An Evaluation of WebAssembly and eBPF as Offloading Mechanisms in the Context of Computational Storage

Wenjun Huang
German Aerospace Center
wenjun.huang@dlr.de

Marcus Paradies*
German Aerospace Center
marcus.paradies@dlr.de

Abstract

As the volume of data that needs to be processed continues to increase, we also see renewed interests in near-data processing in the form of computational storage, with eBPF (extended Berkeley Packet Filter) being proposed as a vehicle for computation offloading. However, discussions in this regard have so far ignored viable alternatives, and no convincing analysis has been provided. As such, we qualitatively and quantitatively evaluated eBPF against WebAssembly, a seemingly similar technology, in the context of computation offloading. This report presents our findings.

1 Introduction

Lately, a data processing paradigm called "computational storage" has been gaining traction. The idea behind it is to do "code shipping" and process data within storage devices directly, thereby reducing data movement and energy consumption [4].

While past research and commercial development in this direction have largely focused on specialized, ad-hoc hardware or software that tries to eke out as much performance as possible for specific types of workloads [41, 48, 54, 58], lately we have observed some renewed interests in enabling general computation offloading (i.e., user-programmable, dynamically loadable computation).

However, recent discussions in this direction [6, 35, 51] have focused almost exclusively on using eBPF [9] as the offloading mechanism, ignoring other platform-independent bytecodes. Naturally, we have some reservations about this trend, as there has yet been a convincing analysis showing that eBPF is the best option for the job.

In this work, we hope to initiate the conversation around general offloading mechanisms by qualitatively and quantitatively evaluating eBPF and WebAssembly. Specifically, we would like to pose and answer the following questions:

- What are the current states of the options?

- What are the advantages and disadvantages of each of the options?

- How can we improve these options with regards to computational storage (and near-data processing in general)?

2 Background

In this section, we describe briefly the various topics and building blocks relevant to our work.

Near-data processing [1, 5, 36] Near-data processing (NDP) puts computations close to data to reduce data movement. It promises a number of benefits, such as avoiding I/O bottlenecks. It can take various forms, including, but not limited to, processing in-memory (PIM) and computational storage.

The idea of NDP is not new, and research about it can be dated back to at least 2 decades ago. Nonetheless, we note that recently there has been a resurgence of interests in NDP, both in academia and in industry (e.g., [2, 45]).

Computational Storage [4, 35] Strictly speaking, computational storage is a special case of NDP that concerns only with processing data inside of storage devices. However, in practice, computational storage is not always implemented in storage (e.g., in the *computational storage array (CSA)* architecture). In that sense, *near-storage processing* is perhaps a better name for it.

Storage devices that support this kind of data processing are called *computational storage devices (CSDs)*, and there are already a few generally available CSDs on the market (e.g., [50, 52]). Moreover, there are some ongoing standardization efforts as well (e.g., [6, 35]).

eBPF [9] eBPF is an umbrella term covering numerous pieces of technology that enable running sandboxed programs inside the Linux kernel (i.e., "kernel programming"), but we are specifically interested in the register-based eBPF bytecode here. eBPF has a number of use cases, such as packet filtering and system monitoring with minimal overhead. Consider the following possible workflow: users can write an I/O tracer

---
*Corresponding author

in C and compile it into eBPF bytecode using *clang*, and the bytecode can then be loaded into the kernel using *libbpf* [14].

While eBPF is primarily used in the context of kernel programming, efforts have been made to enable running eBPF bytecode in a standalone fashion [11, 34, 49, 53]. However, to the best of our knowledge, there has yet been any significant usage of eBPF for general-purpose computations.

WebAssembly [26, 37] WebAssembly is a portable, low-level bytecode format for a stack-based VM. It is designed for secure and efficient execution in web browsers, but recent research has shown that WebAssembly can still use some improvements in terms of security and efficiency [39, 42].

To generate WebAssembly code, tooling exists for several languages (e.g., [17] for Python and [10] for C/C++). To execute WebAssembly code outside of browsers, one can use *wasmtime*, a VM with JIT and embedding support [55], *wasmer* [23], a more customizable option than *wasmtime*, or *GraalVM*,[1] a VM with JIT and language interop support [12]. To interface with certain system calls (e.g., for I/O), WASI may be used [19].

3   Limitations

Before diving into the evaluation, we would like to address some potential issues with our work.

Implementation Dependence Given that both eBPF and WebAssembly are just bytecode formats at the core, any evaluation of them that goes beyond mere theoretical analysis will inevitably be implementation-dependent. However, we believe that such evaluations still have their value.

First, having an implementation-dependent analysis is better than having no analysis at all. Second, from an engineering point of view, "what is achievable today?" is just as important as "what is possible in the future?" If a certain solution is rather lacking at the moment, the question of whether we can justify the costs of improving it naturally arises. Finally, if the analysis happens to reveal some potential improvements for the implementations, then being implementation-dependent actually helps to "kill two birds with one stone".

Single Node We do not concern ourselves with offloading in a distributed setting, as then the focus would lean towards scheduling and coordination, which are out of the scope of this work.

Safety and not Security We will discuss the safety (i.e., protection against human errors) afforded by the offloading mechanisms, but not their security (i.e., protection against attacks), as we are not security experts.

That said, we speculate that securing a WebAssembly runtime is likely easier than securing an eBPF runtime, as the former has some isolation features built in (e.g., linear memory [37]), but the latter relies more heavily on load-time ver-

	eBPF	WebAssembly
safety	?	***
tooling	**	***
compatibility	*	**
portability	**	***
ergonomics	**	**
language-agnostic-ity	*	**

Table 1: Evaluation summaries of the offloading mechanisms

ification [9], which not only is prone to false positives [34] but also has had flaws discovered in the past (e.g. [46, 47]).

Moreover, WebAssembly has already undergone a significant amount of scrutiny regarding its semantics and security [37, 38, 42–44, 56, 57], but the same cannot yet be said of eBPF in the context of general-purpose computations.

4   Qualitative Evaluation

In this section, we will present our findings on the usability (i.e., the user experience) of the two offloading mechanisms. Specifically, we will compare them along 6 axes: safety, tooling, compatibility, portability, ergonomics, and language-agnostic-ity. A summary of the comparison is provided in Table 1 (* = poor, ** = ok, *** = good).

4.1   Safety

As suggested above, we will not cover multi-tenant or public-facing usage here, so we only consider safety in terms of programmer mistakes – "what's the worst that can happen if the offloads misbehave due to bugs in them"?

For WebAssembly, the safety is pretty good, as the offload code will run in an isolated memory space [28, 37] and also has no access to the file system by default [21]. Since these properties are specified at the design level, they are not implementation-dependent. While it is possible for WebAssembly code to hang or waste CPU cycles[2], it is rather unlikely for careless mistakes to wreck havoc outside of the WebAssembly runtime.

For eBPF, the situation is mostly dependent on the verifier. The one used by the Linux kernel is very strict, banning many kinds of loops and memory accesses [15]. However, using that verifier (and kernel-oriented verifiers [34] in general) for data processing is not very practical, and a new verifier implementation is needed [40]. Therefore, we cannot yet say much about the safety of eBPF as an offloading mechanism.

Nonetheless, we suspect that getting eBPF runtime safety right will be a trickier task, as it relies on a specific verifier to catch undesirable behaviors. In comparison, WebAssembly clearly specifies what is allowed at the spec level.

---

[1]As of v21.2, *GraalVM*'s WebAssembly support is still "experimental."

[2]For *wasmtime* [55] hanging can be avoided via the "fuel" feature.

4.2 Tooling

Development Tools For WebAssembly, a suite of tools called *wabt* (The WebAssembly Binary Toolkit) is available for inspecting and transforming WebAssembly binaries [25].

For eBPF, *bpftool* can be used to inspect eBPF programs and maps [33]. It also has some features meant for in-kernel eBPF usage, such as loading programs and updating maps. However, since *bpftool* is inherently tied to in-kernel usage, eBPF programs cannot be inspected unless they are loaded or attached into the kernel. For a standalone option, *llvm-objdump* is a better choice [16].

Virtual Machines As mentioned before, WebAssembly has at least three feature-rich VM implementations: *wasmtime* [55], *wasmer* [23], and *GraalVM* [12]. While they share some common features like JIT support for common architectures, each offers a number of unique features as well. For *wasmtime*, as of v0.29:

- profiling via *perf* and *VTune*
- debugging via *lldb* and *gdb*
- embedding in several host languages
- async host functions
- host functions from closures
- traps and backtraces
- custom memory allocator
- stack and heap limits
- caching of compiled binaries
- controlled code execution via fuel credits
- SIMD and threading

For *wasmer*, as of v2.0:

- embedding in several host languages
- ahead-of-time and cross compilation via LLVM
- multiple ways of storing compiler output
- dynamic and static functions
- custom heap management via `Tunables`
- errors and backtraces
- custom import resolution via `Resolver`
- support for many experimental features like SIMD

For *GraalVM*, as of v21.2:

- interop & embedding for a dozen languages
- access to mature JVM technologies
- creating standalone native binaries ("native images")
- debugging via Chrome DevTools and more
- built-in CPU & memory profilers
- visualizing compilation graphs

eBPF, on the other hand, still has a long way to go in this department. Both *uBPF* [53] and *rBPF* [49] have JIT support (`x86_64` only), but they are still rather lacking when it comes to features like profiling and debugging, which is not surprising, as they are not meant for in-production usage. In fact, the README of *rBPF* clearly states that:

- several features of the kernel eBPF implementation are not yet available (e.g., tail calls)
- it has a very simple verifier, so safety cannot be guaranteed
- the JIT compiler does not emit run-time memory checks, so crashes are possible
- a small number of eBPF instructions are not yet implemented
- *uBPF* is mostly the same, so the above points likely hold for *uBPF* as well

4.3 Compatibility

Compatibility here refers to the users' ability to re-use existing libraries in the ecosystem. Any external dependency, including *libc*, must be re-compiled and statically linked. For WebAssembly, this means that the *wasi-sdk* [22] is likely needed for compilation. For example, to build *zstd* [32]:

```
CXX="wasi-sdk/bin/clang++" \
CC="wasi-sdk/bin/clang" \
AR="wasi-sdk/bin/ar" \
CFLAGS="--sysroot=wasi-sdk/share/wasi-sysroot" \
$(MAKE) -C zstd/lib libzstd.a
```

For eBPF, building *zstd* would fail, as *libc* is not available in the absence of a WASI equivalent:

```
CXX="clang++" CC="clang" CFLAGS="-target bpf -Wall -Werror -O2"
    ↪ make -C zstd/lib libzstd.a

make: Entering directory 'zstd/lib'
CC obj/conf_01069ccb7af67d6f99eda2da05bd0468/static/debug.o
CC obj/conf_01069ccb7af67d6f99eda2da05bd0468/static/entropy_common.
    ↪ o
In file included from common/entropy_common.c:18:
In file included from common/mem.h:24:
In file included from common/zstd_deps.h:27:
In file included from /usr/lib/llvm-12/lib/clang/12.0.0/include/
    ↪ limits.h:21:
/usr/include/limits.h:26:10: fatal error: 'bits/libc-header-start.h
    ↪ ' file not found
#include <bits/libc-header-start.h>
         ^~~~~~~~~~~~~~~~~~~~~~~~~~
1 error generated.
make[1]: *** [Makefile:344: obj/
    ↪ conf_01069ccb7af67d6f99eda2da05bd0468/static/
    ↪ entropy_common.o] Error 1
make: *** [Makefile:240: libzstd.a] Error 2
make: Leaving directory 'zstd/lib'
```

Without access to *libc*, a huge swath of existing libraries would be unusable, so the compatibility of eBPF is quite poor. Additionally, `clang -target bpf`, as of v12.0.1, also does not support several kinds of numeric operations. For example, `i64` divisions:

```
Error: Unsupport signed division for DAG: 0x556059b1c260: i64 =
    ↪ sdiv 0x556059b1ef90, 0x556059b1bf20Please convert to
    ↪ unsigned div/mod.
fatal error: error in backend: Cannot select: 0x556059b1c260: i64 =
    ↪ sdiv 0x556059b1ef90, 0x556059b1bf20
  0x556059b1ef90: i64 = add 0x556059b1ef28, Constant:i64<2>
    0x556059b1ef28: i64 = add 0x556059b1bbe0, 0x556059b1bd18
      0x556059b1bbe0: i64 = mul 0x556059b1bc48, Constant:i64<7>
        0x556059b1bc48: i64 = mul 0x556059b1beb8, 0x556059b1bff0
          0x556059b1beb8: i64,ch = CopyFromReg 0x556059a67458,
              ↪ Register:i64 %3
            0x556059b1bf88: i64 = Register %3
          0x556059b1bff0: i64,ch = CopyFromReg 0x556059a67458,
              ↪ Register:i64 %7
            0x556059b1c0c0: i64 = Register %7
```

```
     0x556059b1b7d0: i64 = Constant<7>
   0x556059b1bd18: i64 = mul nsw 0x556059b1bb10, 0x556059b1ba40
    0x556059b1bb10: i64,ch = CopyFromReg 0x556059a67458,
         ↪ Register:i64 %5
       0x556059b1b700: i64 = Register %5
    0x556059b1ba40: i64,ch = CopyFromReg 0x556059a67458,
         ↪ Register:i64 %8
       0x556059b1bcb0: i64 = Register %8
  0x556059b1b8a0: i64 = Constant<2>
 0x556059b1bf20: i64 = mul nsw 0x556059b1bb10, 0x556059b1be50
   0x556059b1bb10: i64,ch = CopyFromReg 0x556059a67458, Register:
        ↪ i64 %5
     0x556059b1b700: i64 = Register %5
   0x556059b1be50: i64,ch = CopyFromReg 0x556059a67458, Register:
        ↪ i64 %2
     0x556059b1bde8: i64 = Register %2
```

and floating point operations (passing `-fno-builtin` did not help here):

```
4-pi.c:1:8: error: A call to built-in function '__muldf3' is not
   ↪ supported.
double pi_digit(unsigned long n_th) {
     ^
4-pi.c:1:8: error: A call to built-in function '__adddf3' is not
   ↪ supported.
4-pi.c:1:8: error: A call to built-in function '__subdf3' is not
   ↪ supported.
4-pi.c:1:8: error: A call to built-in function '__muldf3' is not
   ↪ supported.
4-pi.c:1:8: error: A call to built-in function '__ltdf2' is not
   ↪ supported.
4-pi.c:1:8: error: A call to built-in function '__adddf3' is not
   ↪ supported.
4-pi.c:1:8: error: A call to built-in function '__adddf3' is not
   ↪ supported.
4-pi.c:1:8: error: A call to built-in function '__muldf3' is not
   ↪ supported.
4-pi.c:1:8: error: A call to built-in function '__muldf3' is not
   ↪ supported.
4-pi.c:1:8: error: A call to built-in function '__adddf3' is not
   ↪ supported.
4-pi.c:1:8: error: A call to built-in function '__muldf3' is not
   ↪ supported.
4-pi.c:1:8: error: A call to built-in function '__muldf3' is not
   ↪ supported.
4-pi.c:1:8: error: A call to built-in function '__adddf3' is not
   ↪ supported.
4-pi.c:1:8: error: A call to built-in function '__adddf3' is not
   ↪ supported.
```

The lack of support for these operations is not due to *clang*. Rather, the issue stems from the fact that eBPF lacks the appropriate instructions for *clang* to emit [13, 15]. As such, we do not see how these operations can be used by eBPF offload programs unless eBPF is given significant extensions.

Of course, the compatibility of WebAssembly is not perfect either – since WASI does not yet provide functions like `fork()` and `pthread_*()` [20], certain existing libraries would not build (e.g., the multi-threaded version of *zstd*), but the situation is arguably better than eBPF.[3]

4.4 Portability

Since eBPF and WebAssembly are bytecodes designed to be executed by virtual machines, they both have decent portability in theory (i.e., users can compile their code once and reuse the built binaries everywhere).

However, in practice, eBPF is at a slight disadvantage on this front because it is not endianness-independent. The LLVM static compiler `llc` 12.0.1 gives:

```
Registered Targets:
  ...
  bpf    - BPF (host endian)
  bpfeb  - BPF (big endian)
  bpfel  - BPF (little endian)
  ...
  wasm32 - WebAssembly 32-bit
  wasm64 - WebAssembly 64-bit
  ...
```

For many use cases, the host endianness is not that relevant, as they do not involve parsing and interpreting data at the byte level. However, this "parsing" task is arguably quite common during data processing (e.g., reading numbers out of a file to perform calculations), so eBPF's endianness-dependence will likely affect the portability of many offload binaries.

For WebAssembly, little-endianness is always used [26], and it is up to the runtime to handle the conversion if the host architecture is big-endian [30], so there is no endianness-related portability concern.[4] Moreover, the difference between `wasm32` and `wasm64` is mostly about the largest memory size supported [24], which can be transparently handled by the runtime and hence should not lead to any difference in program behavior.

4.5 Ergonomics

For this section, we will take a look at the development processes of the two offloading mechanisms and see if they are easy and straightforward.

4.5.1 Compilation

WebAssembly As mentioned before, it is possible to generate WebAssembly code from several source languages. To have a direct comparison with eBPF, we will focus on C here. From C code, there are at least 2 ways to compile to WebAssembly with *libc* support: *Emscripten* [10] and *wasi-sdk* [22]. With *Emscripten*:

```
emcc \
-Wall -O3 -flto \
--no-entry -Wl,--export-all \
-o code.wasm code.c
```

With *wasi-sdk*:

```
wasi-sdk/bin/clang \
--sysroot=wasi-sdk/share/wasi-sysroot \
-Wall -O3 -flto -nostartfiles \
-Wl,--no-entry -Wl,--export-all \
-o code.wasm code.c
```

`-flto` enables link-time optimization (LTO), `-nostartfiles` disables the system *libc*, `-Wl,-no-entry` allows the absence of `main()`, and `-Wl,-export-all` makes sure that all functions, including unused ones, are emitted.

After compilation, *wasm-opt* from the *binaryen* toolchain [8] may be used to further reduce code size:

---

[3]Furthermore, as WebAssembly gains more traction, we expect to see more libraries providing official WebAssembly support. TensorFlow would be one example. For now, the same cannot be said of eBPF.

[4]This should not be a performance concern either, as modern architectures often provide instructions for handling endianness (e.g., `MOVBE` from x86).

```
wasm-opt -Oz code.wasm -o code.wasm.
```

eBPF   For eBPF, both of the two mainstream compilers, *gcc* and *clang*, may be used. We will stick with *clang* here since it is the more popular and mature option: `clang -target bpf -Wall -O2 -o code.o -c code.c`

Compiling to eBPF appears to be much simpler than compiling to WebAssembly. However, if we are actually compiling for in-kernel usage, then some additional `-I` flags are likely needed to access kernel types.

4.5.2   Bytecode Inspection

WebAssembly   A tool called *wasm2wat* is included in *wabt* [25] for the purpose of converting WebAssembly binaries to an official, human-readable text format[5] [31]. Here is an example of this text format:

```
(module
  (func (export "addTwo") (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add))
```

One can also disassemble WebAssembly binaries using *wasm-objdump*:

```
wasm-objdump add_one.wasm

add_one.wasm: file format wasm 0x1

Code Disassembly:

0001bc func[0] <__wasm_call_ctors>:
 0001bd: 01 | nop
 0001be: 0b | end
0001c0 func[1] <malloc>:
 0001c1: 20 00 | local.get 0
 0001c3: 10 02 | call 2
 0001c5: 0b | end
0001c8 func[2]:
 0001c9: 0b 7f | local[0..10] type=i32
 0001cb: 23 00 | global.get 0
 0001cd: 41 10 | i32.const 16
 0001cf: 6b | i32.sub
 0001d0: 22 0a | local.tee 10
 0001d2: 24 00 | global.set 0
 0001d4: 02 40 | block
 0001d6: 41 a0 08 | i32.const 1056
```

eBPF   Although eBPF does not yet have an official textual format,[6] if the eBPF binary of interest was compiled with debugging information, it is possible to view the disassembled binary with interleaved source code:

```
llvm-objdump -S --no-show-raw-insn bpf.o

bpf.o: file format elf64-bpf

Disassembly of section fentry/blk_account_io_start:

0000000000000000 <blk_account_io_start>:
; int BPF_PROG(blk_account_io_start, struct request *req)
      0: r8 = *(u64 *)(r1 + 0)
      1: *(u64 *)(r10 - 8) = r8
      2: r1 = 0
; struct req_info info = { 0 };
      3: *(u64 *)(r10 - 16) = r1
      4: *(u64 *)(r10 - 24) = r1
      5: *(u64 *)(r10 - 32) = r1
```

---

[5]*wasmtime* supports executing this text format directly.

[6]Both *rBPF* and *uBPF* provide (dis)assemblers for a textual assembly format, but it is not an official format.

```
      6: *(u64 *)(r10 - 40) = r1
      7: *(u64 *)(r10 - 48) = r1
      8: *(u64 *)(r10 - 56) = r1
      9: *(u64 *)(r10 - 64) = r1
     10: *(u64 *)(r10 - 72) = r1
     11: *(u64 *)(r10 - 80) = r1
     12: *(u64 *)(r10 - 88) = r1
     13: *(u64 *)(r10 - 96) = r1
     14: *(u64 *)(r10 - 104) = r1
     15: *(u64 *)(r10 - 112) = r1
     16: *(u64 *)(r10 - 120) = r1
     17: *(u64 *)(r10 - 128) = r1
     18: *(u64 *)(r10 - 136) = r1
; u64 t = bpf_ktime_get_ns();
     19: call 5
     20: r6 = r0
```

4.6   Language-agnostic-ity

For WebAssembly, a variety of source languages can be used in theory [26]. However, in reality, the source language has to support exporting `malloc()`, which is needed to allocate WebAssembly buffers and pass data between the host and the WebAssembly VM. That said, this situation will change when the "module linking" feature is stabilized [29], as then it will be possible for the host to link needed functions, and more source languages may be used.

For eBPF, the only supported source language at the moment is C. While it is possible to use other source languages in theory [7], significant efforts are required from the user.

5   Quantitative Evaluation

In this section, we will describe how we evaluated the performance of the two offloading mechanisms. We will also present the evaluation results.

5.1   Setup

The evaluation was carried out on two platforms, an AMD workstation (Ryzen Threadripper Pro 3975WX [3] (x64), 128 GiB of RDIMM DDR4 ECC DRAM (3200 MHz), Linux kernel 5.13.13) and an AWS c6g.xlarge instance (4 vCPU from AWS Graviton2 (arm64), 8 GiB DRAM, Linux kernel 5.11.0) and for the configurations as summarized in Table 2.

VM	*wasmtime* v0.29.0 [55]
	wasmer v2.0.0 [23]
	GraalVM v21.2.0 [12]
	rBPF @ 6c524b3 [49]
	uBPF @ 9eb26b4 [53]
toolchain	*llvm* / *clang* 12.0.1 (WX) & 12.0.0 (AWS)
	wabt 1.0.24 (WX) & 1.0.20 (AWS) [25]
	rustc 1.57.0-nightly (e30b68353 2021-09-05)

Table 2: Evaluation configurations

Since there is currently no *libc* support for eBPF, our evaluation consisted only of simple arithmetic programs that are written in C:

- a baseline dummy program that simply returns 1
- fibonacci calculation
- integer summing
- Erdős prime counting
- multi-factorial calculation

For our evaluation, `clang -Oz` was used to compile to all targets,[7] and *wasm-opt* was used to further optimize WebAssembly binaries after *clang* finishes. Moreover, the following VM configurations were evaluated:

- *graal*: for running WASM binaries on GraalVM via a native image
- *rbpf*: for running patched (see next section) eBPF binaries on rBPF
- *ubpf*: for running patched (see next section) eBPF binaries on uBPF
- *wasmer*: for running WASM binaries on wasmer
- *wasmtime*: for running WASM binaries on wasmtime
- *wasmtime_cached*: like *wasmtime* but with caching enabled

In addition to the WebAssembly VMs and the eBPF VMs, the native binaries were also benchmarked[8] so that they may serve as a baseline. The performance metrics shown below were collected using our own scripts and GNU *time* v1.9 [18] over 10 consecutive runs.

Note that, since *rBPF* and *uBPF* do not currently provide JIT implementations for `arm64`, we did not evaluate them on that architecture. The same applies for *wasmer*, which was not able to compile the WebAssembly code on `arm64` due to LLVM errors. Thus, the tables below will show "NA" for the relevant cells.

5.2 Patching eBPF Binaries

eBPF programs cannot allocate memory (either statically or dynamically) or call non-helper functions (i.e., functions that are not registered with the VM). To make things work,[9] the binaries produced directly by *clang* had to be disassembled and manually patched for our evaluation:[10]

1. in the original code file, add a pointer parameter to all functions that need external memory, which shall be accessed through the said pointer
2. compile the code file and disassemble the binary file produced
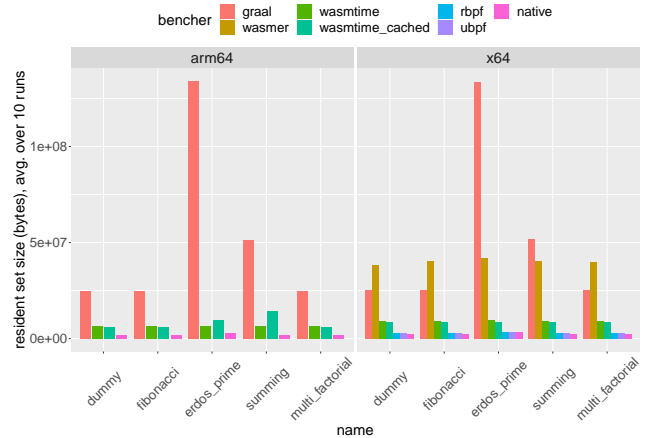3. create a plaintext file

---

Figure 1: 10-run averages of max resident set sizes (bytes) of all (`program`,`VM`) combinations, absolute values

4. put the expected output of the program on the first line; this should be a `u64` integer
5. put the size of the needed static memory on the second line; this should be in unit of bytes and can be `0`
6. paste the disassembled code after the second line, with the entry point pasted first and all other functions pasted afterwards
7. replace all `call` instructions and all `exit` instructions of the callees with `ja` instructions to simulate function calls
8. the address of the external memory will be placed in register `r1`, so make sure you do not mess with `r1` at the beginning
9. make sure the file ends with an `exit` instruction

We will also surmise that it is unlikely for this patching process to be automated post-compilation, as *clang* currently emits all eBPF function calls as `call -1`, making it impossible to stitch assembly blocks together without source information.

5.3 Runtime Memory Footprint

As the environments for computational storage and near-data processing tend to be resource-constrained, the memory footprints of the VMs are naturally of interest here.[11] Thus, we measured the memory usage of the VMs for each of the programs mentioned before.

From Tables 3 and 4 (column 3) and Figures 1 and 2, we can see that on `x64` *rBPF* and *uBPF* were able to achieve near-native performance when it comes to max resident set

---

[7]Of course, there were additional flags involved. Please check the `Makefile`s in the repo.

[8]The native binaries were loaded using `dlopen()` and the functions were invoked via `dlsym()`, so that the native binaries were used in a way that is similar to how eBPF and WebAssembly binaries were used.

[9]In our case, that means to have *some* support for function calls and static memory.

[10]See our repository for more information and examples.

[11]It is worth noting that neither WebAssembly nor eBPF has GC at the moment. For the former, a GC proposal is in early-stage development [27], and the current approach to reclaim resources is to use short-lived, one-off execution contexts, which admittedly are inadequate for long-running programs. For the latter, since in-kernel usage does not support *libc* and therefore memory allocation, how memory management will be handled in a data processing context is yet to be seen.

program	VM	rel. avg. max RSS	rel. avg. latencies	rel. avg. times	rel. avg. times
dummy	*wasmtime*	3.85	92.0	4.37	2.71
	wasmtime_cached	3.52	80.6	4.25	2.59
	graal	10.4	25.5	20.4	2.87
	wasmer	16.0	267.0	4.63	7.31
	rbpf	1.11	1.59	0.0972	0.852
	ubpf	1.10	3.23	0.0887	1.11
fibonacci	*wasmtime*	3.85	94.0	3.77	2.89
	wasmtime_cached	3.52	79.2	4.10	2.62
	graal	10.4	27.9	34.1	3.22
	wasmer	16.8	283.0	3.82	8.00
	rbpf	1.09	2.53	0.185	0.962
	ubpf	1.11	3.83	0.160	1.07
integer summing	*wasmtime*	3.82	79.7	1.99	2.17
	wasmtime_cached	3.48	64.5	1.99	2.10
	graal	21.4	18.7	12.5	10.6
	wasmer	16.6	223.0	1.06	2.35
	rbpf	1.09	2.26	2.22	1.97
	ubpf	1.09	2.97	1.98	1.79
prime counting	*wasmtime*	2.81	77.7	0.969	1.55
	wasmtime_cached	2.48	58.8	0.978	1.43
	graal	39.1	26.1	36.8	27.2
	wasmer	12.3	297.0	0.814	3.34
	rbpf	1.02	4.73	0.907	0.935
	ubpf	1.05	6.71	0.942	0.999
multi-factorial	*wasmtime*	3.80	78.7	3.55	2.58
	wasmtime_cached	3.47	68.6	3.32	2.45
	graal	10.2	22.8	27.1	2.91
	wasmer	16.2	238.0	3.93	7.30
	rbpf	1.08	2.12	0.171	0.997
	ubpf	1.07	3.14	0.140	1.03

Table 3: Evaluation results for all (`program`,`VM`) combinations from the AMD workstation. **Col. 3**: max resident set sizes (bytes). **Col. 4**: startup latencies (ms). **Col. 5**: execution times (ms). **Col. 6**: total running times (ms). All values are 10-run averages and are relative to those of native binaries (= 1).
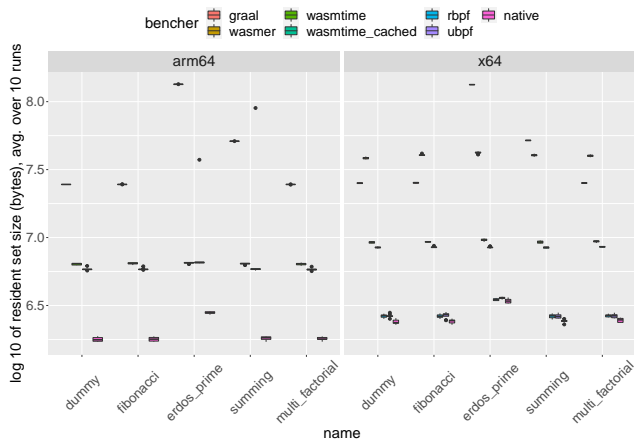

Figure 2: max resident set sizes (bytes) of all (`program`,`VM`) combinations, absolute values in `log10` scale

sizes (RSS). Next comes *wasmtime*, whose max RSS values were 3∼4 times those of native binaries on both architectures.

The much higher max RSS values of *wasmer* on x64 were likely due to its usage of LLVM for ahead-of-time compilation (AOT). As for *GraalVM* native images, we do not have sufficient knowledge of their inner workings to comment, but we speculate that its support for language interop would require a heavier runtime, which might have contributed to the high max RSS values.

From Figure 2, we can see that the max RSS values were fairly consistent across runs, even though a few outliers do exist. However, it is worth noting that max RSS values are *max* values – the VMs most likely do not need this much memory for their entire lifetimes.

5.4 Runtime Startup Latency

Generally, runtime startup latencies do not matter, as they are one-off costs that can be amortized by the execution times

program	VM	rel. avg. max RSS	rel. avg. latencies	rel. avg. times	rel. avg. times
dummy	wasmtime	3.58	48.3	51.6	1.84
	wasmtime_cached	3.28	29.2	63.1	1.65
	graal	13.8	40.4	233.0	3.60
	wasmer	NA	NA	NA	NA
	rbpf	NA	NA	NA	NA
	ubpf	NA	NA	NA	NA
fibonacci	wasmtime	3.61	39.1	27.8	1.80
	wasmtime_cached	3.28	24.9	31.1	1.63
	graal	13.8	34.7	241.0	3.58
	wasmer	NA	NA	NA	NA
	rbpf	NA	NA	NA	NA
	ubpf	NA	NA	NA	NA
integer summing	wasmtime	3.53	39.8	1.24	1.30
	wasmtime_cached	7.84	26.4	1.24	1.31
	graal	28.2	37.1	11.5	10.7
	wasmer	NA	NA	NA	NA
	rbpf	NA	NA	NA	NA
	ubpf	NA	NA	NA	NA
prime counting	wasmtime	2.33	49.2	1.27	1.47
	wasmtime_cached	3.44	23.6	1.27	1.40
	graal	48.0	45.3	48.1	38.1
	wasmer	NA	NA	NA	NA
	rbpf	NA	NA	NA	NA
	ubpf	NA	NA	NA	NA
multi-factorial	wasmtime	3.55	38.1	32.3	1.80
	wasmtime_cached	3.24	23.7	37.2	1.67
	graal	13.7	34.1	243.0	3.57
	wasmer	NA	NA	NA	NA
	rbpf	NA	NA	NA	NA
	ubpf	NA	NA	NA	NA

Table 4: Evaluation results for all (program, VM) combinations from AWS c6g.xlarge. **Col. 3**: max resident set sizes (bytes). **Col. 4**: startup latencies (ms). **Col. 5**: execution times (ms). **Col. 6**: total running times (ms). All values are 10-run averages and are relative to those of native binaries (= 1). **NA** is shown for failed configurations.

of the offload code. However, in certain cases, offload code might have to be repeatedly invoked (e.g., on a series of small files/objects), and significant runtime startup latencies can be quite undesirable or even unacceptable in these situations. Thus, we will also take a look at the startup latencies of the VMs when running our test programs.

To get the startup latencies, we would record t_0, the time when runtime setup starts, and t_1, the time when the test program can be executed, and calculate $t_1 - t_0$. Note that this value also includes the JIT/AOT overhead. Fortunately, this process is not that complicated, as all VMs' APIs clearly differentiate between runtime setup calls and code execution calls.

From Tables 3 and 4 (column 4) and Figures 3 and 4, we can see some quite dramatic variations in the startup latencies of different VMs. As discussed before, *rBPF* and *uBPF* are a bit lacking in terms of their features, which means they are likely more lightweight than the competition, so it is not surprising that they were able to perform well in this category.

On the other hand, wasmer was once again bogged down by the expensive AOT compilation step, making it the worst performer in this category.

That said, if one pays attention to the scale of the y-axis in Figure 3, which shows absolute values, then one would notice that the startup latencies are all quite small – on both architectures, they are mostly under 3 ms. As such, they should not become a performance concern in general.

5.5 Execution Speed

In this section, we present the execution times of the aforementioned programs on different VMs. Here, execution time refers to the amount of time spent on actual computation (i.e., from calling the test program's entry point function to receiving its return value).

The results can be found in column 5 of Tables 3 and 4 and in Figures 5 and 6. For the execution times of the actual computation, the programs can be divided into 2 groups for dis-
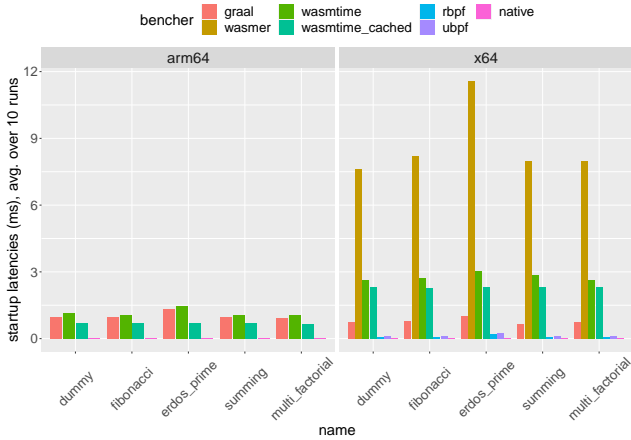
Figure 3: 10-run averages of startup latencies (ms) of all (`program`,`VM`) combinations, absolute values
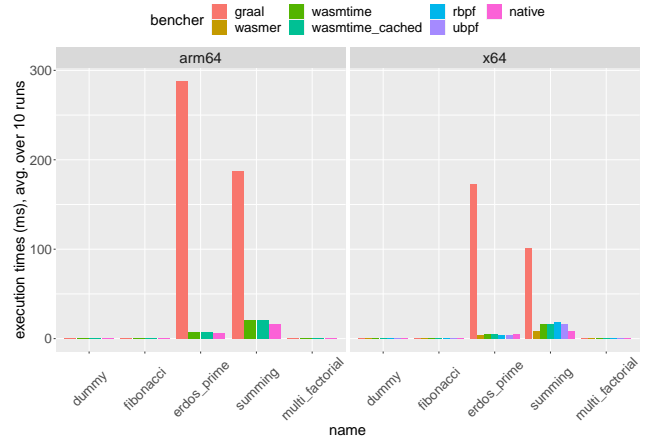

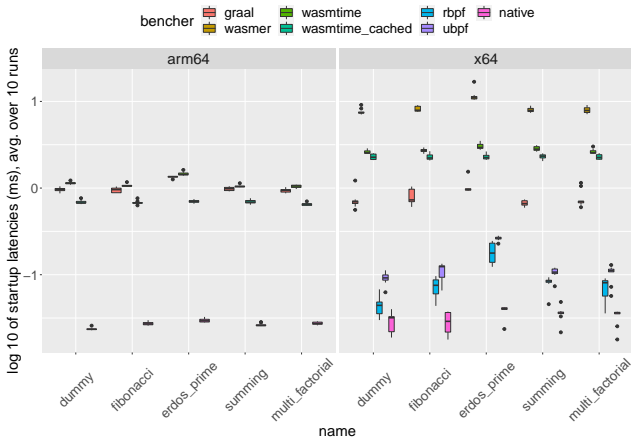Figure 5: 10-run averages of execution times (ms) of all (`program`,`VM`) combinations, absolute values

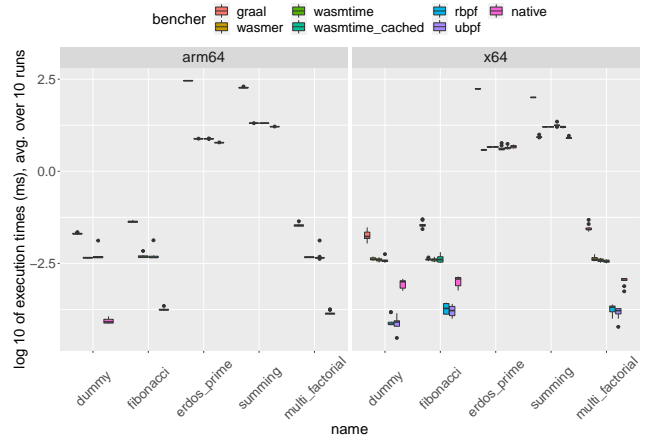Figure 4: startup latencies (ms) of all (`program`,`VM`) combinations, absolute values in `log10` scale

Figure 6: Execution times (ms) of all (`program`,`VM`) combinations, absolute values in `log10` scale

cussion: (*dummy*, *fibonacci*, *multi_factorial*), (*erdos_prime*, and *summing*).

The first group is relatively light on computation – their bars are not even visible in Figure 5. For these programs, *rBPF* and *uBPF* performed so well that, amusingly, they were able to beat the native binaries by some margin. But since these programs finished so quickly, it is not clear if we can draw any meaningful conclusion here.

The second group of programs, due to their significant usage of loops and/or memory accesses, are more intensive and can arguably put the VMs to test a little bit. For these programs, *wasmer*'s high peak memory usage and startup latencies finally paid off, as it ended up getting the best performance for *erdos_prime* and *summing* on x64, excluding the native binaries in the latter case. Otherwise, *wasmtime*'s performance was similar to that of *rBPF* and *uBPF*.

There are also two more points worth mentioning. First, on x64, the native binary for *erdos_prime* performed rather poorly, as it got outperformed by all VMs except *GraalVM*.

Since *erdos_prime* is heavy on memory accesses, one might speculate that the VMs somehow emitted more efficient machine code in this regard. Second, *GraalVM*'s performance was pretty poor across the board, which was, again, likely due to the weight of its language interop infrastructure.

5.6 Total Time

In this section, we present the total running times of the aforementioned programs on different VMs. The total running time of a test program goes from when the VM process is spawned to when the VM process exits. Thus, it can include the time taken by the OS to perform symbol resolution and relocation, spawn the process, etc., thereby also reflecting the weightiness of the VMs a bit.

Given that we collected the data in a Python script, on top of GNU `time`, the exact values might be slightly off, but since we only report the relative values in the tables above, the skew should be consistent and thus irrelevant. The results can be
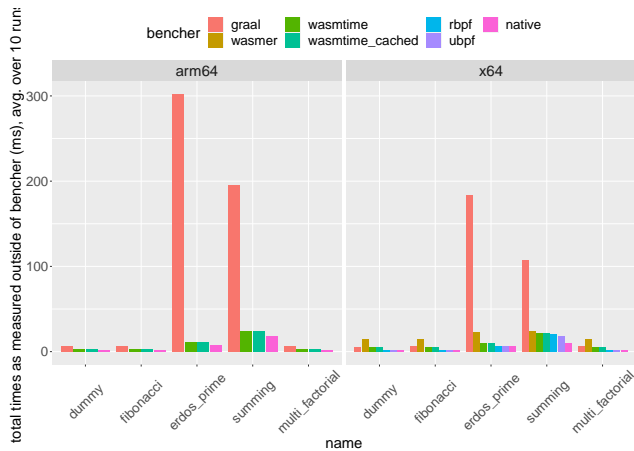
Figure 7: 10-run averages of total running times (ms) of all (`program`,`VM`) combinations, absolute values
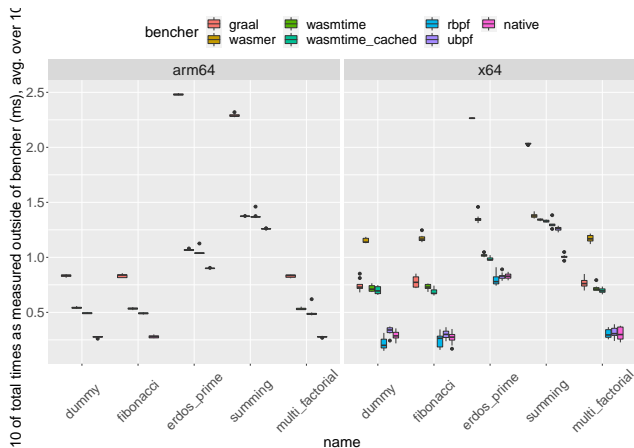

Figure 8: Total running times (ms) of all (`program`,`VM`) combinations, absolute values in `log10` scale

found in column 6 of Tables 3 and 4 and in Figures 7 and 8. Taking everything into account, *rBPF* and *uBPF* come on top once again on x64, often beating the native binaries. After them comes *wasmtime*, whose total times are still close to those of the native binaries, especially on the arm64 AWS instance. Enabling caching for *wasmtime* certainly helps with the total times, but the margin is not dramatic by any means.[12] The AOT cost of *wasmer* cannot yet be fully amortized on small programs like ours, but its numbers for *erdos_prime* and *summing* are not that far behind *wasmtime*'s, so we believe it has great potentials. As for *GraalVM*, once there is substantial operation involved, it is a bit too slow to compete with the other options, so unless language interop or the Java ecosystem is indispensable, we do not see why it should be used for running WebAssembly programs.[13]

---

[12]This will of course change as the code complexity increases.

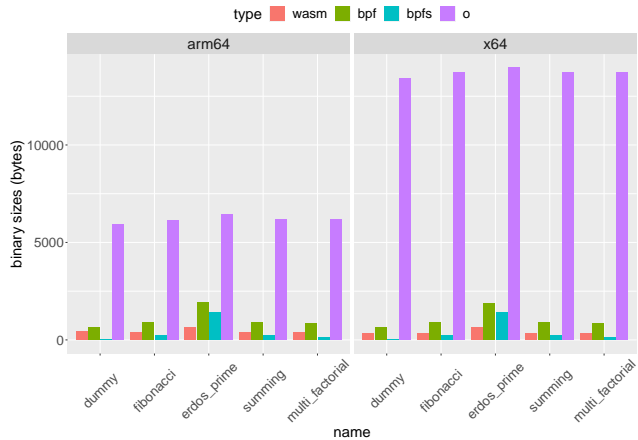[13]But to be fair, WebAssembly is marked as an experimental feature of *GraalVM*.

Figure 9: Binary sizes (bytes)

5.7 Binary Sizes

The sizes of the compiled binaries can be of interest too, as over time, they can have quite some impact on the efficiency of code transmission. The binary sizes of our test programs can be found in Table 5 and Figure 9.

	x64	arm64	eBPF	patched eBPF	wasm
dummy	13440	5936	624	23	318
fibo-nacci	13728	6152	920	238	354
sum-ming	13728	6160	904	212	339
prime	13960	6456	1856	1398	639
multi-fact	13736	6160	856	148	344

Table 5: Binary sizes (bytes)

The native binaries consistently have the largest sizes due to their inclusion of much metadata information (e.g., the `.symtab`, or symbol table, section), whereas the WebAssembly binaries consistently have the smallest sizes, if we ignore "patched eBPF" for a second.

Patched eBPF "binaries" are a bit special. As discussed before, these have to be manually created by a human at the moment, and they are stored as plaintext, so they are not really compiled binaries in some sense. Nevertheless, if we take them into consideration as well, then they are indeed the smallest except for the *erdos_prime* program.

However, since *erdos_prime* also happens to be the most complicated among the programs, we would argue that, as program complexity increases, WebAssembly will likely be the winner in terms of binary sizes.

6 Discussion

While the eBPF VMs had a slight edge in our quantitative evaluation, the amount of restrictions and extra efforts that eBPF

currently imposes on its users suggests that it is not yet ready to support data processing and general-purpose computation. For that, WebAssembly seems to be a more sensible choice, and it is ready for deployment. Below, we will propose some potential improvements for both technologies.

WebAssembly Being a fairly mature technology, WebAssembly and its implementations can still use some improvements in these directions:

- stabilize and finish implementing the SIMD and threading proposals
- provide an interface for (preferably zero-copy) sharing of host data
- address the security concerns pointed out by researchers (e.g. [42])

eBPF Following the evaluation in the previous sections, it is our opinion that eBPF still needs a lot of work before it can become a suitable vehicle for data processing:

- a mature VM, with support for things like debugging and profiling
- an extended instruction set to support all common numeric operations (e.g., signed divisions)
- a specification that clearly defines permitted operations
- a reliable verifier developed in tandem with the specification
- implementation of *libc* and other system services
- undergo thorough security evaluations in the context of data processing

7 Conclusion

In this work, we have evaluated two mechanisms for the purpose of enabling general computation offloading in the context of near-data processing – WebAssembly and eBPF. From our point of view, we believe that WebAssembly is a better choice for the job, but we welcome differing opinions, and we hope more discussions in these directions will ensue.

Acknowledgments

References

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. *ACM SIGOPS Operating Systems Review*, 32(5):81–91, 1998.

[2] Paul Alcorn. Samsung's New HBM2 Memory Has 1.2 TFLOPS of Embedded Processing Power. https://www.tomshardware.com/news/samsung-hbm2-hbm-pim-memory-tflops.

[3] AMD. Ryzen Threadripper PRO 3975WX. https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-pro-3975wx.

[4] Antonio Barbalace and Jaeyoung Do. Computational storage: Where are we today? In *CIDR*, 2021.

[5] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 56–61, 2017.

[6] NVMe Computational Storage Task Group Co-Chairs. Bringing Compute to Storage with NVM Express. https://nvmexpress.org/bringing-compute-to-storage-with-nvm-express/.

[7] Geoffroy Couprie. Poc: compiling to eBPF from Rust. http://unhandledexpression.com/general/rust/2018/02/02/poc-compiling-to-ebpf-from-rust.html.

[8] binaryen developers. binaryen. https://github.com/WebAssembly/binaryen.

[9] eBPF developers. eBPF. https://ebpf.io.

[10] Emscripten developers. Emscripten. https://github.com/emscripten-core/emscripten.

[11] generic-ebpf developers. generic-ebpf. https://github.com/generic-ebpf/generic-ebpf.

[12] GraalVM developers. GraalVM. https://www.graalvm.org.

[13] IO Visor Project developers. Unofficial eBPF spec. https://github.com/iovisor/bpf-docs/blob/master/eBPF.md.

[14] libbpf developers. libbpf. https://github.com/libbpf/libbpf.

[15] Linux developers. Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt.

[16] LLVM developers. llvm-objdump. https://llvm.org/docs/CommandGuide/llvm-objdump.html.

[17] RustPython developers. RustPython. https://github.com/RustPython/RustPython/.

[18] time developers. GNU time. https://www.gnu.org/software/time/.

[19] WASI developers. WASI. https://wasi.dev.

[20] WASI developers. Wasi API Documentation. https://github.com/WebAssembly/WASI/blob/main/phases/snapshot/docs.md.

[21] WASI developers. Wasi Design Principles. https://github.com/WebAssembly/WASI/blob/main/docs/DesignPrinciples.md.

[22] wask-sdk developers. wasi-sdk. https://github.com/WebAssembly/wasi-sdk.

[23] wasmer developers. wasmer. https://github.com/wasmerio/wasmer.

[24] WebAssembly developers. Memory64 Proposal for WebAssembly. https://github.com/WebAssembly/memory64.

[25] WebAssembly developers. Wabt: The WebAssembly Binary Toolkit. https://github.com/webassembly/wabt.

[26] WebAssembly developers. WebAssembly. https://webassembly.org.

[27] WebAssembly developers. Webassembly GC Proposal. https://github.com/WebAssembly/gc.

[28] WebAssembly developers. Webassembly Memories. https://webassembly.github.io/spec/core/syntax/modules.html#syntax-mem.

[29] WebAssembly developers. Webassembly Module Linking Explainer. https://github.com/WebAssembly/module-linking/blob/master/proposals/module-linking/Explainer.md.

[30] WebAssembly developers. Webassembly Portability. https://webassembly.org/docs/portability/.

[31] WebAssembly developers. Webassembly Text Format. https://webassembly.github.io/spec/core/text/index.html.

[32] zstd developers. zstd. https://github.com/facebook/zstd.

[33] eBPF developers. bpftool. https://man.archlinux.org/man/bpftool.8.en.

[34] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhik, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1069–1084, 2019.

[35] SNIA Computational Storage Technical Work Group. Computational Storage Architecture and Programming Model v0.8 rev 0. Technical report, SNIA, 2021. https://www.snia.org/tech_activities/publicreview.

[36] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. Biscuit: A framework for near-data processing of big data workloads. *ACM SIGARCH Computer Architecture News*, 44(3):153–165, 2016.

[37] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[38] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. An empirical study of real-world webassembly binaries: Security, languages, use cases. In *Proceedings of the Web Conference 2021*, pages 2696–2708, 2021.

[39] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not so fast: Analyzing the performance of webassembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, 2019.

[40] Kornilios Kourtis, Animesh Trivedi, and Nikolas Ioannou. Safe and efficient remote application code execution on disaggregated nvm storage with ebpf. *arXiv preprint arXiv:2002.11528*, 2020.

[41] Jeff LeFevre and Carlos Maltzahn. Scaling databases and file apis with programmable ceph object storage. Santa Clara, CA, February 2020. USENIX Association.

[42] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234, 2020.

[43] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. Security chasms of wasm. *NCC Group Whitepaper*, 2018.

[44] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. Swivel: Hardening webassembly against spectre. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[45] Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, Mohammad Dashti, Romaric Jodin, Alexandre Ghiti, Jordi Chauzi, et al. A case study of processing-in-memory in off-the-shelf systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 117–130, 2021.

[46] NIST NVD. Cve-2020-8835. https://nvd.nist.gov/vuln/detail/CVE-2020-8835.

[47] NIST NVD. Cve-2021-20268. https://nvd.nist.gov/vuln/detail/CVE-2021-20268.

[48] Ivan Luiz Picoli. *OX: Deconstructing the FTL for Computational Storage*. IT University of Copenhagen, Computer Science, 2019.

[49] rBPF developers. rbpf. https://github.com/qmonnet/rbpf.

[50] Samsung. Samsung SmartSSD. https://samsungsemiconductor-us.com/smartssd/.

[51] SNIAOnStorage. What is eBPF, and Why Does it Matter for Computational Storage? https://sniacmsiblog.org/2021/07/what-is-ebpf-and-why-does-it-matter-for-computational-storage/.

[52] NGD System. NGD Newport NVMe Computational Storage Platform. https://www.ngdsystems.com/solutions.

[53] uBPF developers. ubpf. https://github.com/iovisor/ubpf.

[54] Tobias Vinçon, Arthur Bernhardt, Ilia Petrov, and Andreas Koch. nkv in action: accelerating kv-stores on native computational storage with near-data processing. *Proceedings of the VLDB Endowment*, 13(12):2981–2984, 2020.

[55] wasmtime developers. wasmtime. https://github.com/bytecodealliance/wasmtime.

[56] Conrad Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on certified programs and proofs*, pages 53–65, 2018.

[57] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. Weakening webassembly. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28, 2019.

[58] Tong Zhang. The true value of storage drives with built-in transparent compression. SDC2020, October 2020. SNIA.