Extending the LLVM AArch32 JITLink backend

Stefan Gränitz · Freelance Compiler Developer · Phabricator / Discord: @sgraenitz
Extending the LLVM AArch32 JITLink backend
Target-specific complexities and mitigations

(1) Small embedded systems are no suitable development machines

(2) Architecture and considerations in the initial patch
   • Mixing two instruction sets: ARM and Thumb
   • High density encodings
   • Many sub-arches with varying requirements

(3) Let’s iterate! A practical development workflow.
Embedded systems are no suitable development machines

LLVM is too heavy to build on the device

→ git clone https://github.com/llvm/llvm-project
→ du -sh llvm-project
   5.6G

→ cd llvm-project
→ cmake -Sllvm -Bbuild -GNinja -DLLVM_TARGETS_TO_BUILD=ARM

→ ninja -C build -t commands llvm-jitlink | wc -l
   1510

→ ninja -C build -t commands llvm-jitlink-executor | wc -l
   289
Remote execution in llvm-jitlink

llvm-jitlink supports remote execution:

```
> llvm-jitlink --help | grep connect
--oop-executor-connect=<string>   Connect to an out-of-process
        executor via TCP
```

Simple option: Build llvm-jitlink-executor from a stripped fork on-device

```
> git clone https://github.com/echtzeit-dev/llvm-jitlink-executor
> cd llvm-jitlink-executor
> CC=clang CXX=clang++ cmake -Sllvm -Bbuild -GNinja \
    -DCMAKE_BUILD_TYPE=Release -DLLVM_TARGETS_TO_BUILD=host \
    -DLLVM_USE_LINKER=lld -DLLVM_PARALLEL_LINK_JOBS=1
> ninja -j2 llvm-jitlink-executor
```

What about debugging?

- There is support for remote debugging in ORCv2: examples/LLJITWithRemoteDebugging.cpp
- Blog post with more details for LLDB: GDB JIT Interface 101
Remote debugging with LLDB

Latest LLDB release still lacked AArch32 relocation support
Ticket: llvm-project#61948    Fix on mainline: D147642

Cross-compile remote tools with docker, e.g. for latest stable bullseye Raspbian: https://hub.docker.com/r/echtzeit/ubuntu-cross-aarch32-lldb

→ cmake -S llvm -G Ninja -B build-aarch32 -DCMAKE_BUILD_TYPE=Release
   -DLLVM_ENABLE_PROJECTS="clang;lldb;llvm" \
   -DCMAKE_CROSSCOMPILING=1 \n   -DCMAKE_C_COMPILER=arm-linux-gnueabihf-gcc-10 \n   -DCMAKE_CXX_COMPILER=arm-linux-gnueabihf-g++-10 \n   -DLLVM_HOST_TRIPLE=arm-unknown-linux-gnueabihf \n   -DLLDB_TABLEGEN=/root/build-host/bin/lldb-tblgen \n   -DLLVM_TABLEGEN=/root/build-host/bin/llvm-tblgen \n   -DCLANG_TABLEGEN=/root/build-host/bin/clang-tblgen \n   -DLLDB_ENABLE_PYTHON=Off \n   -DLLDB_ENABLE_LIBEDIT=Off \n   -DLLDB_ENABLE_LIBXML2=Off \n   -DLLDB_ENABLE_CURSES=Off

→ ninja -C build-aarch32 llvm-jitlink-executor lldb-server
[2727/2727] Linking CXX executable bin/lldb-server
Remote debugging with LLDB

Device runs:

> lldb-server platform --server --listen 0.0.0.0:9001 --gdbserver-port 9002 &
  [1] 8850
> build_main/bin/llvm-jitlink-executor listen=0.0.0.0:20000
  Listening at 0.0.0.0:20000

Connect LLDB and let it attach to the executor:

> lldb -o 'platform select remote-linux'
  -o 'platform connect connect://192.168.45.9:9001'
  -o 'process attach --name llvm-jitlink-executor'
  -o 'b TargetExecutionUtils.cpp:43'

Connect llvm-jitlink and run the demo:

> llvm-jitlink --oop-executor-connect=192.168.45.9:20000 demo.o
Remote debugging with LLDB

Set breakpoint on JITed main from LLDB prompt:

(lldb) c
Process 2258 resuming
Process 2258 stopped
* thread #4, name = 'llvm-jitlink-ex', stop reason = breakpoint 1.1
  frame #0: 0x0012930c [...] at TargetExecutionUtils.cpp:43:3

(lldb) b main
Breakpoint 2: 2 locations.

(lldb) br list 2
2: name = 'main', locations = 2, resolved = 2, hit count = 0
  2.1: where = JIT(0x76fd9000)`main`, address = 0x76fdb06a, resolved, hit count = 0
  2.2: where = llvm-jitlink-executor`main + 28 at llvm-jitlink-executor.cpp:120:35, address = 0x000dade4, resolved, hit count = 0

Fixed on mainline
Two instruction sets: ARM and Thumb

- AArch32 supports 2 instruction-set states: ARM and Thumb
- Transition between states needs interworking
- Encoded in least-significant bit in branch target address (Thumb bit)
- Tells the linker to emit a state changing instruction, e.g. change instructions "branch" and "branch with link" to special forms:

  B → BX
  BL → BLX
Thumb bit

Least significant bit of a branch offset encodes state change

\[
\begin{align*}
\text{R}_\text{ARM}_\text{THM}_\text{CALL} \\
0x00000041 & \quad 0x00000040 \\
\text{BL} & \quad \text{BLX}
\end{align*}
\]
**Thumb bit**

Least significant bit of a branch offset encodes state change

- Offsets are calculated from symbols in LinkGraph
- LinkGraph aims to be target-agnostic!
Per symbol target-flags added to LinkGraph

JITLink big picture:

createLinkGraphFromELFObject_aarch32()  \(\downarrow\)

JITLinkContext  LinkGraph  \(\downarrow\)

link_ELF_aarch32()  \(\times\)

→ Inputs: LinkGraph and JITLinkContext

→ [D146641](#) introduced target-flags to represent Thumb-bit in LinkGraph
High density encodings

• ARM: 4 byte fixed-length

• Thumb: variable-sized, most common instructions are 2-byte

• 20% size reduction for Linux (2011) for code and read-only data:

```shell
> size vmlinux-*/
text    data    bss    dec    hex   filename
8420507  463356  826928  9710791  942cc7  vmlinux-arm
6715539  463260  826928  8005727  7a285f  vmlinux-thumb2
```

• Instruction encoding makes heavy use of bit shifting
REL-type relocations store addends in-place

- RELA relocation record = offset + value + addend
- REL relocation record = only offset + value

→ Addend stored in-place: value bits of target instruction
→ Records take less space
→ Linker must be able to decode instructions to extract addend
Separating encoding/decoding from fix-up logic

Promotes a symmetric implementation, e.g.: JITLink/aarch32.cpp#L56-L82

```c
/// Encode 25-bit immediate value for branch instructions with J1J2 range
/// extension (formats B T4, BL T1 and BLX T2).
HalfWords encodeImmBT4BlT1BlxT2_J1J2(int64_t Value) {
    uint32_t S = (Value >> 14) & 0x0400;
    uint32_t J1 = (((~(Value >> 10)) ^ (Value >> 11)) & 0x2000);
    uint32_t J2 = (((~(Value >> 11)) ^ (Value >> 13)) & 0x0800);
    uint32_t Imm10 = (Value >> 12) & 0x03ff;
    uint32_t Imm11 = (Value >> 1) & 0x07ff;
    return HalfWords{S | Imm10, J1 | J2 | Imm11};
}

/// Decode 25-bit immediate value for branch instructions with J1J2 range
/// extension (formats B T4, BL T1 and BLX T2).
int64_t decodeImmBT4BlT1BlxT2_J1J2(uint32_t Hi, uint32_t Lo) {
    uint32_t S = Hi & 0x0400;
    uint32_t I1 = ~(Lo ^ (Hi << 3)) << 10) & 0x00800000;
    uint32_t I2 = ~(Lo ^ (Hi << 1)) << 11) & 0x00400000;
    uint32_t Imm10 = Hi & 0x03ff;
    uint32_t Imm11 = Lo & 0x07ff;
    return SignExtend64<25>(S << 14 | I1 | I2 | Imm10 << 12 | Imm11 << 1);
}
```
Separating encoding/decoding from fix-up logic

Allows testing in isolation, e.g.: JITLink/AArch32Tests.cpp#L105-L123

```cpp
auto EncodeDecode = [ImmMask](int64_t In, MutableHalfWords &Mem) {
    Mem.patch(encodeImmBT4BlT1BlxT2_J1J2(In), ImmMask);
    return decodeImmBT4BlT1BlxT2_J1J2(Mem.Hi, Mem.Lo);
};

for (MutableHalfWords Mem : MemPresets) {
    HalfWords UnaffectedBits(Mem.Hi & ~ImmMask.Hi, Mem.Lo & ~ImmMask.Lo);

    EXPECT_EQ(EncodeDecode(1, Mem), 0); // Zero value
    EXPECT_EQ(EncodeDecode(0x41, Mem), 0x40); // Common value
    EXPECT_EQ(EncodeDecode(16777215, Mem), 16777214); // Maximum value
    EXPECT_EQ(EncodeDecode(-16777215, Mem), -16777216); // Minimum value
    EXPECT_NE(EncodeDecode(16777217, Mem), 16777217); // First overflow
    EXPECT_NE(EncodeDecode(-16777217, Mem), -16777217); // First underflow

    EXPECT_TRUE(UnaffectedBits.Hi == (Mem.Hi & ~ImmMask.Hi) &&
                UnaffectedBits.Lo == (Mem.Lo & ~ImmMask.Lo)) << "Diff outside immediate field";
}
Grouping bit-offsets in FixupInfo<> per EdgeKind

Common structure and allows generic operations, e.g.: JITLink/aarch32.cpp#L186-L193

```cpp
template <EdgeKind_aarch32 Kind> struct FixupInfo {}

template <> struct FixupInfo<Thumb_Call> {
    static constexpr HalfWords Opcode{0xf000, 0xc000};
    static constexpr HalfWords OpcodeMask{0xf800, 0xc000};
    static constexpr HalfWords ImmMask{0x07ff, 0x2fff};
    static constexpr uint16_t LoBitH = 0x0001;
    static constexpr uint16_t LoBitNoBlx = 0x1000;
};

template <EdgeKind_aarch32 Kind>
void writeImmediate(WritableThumbRelocation &R, HalfWords Imm) {
    static constexpr HalfWords Mask = FixupInfo<Kind>::ImmMask;
    assert((Mask.Hi & Imm.Hi) == Imm.Hi && (Mask.Hi & Imm.Hi) == Imm.Hi &&
        "Value bits exceed bit range of given mask");
    R.Hi = (R.Hi & ~Mask.Hi) | Imm.Hi;
    R.Lo = (R.Lo & ~Mask.Lo) | Imm.Lo;
}

writeImmediate<Thumb_Call>(R, encodeImmBT4BlT1BlxT2_J1J2(Value));
```
Indirection stubs

Used in all JITLink backends:

• Mimic program-load-table
• Extend branch ranges

Implementation:

• Short sequence of linker-generated code, e.g. JITLink/x86_64.cpp#L75-L76

```cpp
const char PointerJumpStubContent[6] = {
    static_cast<char>(0xFFu), 0x25, 0x00, 0x00, 0x00, 0x00};
```
**Flavors of indirection stubs**

AArch32 requires different kinds of stubs — called "Flavors"

- State change ARM ↔ Thumb must go through interworking stubs for indirections that are not R_ARM_(THM)_CALL

- LLD calls it "Thunks" supports **14 different forms**:
  
  **ARM:** V7ABS/PILong, V5LongLdrPc, V4ABS/PILongBX, V4PILong
  
  **Thumb:** V7ABS/PILong, V6MABS/PILong, V4ABS/PILongBX, V4ABS/PILong
Flavors of indirection stubs

AArch32 JITLink backend prepared for different stub "Flavors"

```cpp
const uint8_t Thumbv7ABS[] = {
    0x40, 0xf2, 0x00, 0x0c, // movw r12, #0x0000 ; lower 16-bit
    0xc0, 0xf2, 0x00, 0x0c, // movt r12, #0x0000 ; upper 16-bit
    0x60, 0x47 // bx   r12
};

template <>
Symbol &StubsManager<aarch32::Thumbv7>::createEntry(LinkGraph &G, Symbol &Target) {
    constexpr uint64_t Alignment = 4;
    Block &B = addStub(G, Thumbv7ABS, Alignment);
    ...
    B.addEdge(Thumb_MovwAbsNC, 0, Target, 0);
    B.addEdge(Thumb_MovtAbs, 4, Target, 0);
    Symbol &Stub = G.addAnonymousSymbol(B, 0, B.getSize(), true, false);
    Stub.setTargetFlags(ThumbSymbol);
    return Stub;
}
```
Many sub-arches with varying requirements

Support/ARMBuildAttributes.h#L91-L113

```c
// Legal Values for CPU_arch, (=6), uleb128
enum CPUArch {
    Pre_v4 = 0,
    v4 = 1,       // e.g. SA110
    v4T = 2,      // e.g. ARM7TDMI
    v5T = 3,      // e.g. ARM9TDMI
    v5TE = 4,     // e.g. ARM946E_S
    v5TEJ = 5,    // e.g. ARM926EJ_S
    v6 = 6,       // e.g. ARM1136J_S
    v6KZ = 7,     // e.g. ARM1176JZ_S
    v6T2 = 8,     // e.g. ARM1156T2_S
    v6K = 9,      // e.g. ARM1176JZ_S
    v7 = 10,      // e.g. Cortex A8, Cortex M3
    v6_M = 11,    // e.g. Cortex M1
    v6S_M = 12,   // v6_M with the System extensions
    v7E_M = 13,   // v7_M with DSP extensions
    v8_A = 14,    // v8_A AArch32
    v8_R = 15,    // e.g. Cortex R52
    v8_M_Base = 16, // v8_M_Base AArch32
    v8_M_Main = 17, // v8_M_Main AArch32
    v8_1_M_Main = 21, // v8_1_M_Main AArch32
    v9_A = 22,    // v9_A AArch32
};
```
Many sub-arches with varying requirements

- Endianness derived from triple, but other details are not
- Add relevant options to ArmCfg to pass them through the link phases:

```c
/// JITLink sub-arch configuration for Arm CPU models
struct ArmConfig {
    bool J1J2BranchEncoding = false;
    StubsFlavor Stubs = Unsupported;
};
```
Going forward

Let's iterate!
Going forward: Good next steps

See Eymen's excellent GSoC proposal: docs.google.com

- GOT-free relocations (plus Thumb equivalents)
  R_ARM_PREL31, R_ARM_ABS32, R_ARM_TARGET1, R_ARM_JUMP24, R_ARM_CALL

- GOT/PLT infrastructure and relocations (plus Thumb equivalents)
  R_ARM_GOT_PREL, R_ARM_BASE_PREL, R_ARM_GOTOFF32, R_ARM_GOT_BREL

- LIT test for each relocation type

- ORC Runtime: TLS support and exception handling

- clang-repl integration
Going forward: Trial and error

Take an example that works:

```c
int printf(const char *fmt, ...);
int main()
{
    printf("%d\n", 42);
    return 0;
}
```

```
> cat test.c
int printf(const char *fmt, ...);
int main()
{
    printf("%d\n", 42);
    return 0;
}
```

```
> clang -target arm-linux-gnueabihf -march=v7m -mthumb -c test.c -o test.o
> llvm-jitlink -noexec test.o
> echo $?
0
```

```
> clang -target arm-linux-gnueabihf -march=v7m -mthumb -fPIC -c test.c -o test.o
> llvm-jitlink -noexec test.o
> echo $?
0
```
Going forward: Trial and error

Change code until it fails:

```c
int printf(const char *fmt, ...);
+int global_var = 42;
int main()
-
  printf("%d\n", 42);
+  printf("%d\n", global_var);
  return 0;
}
```

```bash
> clang --target arm-linux-gnueabihf --march=v7m --mthumb --c test.c -o test.o
> llvm-jitlink --noexec test.o
> echo $?
0
```

```bash
> clang --target arm-linux-gnueabihf --march=v7m --mthumb -fPIC --c test.c -o test.o
> llvm-jitlink --noexec test.o
llvmlite.jitlink error: Unsupported aarch32 relocation 96: R_ARM_GOT_PREL
```
Going forward: Trial and error

Inspect debug output:

```
> clang -target arm-linux-gnueabihf -march=v7m -mthumb -fPIC -c test.c -o test.o
> llvm-jitlink --debug-only=jitlink --noexec test.o
Building jitlink graph for new input test.o...
Created ELFLinkGraphBuilder for "test.o" Preparing to build...
    Creating graph sections...
    0: has type SHT_NULL. Skipping.
    1: Creating section for ".strtab"
    ...
    Creating graph symbols...
        Adding symbols from symtab section ".symtab"
        0: Not creating graph symbol for ELF symbol "" with unrecognized type
        ...
        6: Creating defined graph symbol for ELF symbol "main"
        7: Creating external graph symbol for ELF symbol "printf"
        8: Creating defined graph symbol for ELF symbol "global_var"
Processing relocations:
    .text:
        edge@0x18: 0x0 + 0x18 -- Thumb_Call -> printf + -4
llvm-jitlink error: Unsupported aarch32 relocation 96: R_ARM_GOT_PREL
```
/// Translate from ELF relocation type to JITLink-internal edge kind.
Expected<
aarch32::EdgeKind_aarch32> getJITLinkEdgeKind(uint32_t ELFType) {
    switch (ELFType) {
    case ELF::R_ARM_ABS32:
        return aarch32::Data_Pointer32;
    ...
    case ELF::R_ARM_THM_CALL:
        return aarch32::Thumb_Call;
    ...
    }

    return make_error<JITLinkError>(
        "Unsupported aarch32 relocation " + formatv("{0:d}: ", ELFType) +
        object::getELFRelocationTypeName(ELF::EM_ARM, ELFType));
}
Going forward: Trial and error

Change target / sub-arch:

> clang --target arm-none-gnueabi --march=v6m -c test.c -o test.o
> llvm-jitlink -noexec test.o
llvm-jitlink error: Failed to build ELF link graph: Unsupported CPU arch v6_M

> clang --target arm-none-gnueabi --mcpu=cortex-m0plus --mthumb -c test.c -o test.o
> llvm-jitlink -noexec test.o
llvm-jitlink error: Failed to build ELF link graph: Unsupported CPU arch v6_M

> clang --target thumbv6m-none-gnueabi -c test.c -o test.o
> llvm-jitlink -noexec test.o
llvm-jitlink error: Failed to build ELF link graph: Unsupported CPU arch v6_M
Going forward: Trial and error

Try C++ input:

```cpp
# cat test.cpp
extern "C" int printf(const char *fmt, ...);
struct TestBase {
    virtual void hello() { printf("%d\n", 42); }
};

TestBase test;
int main() {
    test.hello();
    return 0;
}
```

```bash
→ clang++ -target arm-linux-gnueabihf -march=v7m -mthumb -c test.cpp -o test.o
→ llvm-jitlink -noexec test.o
```

```
llvm-jitlink error: Unsupported aarch32 relocation 38: R_ARM_TARGET1
```
Demo

Since code is more relatable than bullet points..

• Inspect working memory

• Debug on-device
Wrapping up: inspect working memory

Read instruction halfwords in working memory before and after applyFixup():

```
(lldb) mem read -c2 -s2 -fx &R.Hi
0x104016159: 0xf7ff 0xffffe

(lldb) mem read -c2 -s2 -fx &R.Hi
0x104016159: 0xf000 0xf816
```

Disassemble for target instruction set:

```
(lldb) dis -c1 -s &R.Hi
0x0000000104016159: addb %dh, %al

(lldb) dis -A thumb -c1 -s &R.Hi
0x0000000104016159: bl 0x04016159
```
Wrapping up: source info on mainline LLDB

Hurray! Source-level debugging of remote cross-JITed code for AArch32:

(lldb) version
lldb version 17.0.0git

(lldb) c
Process 1285 resuming
Process 1285 stopped
* thread #4, name = 'llvm-jitlink-ex', stop reason = breakpoint 2.1
  frame #0: 0x76f1d078 JIT(0x76f1b000)`main(argc=0, argv=0x75700c68) at base64char2val.c:40:7
  37   }
  38
  39   int main(int argc, char *argv[]) {
-> 40     if (argc == 1) {
    41       printf("Usage: %s term [more terms]\n", argv[0]);
    42       return 1;
    43   }

(lldb) p argc
(int) 3
(lldb) p argv[1]
(char *) 0x75700cf8 "Hello"
(lldb) p argv[2]
(char *) 0x75700d08 "ARM"
Extending the LLVM AArch32 JITLink backend

Thanks for your attention! Questions?

Past reviews: D144083, D146641, D147642

Useful links: github.com/echtzeit-dev/llvm-jitlink-executor
hub.docker.com/r/echtzeit/ubuntu-cross-aarch32-lldb

Thanks for reviews and input to: Lang Hames, Peter Smith, Pavel Labath and Kristof Beyls

Special thanks to Eymen Ünay for your GSoC proposal