Enabling AArch64 Instrumentation Support in BOLT

Elvina Yakubova <elvina.yakubova@huawei.com>

Advanced Software Technology Lab, Huawei

Special thanks to

Alexey Moksyakov
Vasily Leonenko
Vladislav Khmelevsky

Contents

Intro

- BOLT is a post-link optimizer developed to speed up large applications. It became part of LLVM since 11 Jan 2022 and was added to 14th release.
- Sample-based profiling:
- Profile is gathered by Linux Perf tool
- Sampling using hardware profile
  - ✓ doesn't require a special build of the application
  - ✓ profile collection overheads are negligible
- Instrumentation:
- In case if necessary advanced hardware counters are not available
- In case if perf record is not available on target environment

Problem Statement

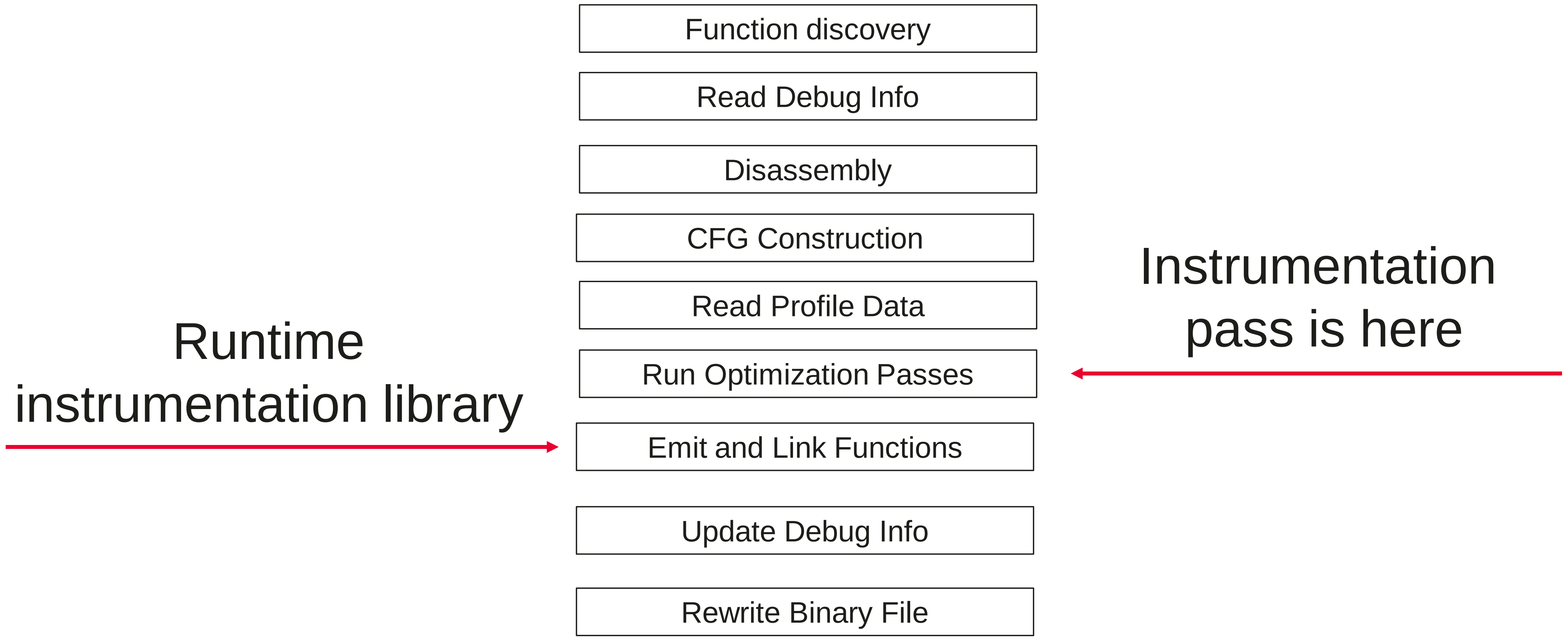
The more accurate profile you get → the better the optimization effect will be.

- For X86_64 both perf and instrumentation options are available. LBR is highly recommended to use.
- For AArch64, LBR feature is missing and it is not always possible to use its analogue, so perf profile is not precise enough. Instrumentation was absent.

What was missing?

- Implementation of AArch64 specific MCInst instructions in AArch64MCPlusBuilder
- AArch64-specific functionality in runtime library
  - Part of runtime library is written on asm directly and contained only X86 specific syscalls

Architecture



Instrumentation Pass

- Modifies branches and calls to increment counters, which are declared in the newly created section .bolt.instr.counters
- Instrumentation snippet is inserted to each basic block

Runtime Library

A runtime instrumentation library is linked into the final binary during binary rewriting. The main functionality of runtime library:

- Initialization of instrumentation part
- Generation of profile after execution finishes/by timer option (has support for both shared libraries and execution files)
- Callbacks support for indirect calls handling
- Output an .fdata file (BOLT profile)

Instrumentation Pass

std::vector<MCInst> Instrumentation::createInstrumentationSnippet(BinaryContext &BC, bool IsLeaf)

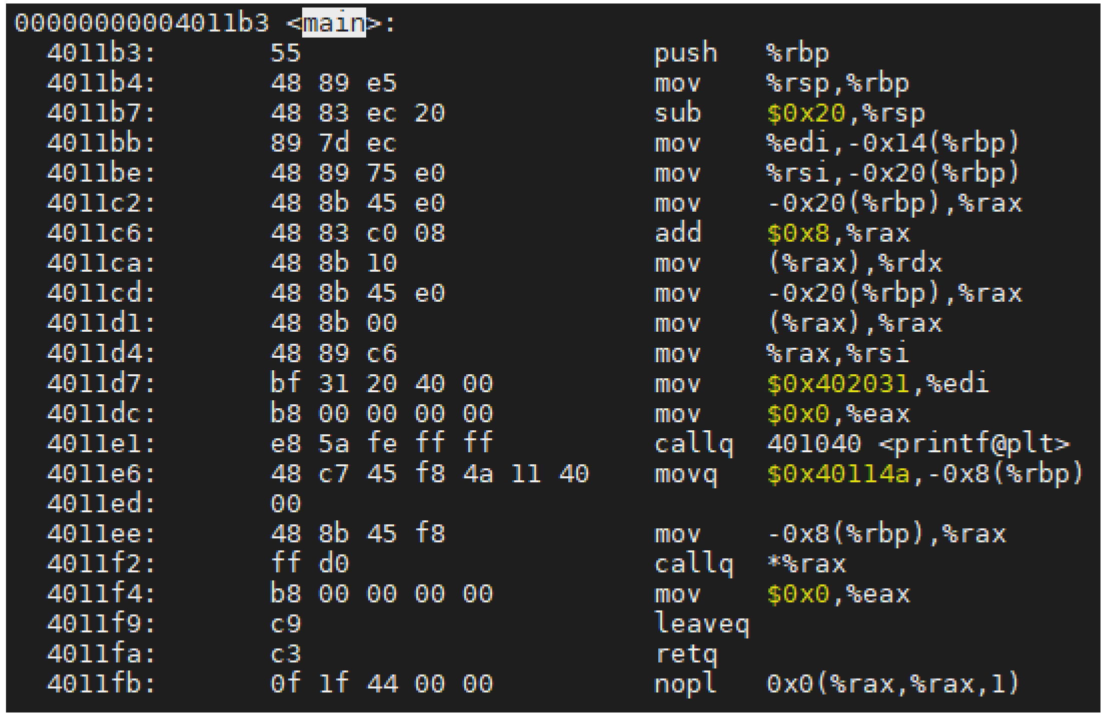
- Creates the sequence of instructions to increment a counter
- Uses MCInst class

MCInst class is a target-independent representation of an instruction, which holds a target-specific opcode and a vector of MCOperands

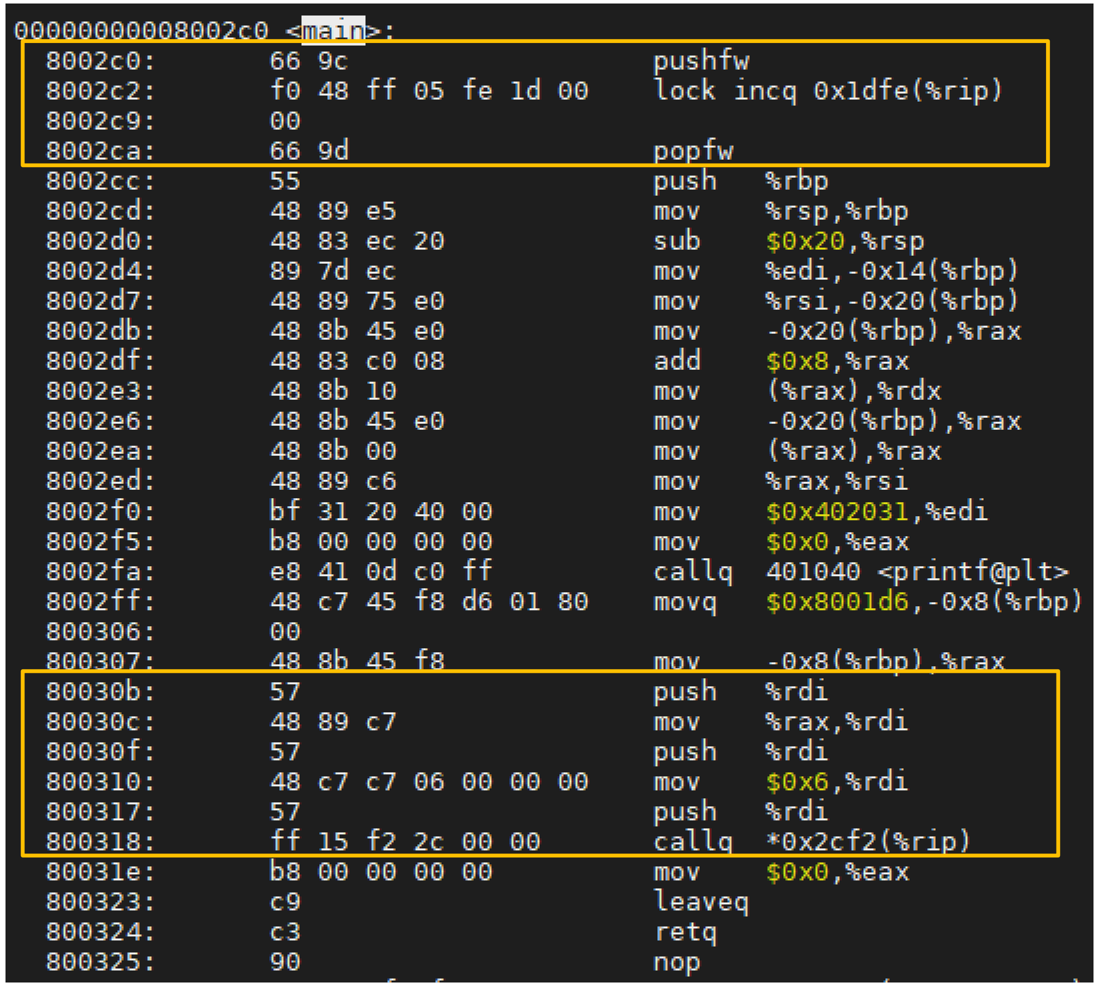
- MCOperand, in turn, is a simple discriminated union of three cases:
  1) a simple immediate
  2) a target register ID
  3) a symbolic expression (e.g. “Lfoo-Lbar+42”) as an MCExpr

```
class MCOperand {
   enum MachineOperandType : unsigned char {
   kInvalid, ///< Uninitialized.
   kRegister, ///< Register operand.
   kImmediate, ///< Immediate operand.
   kSFPImmediate, ///< Single-floating-point immediate.
   kDFPImmediate, ///< Double-Floating-point immediate.
   kExpr, ///< Relocatable immediate operand.
   kInst ///< Sub-instruction operand.
   };
…
```

Instrumentation Pass



```
00000000004011b3 <main>:
  4011b3:       55                      push   %rbp
  4011b4:       48 89 e5                mov    %rsp,%rbp
  4011b7:       48 83 ec 20             sub    $0x20,%rsp
  4011bb:       89 7d ec                mov    %edi,-0x14(%rbp)
  4011be:       48 89 75 e0             mov    %rsi,-0x20(%rbp)
  4011c2:       48 8b 45 e0             mov    -0x20(%rbp),%rax
  4011c6:       48 83 c0 08             add    $0x8,%rax
  4011ca:       48 8b 10                mov    (%rax),%rdx
  4011cd:       48 8b 45 e0             mov    -0x20(%rbp),%rax
  4011d1:       48 8b 00                mov    (%rax),%rax
  4011d4:       48 89 c6                mov    %rax,%rsi
  4011d7:       bf 31 20 40 00          mov    $0x402031,%edi
  4011dc:       b8 00 00 00 00          mov    $0x0,%eax
  4011e1:       e8 5a fe ff ff          callq  401040 <printf@plt>
  4011e6:       48 c7 45 f8 4a 11 40    movq   $0x40114a,-0x8(%rbp)
  4011ed:       00
  4011ee:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  4011f2:       ff d0                   callq  *%rax
  4011f4:       b8 00 00 00 00          mov    $0x0,%eax
  4011f9:       c9                      leaveq
  4011fa:       c3                      retq
  4011fb:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```



```
00000000008002c0 <main>:
  8002c0:       66 9c                   pushfw
  8002c2:       f0 48 ff 05 fe 1d 00    lock incq 0x1dfe(%rip)
  8002c9:       00
  8002ca:       66 9d                   popfw
  8002cc:       55                      push   %rbp
  8002cd:       48 89 e5                mov    %rsp,%rbp
  8002d0:       48 83 ec 20             sub    $0x20,%rsp
  8002d4:       89 7d ec                mov    %edi,-0x14(%rbp)
  8002d7:       48 89 75 e0             mov    %rsi,-0x20(%rbp)
  8002db:       48 8b 45 e0             mov    -0x20(%rbp),%rax
  8002df:       48 83 c0 08             add    $0x8,%rax
  8002e3:       48 8b 10                mov    (%rax),%rdx
  8002e6:       48 8b 45 e0             mov    -0x20(%rbp),%rax
  8002ea:       48 8b 00                mov    (%rax),%rax
  8002ed:       48 89 c6                mov    %rax,%rsi
  8002f0:       bf 31 20 40 00          mov    $0x402031,%edi
  8002f5:       b8 00 00 00 00          mov    $0x0,%eax
  8002fa:       e8 41 0d c0 ff          callq  401040 <printf@plt>
  8002ff:       48 c7 45 f8 d6 01 80    movq   $0x8001d6,-0x8(%rbp)
  800306:       00
  800307:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  80030b:       57                      push   %rdi
  80030c:       48 89 c7                mov    %rax,%rdi
  80030f:       57                      push   %rdi
  800310:       48 c7 c7 06 00 00 00    mov    $0x6,%rdi
  800317:       57                      push   %rdi
  800318:       ff 15 f2 2c 00 00       callq  *0x2cf2(%rip)
  80031e:       b8 00 00 00 00          mov    $0x0,%eax
  800323:       c9                      leaveq
  800324:       c3                      retq
  800325:       90                      nop
```

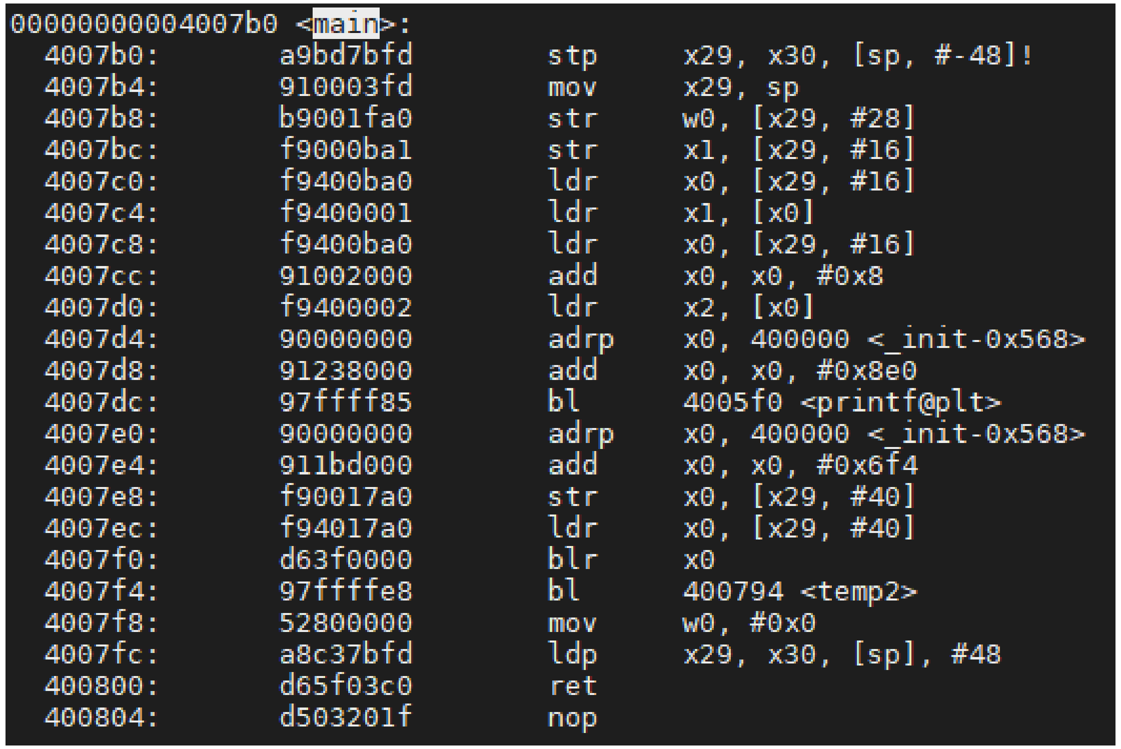
Instrumentation Pass

std::vector<MCInst> Instrumentation::createInstrumentationSnippet(BinaryContext &BC, bool IsLeaf)

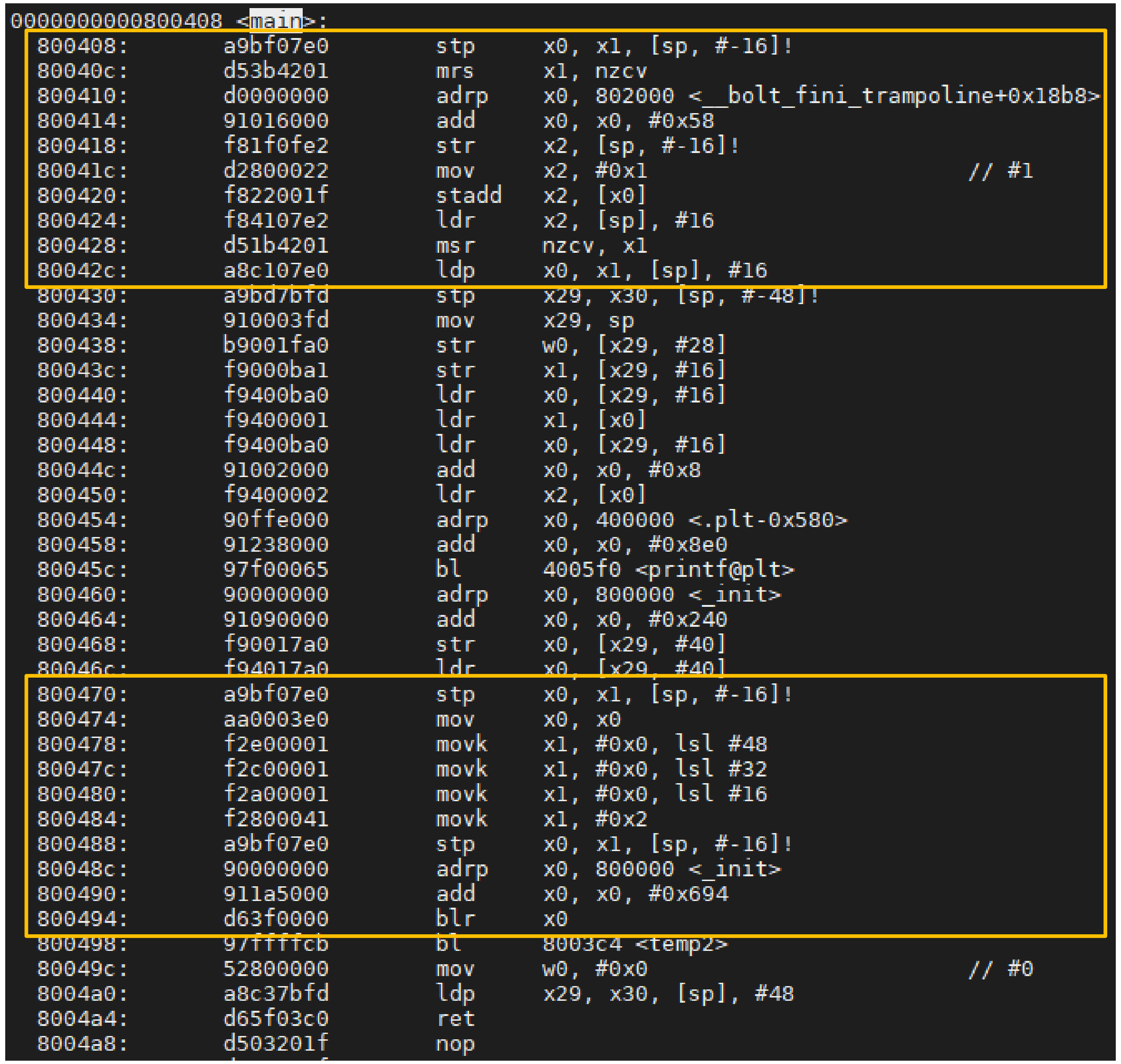
AArch64MCPlusBuilder

`createPushRegisters`	←	Store Pair of Registers (x0, x1)
`getSystemFlag`	←	Move condition flags NZCV to x1 register with MRS instruction
`materializeAddress`	←	Get page-aligned address and add page offset
`storeReg`	←	Store x2 register value to the stack
`createIncMemory`	←	Made atomic add to x0 through x2 register
`loadReg`	←	Load value from the stack back to x2
`setSystemFlag`	←	Restore condition flags value with MSR instruction
`createPopRegisters`	←	Restore x0 and x1 values from the stack

Instrumentation Pass



```
00000000004007b0 <main>:
  4007b0:       a9bd7bfd        stp     x29, x30, [sp, #-48]!
  4007b4:       910003fd        mov     x29, sp
  4007b8:       b9001fa0        str     w0, [x29, #28]
  4007bc:       f9000ba1        str     x1, [x29, #16]
  4007c0:       f9400ba0        ldr     x0, [x29, #16]
  4007c4:       f9400001        ldr     x1, [x0]
  4007c8:       f9400ba0        ldr     x0, [x29, #16]
  4007cc:       91002000        add     x0, x0, #0x8
  4007d0:       f9400002        ldr     x2, [x0]
  4007d4:       90000000        adrp    x0, 400000 <_init-0x568>
  4007d8:       91238000        add     x0, x0, #0x8e0
  4007dc:       97ffff85        bl      4005f0 <printf@plt>
  4007e0:       90000000        adrp    x0, 400000 <_init-0x568>
  4007e4:       911bd000        add     x0, x0, #0x6f4
  4007e8:       f90017a0        str     x0, [x29, #40]
  4007ec:       f94017a0        ldr     x0, [x29, #40]
  4007f0:       d63f0000        blr     x0
  4007f4:       97ffffe8        bl      400794 <temp2>
  4007f8:       52800000        mov     w0, #0x0
  4007fc:       a8c37bfd        ldp     x29, x30, [sp], #48
  400800:       d65f03c0        ret
  400804:       d503201f        nop
```



```
0000000000800408 <main>:
  800408:       a9bf07e0        stp     x0, x1, [sp, #-16]!
  80040c:       d53b4201        mrs     x1, nzcv
  800410:       d0000000        adrp    x0, 802000 <__bolt_fini_trampoline+0x18b8>
  800414:       91016000        add     x0, x0, #0x58
  800418:       f81f0fe2        str     x2, [sp, #-16]!
  80041c:       d2800022        mov     x2, #0x1                        // #1
  800420:       f822001f        stadd   x2, [x0]
  800424:       f84107e2        ldr     x2, [sp], #16
  800428:       d51b4201        msr     nzcv, x1
  80042c:       a8c107e0        ldp     x0, x1, [sp], #16
  800430:       a9bd7bfd        stp     x29, x30, [sp, #-48]!
  800434:       910003fd        mov     x29, sp
  800438:       b9001fa0        str     w0, [x29, #28]
  80043c:       f9000ba1        str     x1, [x29, #16]
  800440:       f9400ba0        ldr     x0, [x29, #16]
  800444:       f9400001        ldr     x1, [x0]
  800448:       f9400ba0        ldr     x0, [x29, #16]
  80044c:       91002000        add     x0, x0, #0x8
  800450:       f9400002        ldr     x2, [x0]
  800454:       90ffe000        adrp    x0, 400000 <.plt-0x580>
  800458:       91238000        add     x0, x0, #0x8e0
  80045c:       97f00065        bl      4005f0 <printf@plt>
  800460:       90000000        adrp    x0, 800000 <_init>
  800464:       91090000        add     x0, x0, #0x240
  800468:       f90017a0        str     x0, [x29, #40]
  80046c:       f94017a0        ldr     x0, [x29, #40]
  800470:       a9bf07e0        stp     x0, x1, [sp, #-16]!
  800474:       aa0003e0        mov     x0, x0
  800478:       f2e00001        movk    x1, #0x0, lsl #48
  80047c:       f2c00001        movk    x1, #0x0, lsl #32
  800480:       f2a00001        movk    x1, #0x0, lsl #16
  800484:       f2800041        movk    x1, #0x2
  800488:       a9bf07e0        stp     x0, x1, [sp, #-16]!
  80048c:       90000000        adrp    x0, 800000 <_init>
  800490:       911a5000        add     x0, x0, #0x694
  800494:       d63f0000        blr     x0
  800498:       97ffffcb        bl      8003c4 <temp2>
  80049c:       52800000        mov     w0, #0x0                        // #0
  8004a0:       a8c37bfd        ldp     x29, x30, [sp], #48
  8004a4:       d65f03c0        ret
  8004a8:       d503201f        nop
```

Direct And Indirect Calls Support

The main difference between the direct and the indirect call, is that:

- The direct call uses an instruction call with a fixed/relative address as argument. After the linker has done its job, this address will be included in the opcode.
- The indirect call uses an instruction call with a register as argument. The register is previously loaded either directly with the fixed address of the subroutine that is to be called, or with a value fetched from somewhere else, such as another register or a place in memory where the subroutine’s address was previously stored.

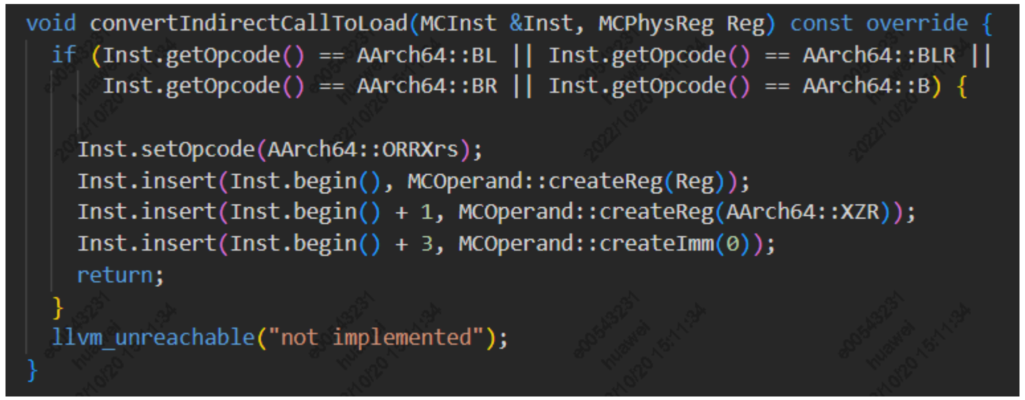
Direct And Indirect Calls Support

instrumentIndirectTarget from *Instrumentation Pass*

createInstrumentedIndirectCall from *AArch64MCPlusBuilder*

```
//    Code sequence used to enter indirect call instrumentation helper:
//    stp x0, x1, [sp, #-16]!
//        mov target x0 -> orr x0 target xzr
//    mov target, x0
//        mov x1 CallSiteID createLoadImmediate ->
//    movk    x1, #0x0, lsl #48
//    movk    x1, #0x0, lsl #32
//    movk    x1, #0x0, lsl #16
//    movk    x1, #0x0
//    stp x0, x1, [sp, #-16]!
//        bl *HandlerFuncAddr createIndirectCall ->
//    adr x0 *HandlerFuncAddr -> adrp + add
//    blr x0
```

convertIndirectCallToLoad



```
void convertIndirectCallToLoad(MCInst &Inst, MCPhysReg Reg) const override {
  if (Inst.getOpcode() == AArch64::BL || Inst.getOpcode() == AArch64::BLR ||
      Inst.getOpcode() == AArch64::BR || Inst.getOpcode() == AArch64::B) {

    Inst.setOpcode(AArch64::ORRXrs);
    Inst.insert(Inst.begin(), MCOperand::createReg(Reg));
    Inst.insert(Inst.begin() + 1, MCOperand::createReg(AArch64::XZR));
    Inst.insert(Inst.begin() + 3, MCOperand::createImm(0));
    return;
  }
  llvm_unreachable("not implemented");
}
```

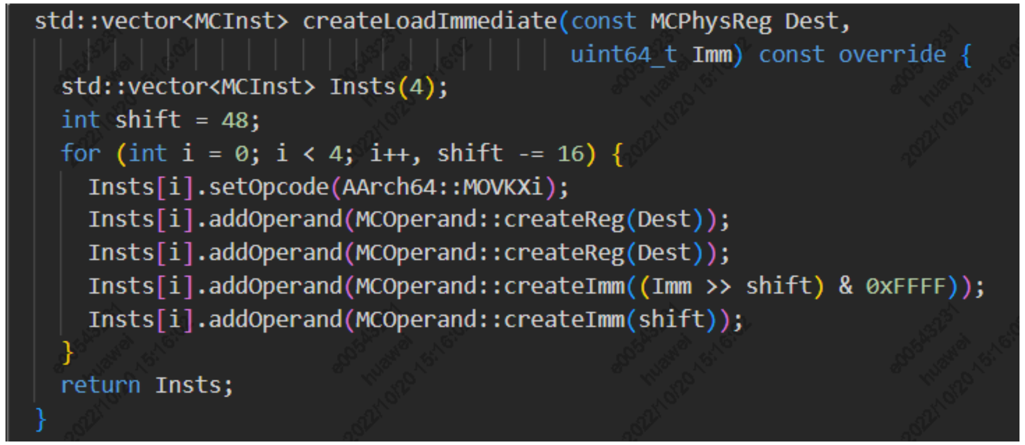
Direct And Indirect Calls Support

instrumentIndirectTarget from *Instrumentation Pass*

createInstrumentedIndirectCall from *AArch64MCPlusBuilder*

```
//    Code sequence used to enter indirect call instrumentation helper:
//    stp x0, x1, [sp, #-16]!
//        mov target x0 -> orr x0 target xzr
//    mov target, x0
//        mov x1 CallSiteID createLoadImmediate ->
//    movk    x1, #0x0, lsl #48
//    movk    x1, #0x0, lsl #32
//    movk    x1, #0x0, lsl #16
//    movk    x1, #0x0
//    stp x0, x1, [sp, #-16]!
//        bl *HandlerFuncAddr createIndirectCall ->
//    adr x0 *HandlerFuncAddr -> adrp + add
//    blr x0
```

createLoadImmediate



```
std::vector<MCInst> createLoadImmediate(const MCPhysReg Dest,
                                        uint64_t Imm) const override {
  std::vector<MCInst> Insts(4);
  int shift = 48;
  for (int i = 0; i < 4; i++, shift -= 16) {
    Insts[i].setOpcode(AArch64::MOVKXi);
    Insts[i].addOperand(MCOperand::createReg(Dest));
    Insts[i].addOperand(MCOperand::createReg(Dest));
    Insts[i].addOperand(MCOperand::createImm((Imm >> shift) & 0xFFFF));
    Insts[i].addOperand(MCOperand::createImm(shift));
  }
  return Insts;
}
```

Direct And Indirect Calls Support

instrumentIndirectTarget from *Instrumentation Pass*

createInstrumentedIndirectCall from *AArch64MCPlusBuilder*

createInstrumentedIndCallTrampoline

`__bolt_instr_ind_call_handler_func`

HandlerFunc

`__bolt_instr_ind_tailcall_handler_func`

Direct And Indirect Calls Support

instrumentIndirectTarget from *Instrumentation Pass*

createInstrumentedIndirectCall from *AArch64MCPlusBuilder*

createInstrumentedIndCallTrampoline

`__bolt_instr_ind_call_handler_func`

HandlerFunc

`__bolt_instr_ind_tailcall_handler_func`

```
//   Code sequence for instrumented indirect call trampoline:
//   stp     x0, x1, [sp, #-16]!
//   mrs     x1, nzcv
//   adr     x0, InstrTrampoline -> adrp + add
//   ldr     x0, [x0]
//   subs    x0, x0, #0x0
//   b.eq    IndCallHandler
//   str     x30, [sp, #-16]!
//   blr      x0
//   ldr     x30, [sp], #16
//   b       IndCallHandler
```

Direct And Indirect Calls Support

instrumentIndirectTarget from *Instrumentation Pass*

createInstrumentedIndirectCall from *AArch64MCPlusBuilder*

createInstrumentedIndCallTrampoline from *AArch64MCPlusBuilder*

__bolt_instr_indirect_call() *from runtime library*

instrumentIndirectCall *from runtime library*

```
extern "C" __attribute((naked))
           void __bolt_instr_indirect_call()
{
#if defined(__aarch64__)
  __asm__ __volatile__(SAVE_ALL
             "ldp x0, x1, [sp, #288]\n"
             "bl instrumentIndirectCall\n"
             RESTORE_ALL
             "ret\n"
             :::);
…
```

Direct And Indirect Calls Support

instrumentIndirectTarget from *Instrumentation Pass*

createInstrumentedIndirectCall from *AArch64MCPlusBuilder*

createInstrumentedIndCallTrampoline from *AArch64MCPlusBuilder*

__bolt_instr_indirect_call() *from runtime library*

instrumentIndirectCall *from runtime library*

createInstrumentedIndCallHandler from *AArch64MCPlusBuilder*

```
//   Code  sequence  for  instrumented
     indirect call handler:
 //   msr  nzcv, x1
 //   ldp  x0, x1, [sp], #16
 //   ldr  x16, [sp], #16
 //   ldp  x0, x1, [sp], #16
 //   br   x16
```

Runtime library/syscalls

Runtime library code is conceived to be independent of any external libraries →

This requires us to develop a set of syscall wrappers for AArch64 →

Around 20 system calls were implemented

```
uint64_t __nanosleep(const timespec *req, timespec *rem) {
  uint64_t ret;
  __asm__ __volatile__("movq $35, %%rax\n"
                       "syscall\n"
                       : "=a"(ret)
                       : "D"(req), "S"(rem)
                       : "cc", "rcx", "r11", "memory");
  return ret;
}
```

nanosleep syscall AArch64 implementation to dump profile data at user-specified intervals

How To Use

With instrumentation

- **Step 0: Build binary**
  - It should be unstripped
  - Add --emit-relocs or -q linker flag
- **Step 1: Instrument binary**

```
llvm-bolt <executable> -instrument -o <instrumented-executable>
```

- **Step 2: Collect Profile with instrumented binary**

```
$ ./<instrumented-executable>
```

- **Step 3: Optimize with BOLT**

```
$ llvm-bolt <executable> -o <executable>.bolt -data=/tmp/prof.fdata -reorder-blocks=cache+ -reorder-functions=hfsort -split-functions=2 -split-all-cold -split-eh -dyno-stats
```

Results

- Up to **20%** of relative performance improvement on internal applications
- Redis benchmark:
    - Up to 5% improvements on SET/GET benchmarks with profile collected with instrumentation. No improvements in case profile was collected with perf.
- Upstream patches preparation - in progress

Thank you.