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

1. Intro
2. Problem Statement
3. Architecture
4. Instrumentation Pass
5. Direct And Indirect Calls Support
6. Runtime library/syscalls
7. How To Use
8. Results
• 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 14\textsuperscript{th} 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

Function discovery
Read Debug Info
Disassembly
CFG Construction
Read Profile Data
Run Optimization Passes
 Emit and Link Functions
Update Debug Info
Rewrite Binary File

Runtime instrumentation library

Instrumentation pass is here
Instrumentation Pass

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

```cpp
class MCOperand {
    enum MachineOperandType : unsigned char {
        kInvalid, ///< Uninitialized.
        kRegister, ///< Register operand.
        kImmediate, ///< Immediate operand.
        kSFPIImmediate, ///< Single-floating-point immediate.
        kDFPI Immediate, ///< Double-Floating-point immediate.
        kExpr, ///< Relocatable immediate operand.
        kInst ///< Sub-instruction operand.
    };
};
...```
### Instrumentation Pass

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

```cpp
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

```
4007b0:    a9c775fd    stp    x29, x30, [sp, #-48]!
4007b4:    916e03fd    mov    x29, sp
4007b8:    b9601fa0    str    w0, [x29, #28]
4007bc:    f9600b1    str    x1, [x29, #16]
4007c0:    f9400ba0    ldr    x0, [x29, #16]
4007c4:    f9400001    ldr    x1, [x0]
4007c8:    f9400ba0    ldr    x0, [x29, #16]
4007cc:    f9102960    add    x0, x0, #x8
4007d0:    f9400002    ldr    x2, [x0]
4007d4:    90600800    adrp    x0, 400000 <__init_0x568>
4007d8:    91238960    add    x0, x0, #0x860
4007dc:    97fff0e5    bl    4005f0 <printf@plt>
4007e0:    98600000    adrp    x0, 400000 <__init_0x568>
4007e4:    911b0d00    add    x0, x0, #0x6f4
4007e8:    f9601e0    str    x0, [x29, #40]
4007ec:    f94017a0    ldr    x0, [x29, #40]
4007f0:    d63f0000    blr    x0
4007f4:    97fff0e8    bl    40079d <temp2>
4007f8:    52800900    mov    w0, #0x0
4007fc:    a8c37b0d    ldp    x29, x30, [sp], #48
400800:    d56593c0    ret
400804:    d563201f    nop
```

```
800408:    a9b0f7e0    stp    x0, x1, [sp, #-16]!
80040c:    d52b4291    mrs    x1, nzcv
800410:    d06b0000    adrp    x0, 802000 <bolt_fini_trampoline+0x186b>
800414:    91010800    add    x0, x0, #0x8
800418:    f81f0e2    str    x2, [sp, #-16]!
80041c:    d2800202    mov    x2, #0x1
800420:    f822001f    stadd    x2, [x0]
800424:    f64307e2    ldr    x2, [sp, #16]
800428:    d51b4210    mosr    nzcv, x1
80042c:    a8c107e0    ldp    x0, x1, [sp], #16
800430:    a9ef76fd    stp    x0, x0, [sp, #-48]!
800434:    910003f0    mov    x9, sp
800438:    b9001fa0    str    w0, [x29, #28]
80043c:    f9000001    str    x1, [x29, #16]
800440:    f9400000    ldr    x0, [x29, #16]
800444:    f9400001    ldr    x1, [x0]
800448:    f9002000    add    x0, x0, #0x8
80044c:    91002000    add    x0, x0, #0x8
800450:    f9400000    ldr    x2, [x0]
800454:    90ffe000    adrp    x0, 400000 <.plt-0x568>
800458:    91238960    add    x0, x0, #0x860
80045c:    9ff00005    bl    400f0 <printf@plt>
800460:    90000000    adrp    x0, 800000 <__init>
800464:    91000000    add    x0, x0, #0x1240
800468:    f9601e0    str    x0, [x29, #40]
80046c:    f94017a0    ldr    x0, [x29, #40]
800470:    a9b0f7e0    stp    x0, x1, [sp, #-16]!
800474:    a90003e0    mov    x0, x0
800478:    f2080001    movk    x2, #0x0, lsl #48
80047c:    f2000001    movk    x1, #0x0, lsl #32
800480:    f2a00001    movk    x0, #0x0, lsl #16
800484:    f2800041    movk    x1, #0x2
800488:    a9b0f7e0    stp    x0, x1, [sp, #-16]!
80048c:    90000000    adrp    x0, 800000 <__init>
800490:    91a15000    add    x0, x0, #0x694
800494:    d63f0000    blr    x0
800498:    97fff0e8    bl    8003c4 <temp2>
80049c:    52660800    mov    w8, #0x8
8004a0:    a8c37b0d    ldp    x29, x30, [sp], #48
8004a4:    d56593c0    ret
8004a8:    d563201f    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*

```assembly
// 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
```
Direct And Indirect Calls Support

instrumentIndirectTarget from *Instrumentation Pass*

createInstrumentedIndirectCall from *AArch64MCPlusBuilder*

```plaintext
// 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 *HandleFuncAddr createIndirectCall ->
//   adr x0 *HandleFuncAddr -> adrp + add
// blr x0
```
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

```c
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*

```c
// 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

```c
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:
  
  o 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.