LINUX PLUMBERS CONFERENCE September 20-24, 2021

Update on the BPF support in the GNU Toolchain

Jose E. Marchesi <jose.marchesi@oracle.com>
David Faust <david.faust@oracle.com>
Guillermo E. Martinez <guillermo.e.martinez@oracle.com>

Summary

- The port: bpf-unknown-none
- Miscellaneaus updates
- BTF support
- BPF CO-RE support
- New options for BPF features and ISA selection
- Support for BPF atomics

bpf-unknown-none

- Binutils port
- GCC backend
- GDB port
- Simulator
- Dejagnu board

Miscellaneous updates

- New xBPF instructions: SDIV and SMOD
- The GCC BPF backend can now emit DWARF.
- Adjustments to pacify the kernel verifier.
- Many bug fixes.
- We are finally on par with LLVM!

BTF support in GCC

- BPF Type Format: debug info format for BPF programs
- Now fully implemented in GCC
- Supported for any target via `-gbtf`
  - Could be used to e.g. generate BTF for kernel directly
- Less general purpose than CTF
  - Tailored to BPF programs

Problem: BPF Portability

- BPF programs generally include kernel headers
  - Definitions of data structures
  - Not limited to userspace API, may be internal
- Two problems:
  - Internal kernel data structures regularly change
  - Might include other, target-specific headers
- Not portable between kernel versions!

Solution: BPF CO-RE

- Compile Once – Run Everywhere
- Designed by kernel and LLVM BPF folks
- Emit relocations annotating accesses to data structures:
  - Which instructions need adjusting
  - How to adjust according to structure definitions of local kernel

```
struct S *foo = ...;
char x = foo->data[3].c

/* do stuff with x */
```

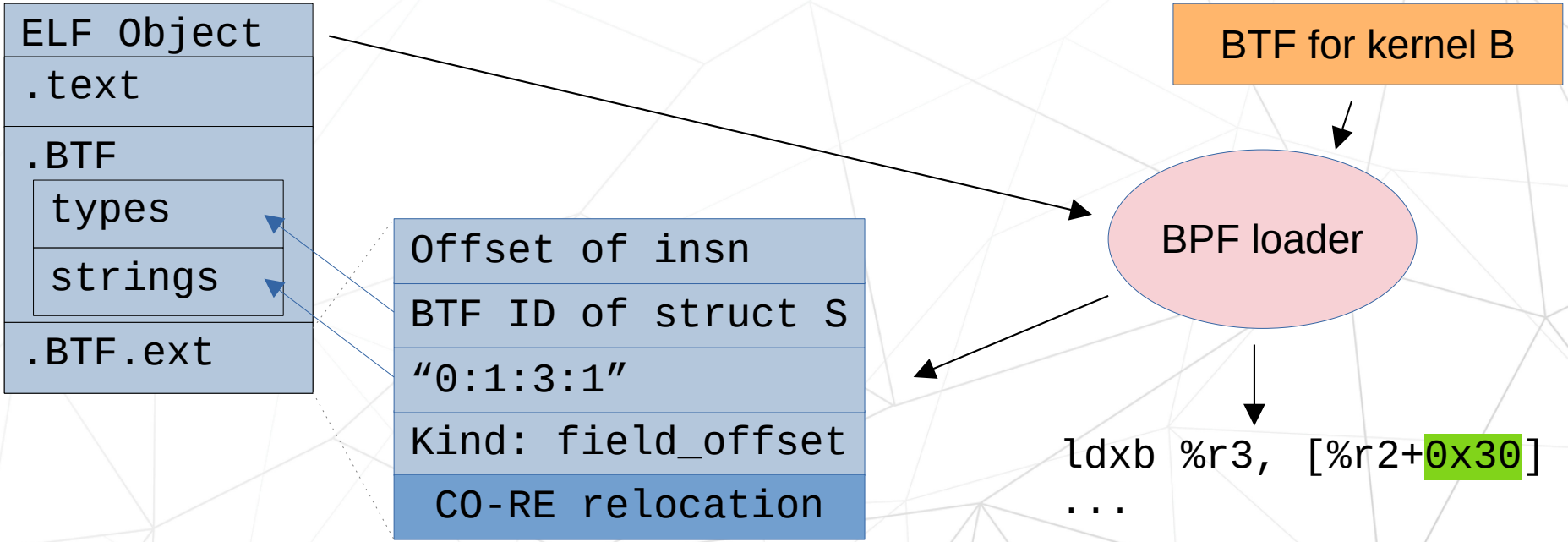
```
ldxb %r3, [%r2+0x20]
...
```

```
struct S {
  int a;
  struct {
    int b;
    char c;
  } data[4];
};
```

A

```
struct S {
  int a;
  struct {
    int new;
    int b;
    char c;
  } data[4];
};
```

B



BPF CO-RE in GCC

- Needs type information of records (BTF)
- … also some unfortunate coupling with .BTF
- Implemented in the BPF backend
- CO-RE relocs emitted in .BTF.ext
- `-mco-re`

BPF CO-RE in GCC

Builtin: marker for accesses to record

```
#define _(x) __builtin_preserve_access_index (x)

char x = _(foo->data[3].x);
```

- Transparent: does not change semantics of program
- Overloaded: return type is type of expression
- Accepts statement-expressions...

```
_(({ int x  = foo->a;
     foo->b = x + foo->c;}));
```

BPF CO-RE in GCC

Attribute: applies to structs/unions

```
struct S {
   int a;
   char c;
   struct T tea;
} __attribute__((preserve_access_index));
```

Any access to an attributed type will produce a CO-RE relocation.

GCC - BPF feature selection

- BPF has gained new instructions over time
  - Extra compare-and-branch operations (<, ≤)
  - 32-bit ALU and 32-bit jump operations
  - New atomic operations
- New options in GCC to enable/disable generation of these instructions
  - `-mjmpext`, `-mjmp32`, `-malu32`
- Use to match features implemented by target kernel

GCC - BPF ISA version

- `-mcpu={v1, v2, v3, latest}`
  - V1: base eBPF with no extensions
  - V2: adds extra jump operations
  - V3: adds 32-bit ALU/jump operations
  - Default: `latest`, synonym for `v3`
- May combine with feature selectors
  - e.g. `-mcpu=v3 -mno-jmp32`

BPF Atomics

Contents

- Concurrency in **eBPF**.
  - Linux Kernel: Atomics for **eBPF**.
  - Changes in **CGEN** Framework.
  - **binutils-gdb**: Atomic **eBPF** instructions.
  - **GCC**: Atomic built-in functions.

Concurrency in **eBPF**.

- More than one process/threads can access the **eBPF** store data to be processes by an **eBPF** program (map).
- Synchronization mechanisms:
  - Spinlocks (https://lwn.net/Articles/779120/)
    - bpf_spin_{lock,unlock}
    - Reentrancy Pattern: Fine granularity!!
  - Atomics instructions: (https://lwn.net/Articles/838884/)

Linux Kernel: Atomics for **eBPF** 1/2

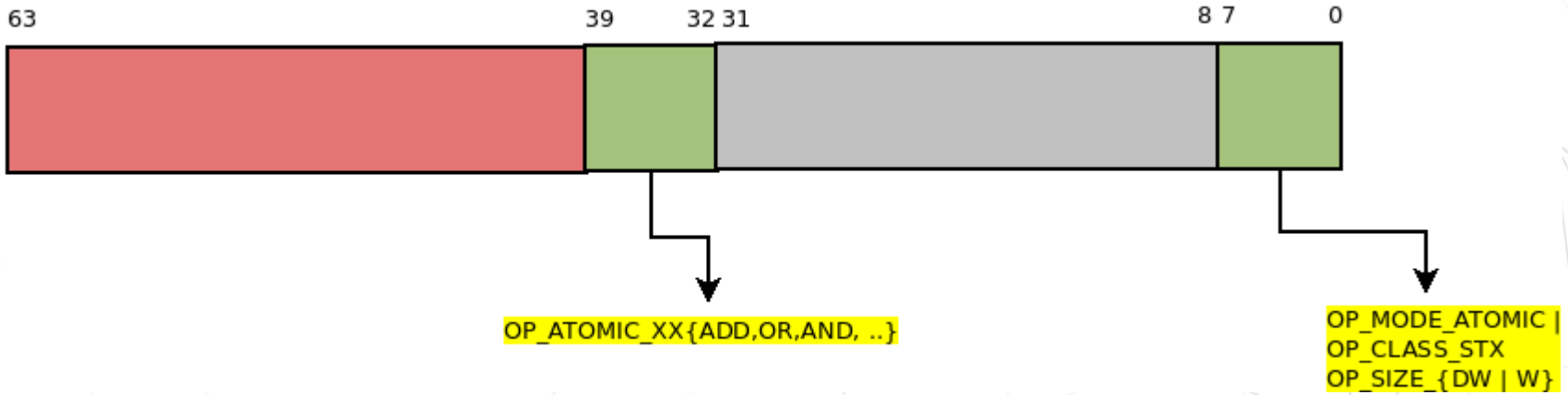
- Linux Kernel 2020.
  - **https://www.kernel.org/doc/Documentation/networking/filter.rst**

- **eBPF** Atomic operations:

BPF_{ADD, AND, OR, XOR, XCHG, CMPXCHG}

"*use the **immediate field** of the existing STX XADD instruction to encode the operation. This works nicely, **without breaking existing programs**, because the immediate field is currently reserved-must-be-zero, and extra-nicely because BPF_ADD happens to be zero*"

Linux Kernel: Atomics for **eBPF** 2/2



Is this an issue in the original eBPF ISA design?

*So we are expecting an **OPERATOR**!! (in the instruction's **immediate field**), but now we have the "extra encoding" i.e an **OPCODE**!! it represent what kind of atomic operation the VM will be execute.!!*

Changes in **CGEN** Framework 1/2

- Opcodes generation: **C structures** for a particular target.
  - opcodes/bpf-opc.c
  - opcodes/bpf-desc.c
  - opcodes/bpf-desc.h
  - ..
- Make sure that **CGEN** is able to compute right structures for the "**eBPF** atomics encoding":

```
(define-normal-insn-enum insn-atomic-op-le "eBPF atomic insn"
    ((ISA ebpfle xbpfle)) OP_ATOMIC_LE_ f-op-atomic
      ((ADD #x01)
       (OR  #x41)
         ...
       (CMP #xf1)))

(define-normal-insn-enum insn-atomic-op-be "eBPF atomic insn"
    ((ISA ebpfbe xbpfbe)) OP_ATOMIC_BE_ f-op-atomic
      ((ADD #x01000000)
       (OR  #x41000000)
        ….
       (CMP #xf1000000)))
```

Changes in **CGEN** Framework *2/2*

- Provides support to compute the mask and get values *for constant fields* where we placed the opcode:

```
mask:  0xffffffff000000ff
value: 0x000000e1000000db ;; xchgdw in L.E
```

- Fix a wrong calculation in *mask and value* when a field is defined in a long form with *an offset* is different to 0:

```
bitrange: #(object <bitrange> 29 32 31 32 64 #t)
mask: 0xffffffff
value: simplify.inc:173:3: Error: Instruction has opcode
       bits outside of its mask.

Fixed to:

bitrange: #(object <bitrange> 29 32 31 32 64 #t)
mask: 0xffffffff00000000
value:      0xaa00000000
```

https://sourceware.org/pipermail/cgen/2021q3/002774.html

binutils-gdb: Atomic **eBPF** instructions 1/3

- Add atomic instructions: (***dni***) in *CPU Description Language:* ***bpf.cpu***

```
(dais add (.if (.eq x-endian le) OP_ATOMIC_LE_ADD OP_ATOMIC_BE_ADD) ...)
...
(dais cmp (.if (.eq x-endian le) OP_ATOMIC_LE_CMP OP_ATOMIC_BE_CMP) ...)
```

- Add atomic instructions to simulator.

- Carefully internal revision and exhaustive regression test.
  - *arm-linuxeabi, aarch64-linux, alpha-dec-vms, am33_2.0-linux, arc-linux-uclibc, avr-elf, cr16-elf, cris-elf, crisv32-linux, crx-elf, d10v-elf, d30v-elf, dlx-elf, epiphany-elf, fr30-elf, x86_64-linux, x86_64-w64-mingw32, x86_64-nacl, xgate-elf, xstormy16-elf, xtensa-elf, z8k-coff, z80-coff, ...*

https://sourceware.org/pipermail/binutils/2021-August/117798.html

binutils-gdb: Atomic eBPF instructions 2/3

Linux Kernel	Mnemonic	eBPF assembly Statements	eBPF decode Instructions
BPF_ADD	xadd{dw,w}	xadddw [%r1+0x1eef],%r2	db 12 1e ef 00 00 00 01
BPF_AND	xand{dw,w}	xanddw [%r1+0x1eef],%r2	db 12 1e ef 00 00 00 51
BPF_OR	xor{dw,w}	xordw [%r1+0x1eef],%r2	db 12 1e ef 00 00 00 41
BPF_XOR	xxor{dw,w}	xxorw [%r1+0x1eef],%r2	c3 12 1e ef 00 00 00 a1
BPF_XCHG	xchg{dw,w}	xchgdw [%r1+0x1eef],%r2	db 12 1e ef 00 00 00 e1
BPF_CMPXCHG	xcmp{dw,w}	xcmpdw [%r1+0x1eef],%r2	db 12 1e ef 00 00 00 f1

Table 1. eBPF Atomic instructions in B.E.

Linux Kernel	Mnemonic	eBPF assembly Statements	eBPF decode Instructions
BPF_ADD	xadd{dw,w}	xadddw [%r1+0x1eef],%r2	db 12 1e ef 01 00 00 00
BPF_AND	xand{dw,w}	xanddw [%r1+0x1eef],%r2	db 12 1e ef 51 00 00 00
BPF_OR	xor{dw,w}	xordw [%r1+0x1eef],%r2	db 12 1e ef 41 00 00 00
BPF_XOR	xxor{dw,w}	xxorw [%r1+0x1eef],%r2	c3 12 1e ef a1 00 00 00
BPF_XCHG	xchg{dw,w}	xchgdw [%r1+0x1eef],%r2	db 12 1e ef e1 00 00 00
BPF_CMPXCHG	xcmp{dw,w}	xcmpdw [%r1+0x1eef],%r2	db 12 1e ef f1 00 00 00

Table 2. eBPF Atomic instructions in L.E.

binutils-gdb: Atomic **eBPF** instructions 3/3

```
$ bpf-as --EB  gas/testsuite/gas/bpf/atomic.s
```

```
$ bpf-objdump -dr a.out
```

```
a.out:     file format elf64-bpfbe

Disassembly of section .text:

0000000000000000 <.text>:
   0:   db 12 1e ef 00 00 00 01         xadddw [%r1+0x1eef],%r2
   8:   c3 12 1e ef 00 00 00 01         xaddw [%r1+0x1eef],%r2
  10:   db 12 1e ef 00 00 00 41         xordw [%r1+0x1eef],%r2
  18:   c3 12 1e ef 00 00 00 41         xorw [%r1+0x1eef],%r2
  20:   db 12 1e ef 00 00 00 51         xanddw [%r1+0x1eef],%r2
  28:   c3 12 1e ef 00 00 00 51         xandw [%r1+0x1eef],%r2
  30:   db 12 1e ef 00 00 00 a1         xxordw [%r1+0x1eef],%r2
  38:   c3 12 1e ef 00 00 00 a1         xxorw [%r1+0x1eef],%r2
  40:   db 12 1e ef 00 00 00 e1         xchgdw [%r1+0x1eef],%r2
  48:   c3 12 1e ef 00 00 00 e1         xchgw [%r1+0x1eef],%r2
  50:   db 12 1e ef 00 00 00 f1         xcmpdw [%r1+0x1eef],%r2
  58:   c3 12 1e ef 00 00 00 f1         xcmpw [%r1+0x1eef],%r2
```

GCC: Atomic built-in functions 1/4

- Fully compliant with:
  - **https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html**
  - **https://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html**
- Support to:
  - `__atomic_fetch_{add, sub, and, xor, or}`
  - `__atomic_exchange`
  - `__atomic_compare_exchange_n`
  - `__atomic_<operation>_fetch`
  - `__sync_fetch_and_<operation>`
  - `__sync_<operation>_and_fetch`

GCC: Atomic built-in functions 2/4

- Defining atomics instructions in *bpf* machine description file:
  - **gcc/config/bpf/bpf.md:**

```
(define_insn "atomic_fetch_add<AMO:mode>"
  [(set (match_operand:AMO 0 "register_operand" "=r")
  (unspec_volatile:AMO
    [(match_operand:AMO 1 "memory_operand" "+o")
     (match_operand:AMO 2 "nonmemory_operand" "0")
     (match_operand:AMO 3 "const_int_operand")] ;; Memory model
    UNSPEC_XADD))]
  ""
  "xadd<mop>\t%1,%0")
```

GCC: Atomic built-in functions 3/4

- Add constrains to place operands in specific register:

```
(define_register_constraint "t" "R0"
   "Register r0")
...
(define_insn "atomic_compare_and_swap<AMO:mode>_1"
  [(set (match_operand:AMO 0 "register_operand" "=t") ;; must be r0
  (unspec_volatile:AMO
    [(match_operand:AMO 1 "memory_operand" "+o")   ;; memory
     (match_operand:AMO 2 "register_operand" "0") ;; expected
     (match_operand:AMO 3 "register_operand" "r") ;; desired
     (match_operand:SI 4 "const_int_operand")]    ;; success
(unused)
    UNSPEC_CMPXCHG))]
  ""
  "xcmp<mop>\t%1,%3")
```

GCC: Atomic built-in functions 4/4

```
$ bpf-gcc -O2 /home/byby/oracle/src/gcc/gcc/testsuite/gcc.target/bpf/atomic-compare-exchange.c
```

```
$ bpf-objdump -dr a.out
```

```
a.out:     file format elf64-bpfle

Disassembly of section .text:

0000000000400000 <foo>:
  400000:       18 02 00 00 f8 11 80 00         lddw %r2,0x8011f8
  400008:       00 00 00 00 00 00 00 00
  400010:       18 03 00 00 d8 11 80 00         lddw %r3,0x8011d8
  400018:       00 00 00 00 00 00 00 00
  400020:       18 01 00 00 e8 11 80 00         lddw %r1,0x8011e8
  400028:       00 00 00 00 00 00 00 00
  400030:       79 15 00 00 00 00 00 00         ldxdw %r5,[%r1+0]
  400038:       79 34 00 00 00 00 00 00         ldxdw %r4,[%r3+0]
  400040:       bf 50 00 00 00 00 00 00         mov %r0,%r5
  400048:       db 42 00 00 f1 00 00 00         xcmpdw [%r2+0],%r4
  400050:       b7 06 00 00 01 00 00 00         mov %r6,1
  400058:       1d 50 01 00 00 00 00 00         jeq %r0,%r5,1
  400060:       b7 06 00 00 00 00 00 00         mov %r6,0
  400068:       57 06 00 00 01 00 00 00         and %r6,1
  400070:       55 06 01 00 00 00 00 00         jne %r6,0,1
  400078:       7b 01 00 00 00 00 00 00         stxdw [%r1+0],%r0
  400080:       79 35 00 00 00 00 00 00         ldxdw %r5,[%r3+0]
  400088:       61 14 00 00 00 00 00 00         ldxw %r4,[%r1+0]
  400090:       bf 40 00 00 00 00 00 00         mov %r0,%r4
  400098:       c3 52 00 00 f1 00 00 00         xcmpw [%r2+0],%r5
  ....
```

```
printf ("Thanks a lot for coming!!");
```