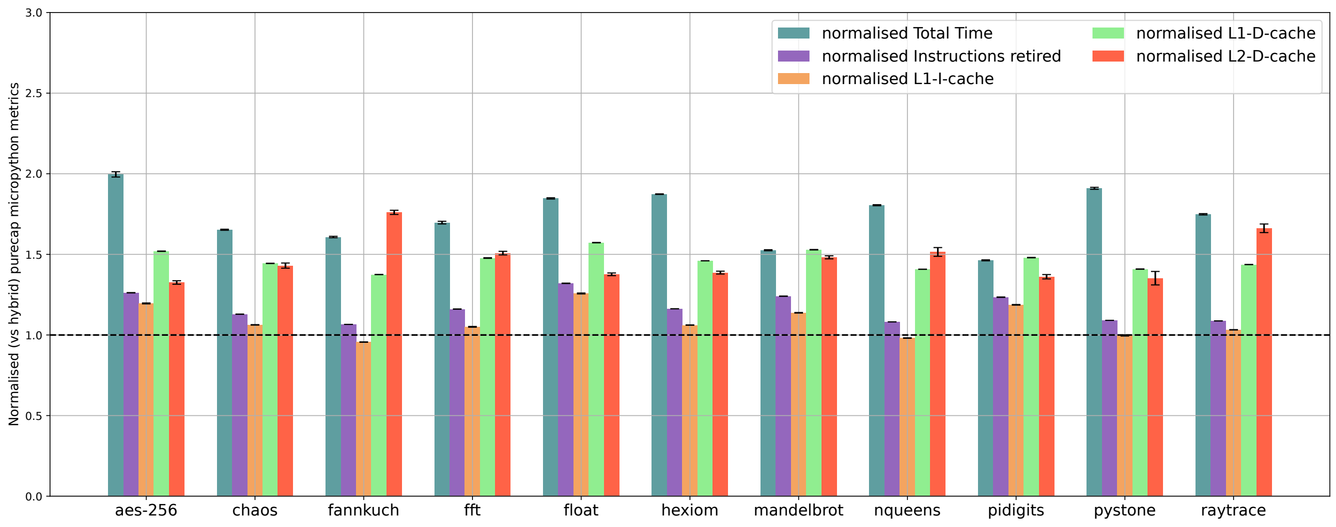

Figure 1. Performance of Python benchmarks running on the purecap interpreter, normalised to the hybrid interpreter performance. For example, the wall-clock execution time *total-time* of aes-256 on purecap is 2x greater than on hybrid. We recorded several performance metrics: *Instructions retired* is the number of instructions retired while executing the benchmark; and *{L1-I, L1-D, L2-D}-cache* the L1 instruction, L1 data, and L2 data, cache misses respectively. For clarity, the horizontal dashed line shows the point of zero overhead (normalised value 1.0).

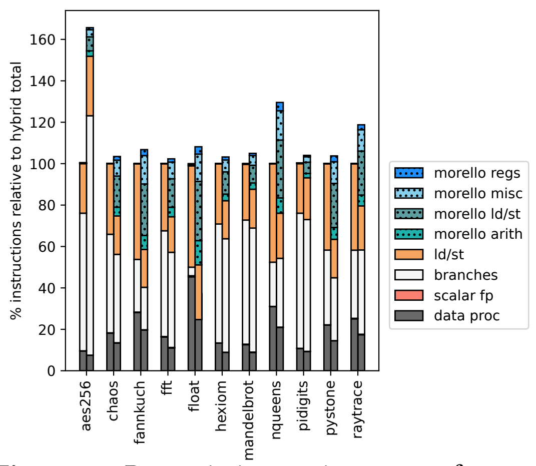

Figure 2. Dynamic instruction counts for execution of Python benchmarks. For each benchmark, there are two runs: purecap (left bar) and hybrid (right bar). For each run, we distinguish Morello instructions (dotted rectangles at top of stacked bars) from normal AArch64 instructions (plain rectangles at bottom of stacked bars).

interesting line of work would be to exploit the compartmentalization mechanism for the foreign function interface (FFI). We will further explore whether it makes sense for compartmentalization to be exposed to hosted Python applications running on MicroPython—can Python software be split into compartments using simple Python decorators?

Finally, we intend to explore the potential for running MicroPython in *baremetal mode* on Morello, rather than as a process hosted by CheriBSD. Although this is not a feasible use case for server grade Morello platforms, we are aware that researchers are developing microcontroller devices with CHERI support [1]. Baremetal MicroPython is a standard deployment scenario for microcontrollers.

10 Conclusions

In this paper, we have described our experiences during the port of the MicroPython framework to the new capability-aware Morello platform. We have indicated the challenges involved in adapting managed runtime environments for platforms that support richer pointer structures, such as CHERI-style architectural capabilities. The performance evaluation shows that there is an overhead associated with CHERI, although we are cautiously optimistic that this overhead will be reduced as industry support improves for tagged pointers and associated processor security mechanisms. We have sketched out a roadmap of future work for leveraging capabilities to add new features to MicroPython, including the notion of library sandboxing with software compartmentalization.

Although this work has focused on a small-scale MicroPython runtime and the experimental Morello platform, we hope that the lessons we have reported are useful to a wider audience with interests in language runtimes and secure processor technologies.

Acknowledgments

This work was funded by the Digital Security by Design (DSbD) programme delivered by UKRI (including grants EP/V000349/1 and EP/X015831/1), also by the UK Defence and Security Accelerator contract ACC6037520.

References

[1] Saar Amar, Tony Chen, David Chisnall, Felix Domke, Nathaniel Filardo, Kunyan Liu, Robert Norton-Wright, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. *CHERIoT: Rethinking security for low-cost embedded systems.* Technical Report MSR-TR-2023-6. Microsoft. https://www.microsoft.com/en-us/research/publication/cheriot-rethinking-security-for-low-cost-embedded-systems/

[2] Arm. 2021. Arm Architecture Reference Manual Supplement — Morello for A-profile Architecture. https://developer.arm.com/documentation/ddi0606/.

[3] Jacob Bramley, Dejice Jacob, Andrei Lascu, Jeremy Singer, and Laurence Tratt. 2023. Picking a CHERI Allocator: Security and Performance Considerations. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management.* 111–123. https://doi.org/10.1145/3591195.3595278

[4] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* 379–393. https://doi.org/10.1145/3297858.3304042

[5] Damien P. George. 2013. MicroPython. https://micropython.org.

[6] Damien P. George, Paul Sokolovsky, et al. 2023. MicroPython differences from CPython. https://docs.micropython.org/en/latest/genrst/.

[7] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. 2021. Enclosure: Language-Based Restriction of Untrusted Libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.* 255–267. https://doi.org/10.1145/3445814.3446728

[8] Linda Grandell, Mia Peltomäki, Ralph-Johan Back, and Tapio Salakoski. 2006. Why Complicate Things? Introducing Programming in High School Using Python. In *Proceedings of the 8th Australasian Conference on Computing Education.* 71–80. https://doi.org/10.5555/1151869.1151880

[9] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. 2023. The Arm Morello Evaluation Platform—Validating CHERI-Based Security in a High-Performance System. *IEEE Micro* 43, 3 (2023), 50–57. https://doi.org/10.1109/MM.2023.3264676

[10] Brett Gutstein. 2022. *Memory safety with CHERI capabilities: security analysis, language interpreters, and heap temporal safety.* Technical Report UCAM-CL-TR-975. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-975

[11] Dejice Jacob and Jeremy Singer. 2022. Capability Boehm: Challenges and Opportunities for Garbage Collection with Capability Hardware. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments.* 81–87. https://doi.org/10.1145/3516807.3516823

[12] Nicolas Joly, Saif ElSherei, and Saar Amar. 2020. Security Analysis of CHERI ISA. https://msrc.microsoft.com/blog/2020/10/security-analysis-of-cheri-isa/.

[13] Duncan Lowther, Dejice Jacob, and Jeremy Singer. 2023. CHERI Performance Enhancement for a Bytecode Interpreter. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages.* https://doi.org/10.1145/3623507.3623552 arXiv:2308.05076 [cs.PL]

[14] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 89–100. https://doi.org/10.1145/1250734.1250746

[15] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference.* 28.

[16] Soteria Project. 2023. Researching solutions for a safer web. https://soteriaresearch.org.

[17] Frank Stajano. 2000. Python in education: Raising a generation of native speakers. In *Proceedings of the 8th International Python Conference.* 24–27.

[18] UKRI. 2022. CHERI WebAssembly Micro Runtime. https://gtr.ukri.org/projects?ref=10028870.

[19] Robert N. M. Watson, Graeme Barnes, Jessica Clarke, Richard Grisenthwaite, Peter Sewell, Simon W. Moore, and Jonathan Woodruff. 2023. *Arm Morello Programme: Architectural security goals and known limitations.* Technical Report UCAM-CL-TR-982. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-982

[20] Robert N. M. Watson, Jessica Clarke, Peter Sewell, Jonathan Woodruff, Simon W. Moore, Graeme Barnes, Richard Grisenthwaite, Kathryn Stacer, Silviu Baranga, and Alexander Richardson. [n. d.]. *Early performance results from the prototype Morello microarchitecture.* Technical Report UCAM-CL-TR-986. University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500.

[21] Robert N. M. Watson, Ben Laurie, and Alex Richardson. 2021. Assessing the Viability of an Open-Source CHERI Desktop Software. https://www.capabilitieslimited.co.uk/_files/ugd/f4d681_e0f23245dace466297f20a0dbd22d371.pdf

[22] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2020. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8).* Technical Report UCAM-CL-TR-951. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-951

[23] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. 2020. *CHERI C/C++ Programming Guide.* Technical Report UCAM-CL-TR-947. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-947

[24] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. 2019. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Trans. Comput.* 68, 10 (April 2019), 1455–1469. https://doi.org/10.1109/TC.2019.2914037

[25] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecture.* 457–468. https://doi.org/10.1145/2678373.2665740

[26] Hongyan Xia. 2021. *Capability memory protection for embedded systems.* Technical Report UCAM-CL-TR-955. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-955

Received 2023-06-29; accepted 2023-07-31