FOSDEM'24

elfconv: AOT compiler that translates Linux/AArch64 ELF binary to LLVM bitcode targeting WebAssembly

repo: https://github.com/yomaytk/elfconv

2024/02/04

Masashi Yoshimura, NTT

What is WebAssembly? Why using that?

- **WebAssembly** (**WASM**) is virtual machine instruction set
- ✅ portable
  - enables to run apps on **both browsers and servers** without modification
- ✅ secure
  - **highly isolated from the host kernel** on the server by **WASI**.
    - WASI is an API that provides access to several OS-like features (filesystems, sockets, …).
    - WASI is implemented by WASI runtimes (wasmtime, WasmEdge, …).
  - memory isolation with harvard architecture
    - architecture that physically separates memory for instructions and data.

What is WebAssembly? Why using that?

- ❌ limitation in the capability of apps
  - can jump to only the instructions that are determinable at compile time
    - cannot indirectly jump to the instructions generated in the data memory at runtime
  - WASI implementation doesn't cover all POSIX APIs (e.g. fork, exec)

challenging in building WASM

Many programming languages support WASM (e.g. C, C++, Rust, Go, …).
However, it isn't easy to build WASM in some cases as follows.

Case 1. The programming language that you want to use doesn't completely support WASM

Case 2. binaries are available, but the source codes of the binaries are not available

- e.g.) The source code is not available under lisence

Case 3. Time-consuming to building the environment

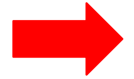
- e.g.) you might be not able to build the dependent libraries because they are not maintained and so on.

Existing projects that run Linux binaries on WASM

NTT

- **TinyEMU**: https://bellard.org/tinyemu/
  - Author: Fabrice Bellard
  - x86 and RISC-V emulator available on the browser
  - Linux kernel can run on the browser
- **container2wasm**: https://github.com/ktock/container2wasm
  - Author: Kohei Tokunaga, NTT
  - enables to run Linux kernel and container runtimes with emulators compiled to WASM (e.g. TinyEMU)
  - can run containers without modification on the browser and WASI runtimes

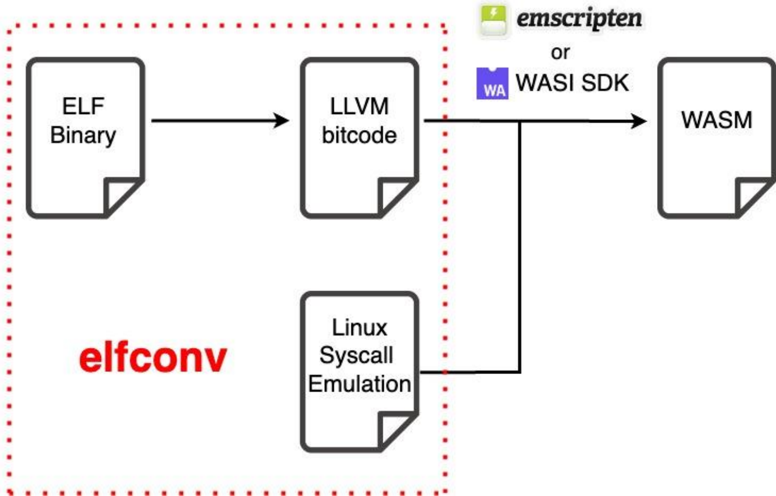
But, emulators possibly incur large performance overheads…

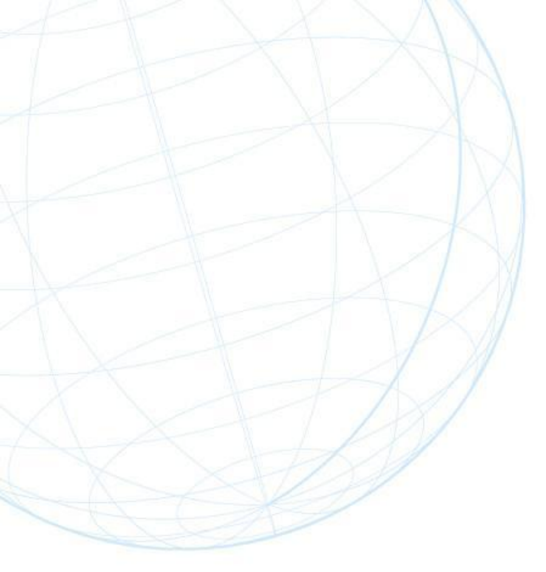


AOT compile Linux binaries to WASM!

elfconv: AOT compiler from Linux/ELF to WASM

- compiles Linux ELF binary to LLVM bitcode
- existing compilers (e.g. emscripten) compile LLVM bitcode and the object of Linux syscalls emulation to WASM
- elfconv is successor to **myAOT:** https://github.com/AkihiroSuda/myaot
  - Author: Akihiro Suda, NTT
  - An experimental AOT-ish compiler (Linux/riscv32 ELF → Linux/x86_64 ELF, Mach-O, WASM, ...)





Demo

How it works? (ELF -> LLVM bitcode)

- elfconv-Lifter
  - parse ELF binary, map every ELF section, etc…
- remill (elfconv-Backend) : https://github.com/lifting-bits/remill
  - library for lifting machine code to LLVM bitcode

ELF

.seciton .data
.section .bss
...

_start:
nop
mov x29, 0
...

_fun1:
mov x2, x0
ldr x0, [x1, 8]
...

.....

Lifting Module

elfconv-Lifter

elfconv-Backend

REMILL

LLVM bitcode

ELF Sections Memory
(.data, .bss, ...)

define ptr @_start_lift (...) {
1_nop:
...
}

define ptr @_fun1_lift (...) {
1_mov:
...
}

.....

How it works? (remill)

- convert a function to a LLVM IR function (e.g. _func1 -> @_func1_lift)
  - But, need to extract every function from ELF

machine code

```
_func1:
        mov x2, x0
        ldr x0, [x1, 8]
        add x2, x2, 1
        ...
        ret
```

REMILL

LLVM IR

```
define ptr @_func1_lift (...) {
1_mov:
        ...
2_ldr:
        ...
3_add:
        ...
...
n_ret:
        ...
}
```

How it works? (remill)

- convert a CPU instruction to a LLVM IR block (e.g. mov x2, x0 -> 1_mov)

machine code

```
_func1:
        mov x2, x0
        ldr x0, [x1, 8]
        add x2, x2, 1
        ...
        ret
```

REMILL →

LLVM IR

```
define ptr @_func1_lift (...) {
1_mov:
        ...
2_ldr:
        ...
3_add:
        ...
...
n_ret:
        ...
}
```

How it works? (remill)

- convert a CPU instruction to a LLVM IR block
  - PC calculation, Operand calculation
  - call the function of the instruction-specific operation

machine code

ldr x7, [x3]

REMILL

LLVM IR

```
28:
  %29 = load i64, ptr %NEXT_PC, align 8
  store i64 %29, ptr %PC, align 8
  %30 = add i64 %29, 4
  store i64 %30, ptr %NEXT_PC, align 8
  %X75 = ptrtoint ptr %X7 to i64
  %31 = load i64, ptr %X3, align 8
  %32 = load ptr, ptr %MEMORY, align 8
  %33 = call ptr @_ZN12_GLOBAL_LoadI3(ptr %32, ptr %state, i64 %X75, i64 %31)
  br label %34
```

PC calculation

Operand calculation

call the function of instruction-specific operation

How it works? (indirect jump)

- The code of WASM can indirectly jump to only the code that is determinable at compile time.
- currently, not support setjmp and longjmp.

machine code

```
_fun1:
  ...
  mov x8, x9
  ...
  br x7
```

indirect jump

REMILL

LLVM IR (psuedo)

```
define ptr @_fun1_lift() {
%IDR = alloca i64
...
7_br
    store x7 %IDR
    br %L_idr
...
L_idr:
    %BAR = call get_label(%IDR)
    indirectbr %BAR [%1_mov, %2_add, ...]
}
```

1. store the address to %IDR and jump to %L_idr

2. get the target label (%BAR) and jump with `indirectbr`

all labels of this function (@_func_lift)

How it works? (LLVM bitcode -> WASM)

- statically link LLVM bitcode and elfconv-Runtime
- elfconv-Runtime
  - mapped memory (stack, heap), Linux system calls emulation

How it works? (Linux syscalls emulation)

- libc implementation: emscripten, wasi-libc, etc...

  Case 1. use libc function if it exists (e.g. write)

```
case AARCH64_SYS_WRITE: /* write (unsigned int fd, const char *buf, size_t count) */
state_gpr.x0.qword = write(state_gpr.x0.dword,
                           _ecv_translate_ptr(state_gpr.x1.qword),
                           static_cast<size_t>(state_gpr.x2.qword));
break;
```

How it works? (Linux syscalls emulation)

- libc implementation: emscripten, wasi-libc, etc...

  Case 2. pseudo-implement the syscall if it doesn't exist (e.g. brk)

```
case AARCH64_SYS_BRK: /* brk (unsigned long brk) */
{
      auto heap_memory = g_run_mgr->mapped_memorys[1];
  if (state_gpr.x0.qword == 0) {
      /* init program break (FIXME) */
      state_gpr.x0.qword = heap_memory->heap_cur;
  } else if (heap_memory->vma <= state_gpr.x0.qword &&
      state_gpr.x0.qword < heap_memory->vma + heap_memory->len) {
      /* change program break */
      heap_memory->heap_cur = state_gpr.x0.qword;
  } else {
      elfconv_runtime_error("Unsupported brk(0x%016llx).\n", state_gpr.x0.qword);
  }
} break;
```

not use brk (unsigned long brk)

Performance

- target sample ELF binary: prime number calculator
  - compute all prime numbers less than the input integer
- Test: ELF/aarch64 -> LLVM bitcode -> ELF/x86_64 (not WASM)
  - current system calls emulation for WASI runtimes is insufficient, so we use x86_64 as the output binary for benchmark tests.
- comparison : QEMU emulation aarch64 to x86_64

QEMU emulation vs. binary AOT compilation

Performance

Case 1. input integer : 10,000,000

- QEMU : 9.437s
- elfconv : 8.353s

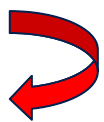
Case 2. input integer : 50,000,000

- QEMU : 1m30.014s
- elfconv : 1m18.972s

Performance

Case 1. input integer : 10,000,000

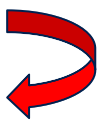
- QEMU : 9.437s
- elfconv : 8.353s



1.13 times faster

Case 2. input integer : 50,000,000

- QEMU : 1m30.014s
- elfconv : 1m18.972s



1.14 times faster

Future works

- output other binary formats
  - support WASM, ELF/x86-64 now

WA

support now

LLVM bitcode

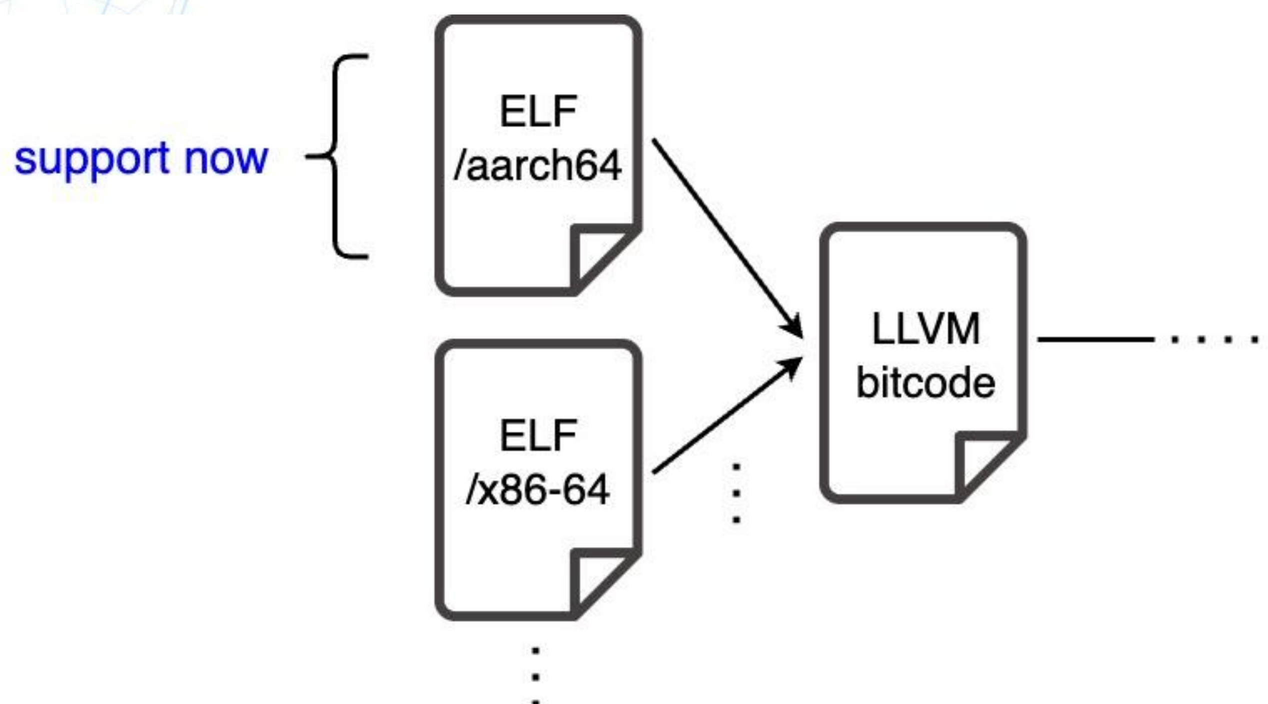
ELF /x86-64

Other Binary

Future works

NTT

- compile ELF of other CPU architectures
  - support aarch64 now



Future works

- append system calls emulation
  - implement a part of system calls now
  - Some system calls (e.g. fork, exec) are difficult to implement when targeting WASM
- support dynamic linking
  - support only static linking now
- performance analysis of WASM target
- make LLVM bitcode more efficient

repo: https://github.com/yomaytk/elfconv

Questions?, and I would like to get your opinions!