Contents

List of Figures

Acronyms

SIHFT Software Implemented Hardware Fault Tolerance

Chapter 1

Installation Guide

In this chapter, we provide instructions on how to integrate our developed SIHFT passes within the LLVM codebase. This allows us later to carry out various SIHFT transformations during LLVM backend compilation in generating RISCV code.

1.1 Development System

The instructions given in this user manual are expected to work on any Linux machine. Specifically, we have prepared this manual while developing on a freshly installed Ubuntu 20.04 PC.

1.1.1 Ubuntu Packages

Following packages are needed to be installed:

- C++ compiler like gcc/g++
- CMake
- git

Following commands can be used to install them:

```
$ sudo apt install build-essential cmake git
```

1.2 Build LLVM Project

The LLVM Project has to be compiled from source. We recommend downloading version 13.0.0 from LLVM Releases webpage:

https://releases.llvm.org/

With the downloaded source package extracted, execute the following commands on your bash shell to build llvm project from source:

```
$ cd <llvm_src_folder>
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release
        -DCMAKE_INSTALL_PREFIX=<install_location>
        -DCMAKE_CXX_STANDARD=17
        -DLLVM_ENABLE_PROJECTS=clang
        -DLLVM_TARGETS_TO_BUILD=RISCV
        -DLLVM_ENABLE_ASSERTIONS=ON
        ../llvm
$ make
```

NOTE: The paths mentioned above as <...> need to be adapted as per your filesystem.

In the `<build>/bin` directory, we should now see llvm executable programs and utilities like `clang`, `llc` etc. Run them to make sure they are built fine. For example, we can run the static compiler `llc` to make sure that RISCV targets are supported as shown in Fig. 1.1.

NOTE: Make using parallel jobs (`make -jN`) would help significantly to reduce the compile time for a large codebase like LLVM. However, we encourage to do this on a server. For systems having lesser memory, make with parallel jobs has to be used cautiously, as the command could fail due to memory exhaustion issues.

```
$ ./llc --version
LLVM (http://llvm.org/):
  LLVM version 13.0.0
  Optimized build with assertions.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: skylake

  Registered Targets:
    riscv32 - 32-bit RISC-V
    riscv64 - 64-bit RISC-V
$
```

Figure 1.1: Expected output of sanely built `llc`

1.3 Integrate SIHFT Passes

The first step is to clone our open-source `compas-ft-riscv` repository within the LLVM source tree. Afterwards, the patch is applied on RISCV backend in order to register the new transformation passes. Finally, the LLVM project is rebuilt.
The corresponding shell commands are shown below. Here `<path_to_patch>` variable should be substituted with a compatible patch file. For this, we provide various patches in our passes repository. Each patch is compatible with a specific llvm project source tree. In case this tutorial is followed to build llvm project using Sec. 1.2, the corresponding patch is to be found at:

`compas-ft-riscv/patches/llvm13.0.0_src.patch`

```
$ cd <llvm_src_folder>/llvm/lib/Target/RISCV/
$ git clone https://github.com/tum-ei-eda/compas-ft-riscv
$ cd <llvm_src_folder>
$ patch -s -p0 < <path_to_patch>
$ cd build
```

```
$ make
$ make install
```

Here, at the end we install the built llvm compiler infrastructure. Again make sure that the compiler is installed fine by using Fig. 1.1.

Chapter 2

Usage Guide

Various SIHFT passes can be invoked via command line arguments to the LLVM static compiler `llc` program. This chapter assumes that the LLVM compiler has been installed as per instructions in Chap. **??**. We refer the installation location of this compiler to `<llvm_install>` in rest of the chapter.

2.1 SIHFT Compilation

Each source file (translation unit) in C/C++ project has to be first compiled to LLVM IR code using `clang`. Following command can be used to generate human readable LLVM IR code representation of given C source file.

```
$ <llvm_install>/bin/clang
    -emit-llvm -S <other options>
    main.c -o main.ll
```

The next step in the compilation is to pass the LLVM IR file to `llc` program:

```
$ <llvm_install>/bin/llc
    <SIHFT options> <other options>
    main.ll -o main_hardened.s
```

Table 2.1: Supported SIHFT options

Option	Description
-NZDC=<foo, bar>	apply NZDC transformation on *foo,bar* functions
-RASM=<foo, bar>	apply RASM transformation on *foo,bar* functions
-CFCSS=<foo, bar>	apply CFCSS transformation on *foo,bar* functions
-FGS	use fine-grain scheduling for NZDC code
-REPAIR	use REPAIR transformation on NZDC code

`<SIHFT options>` refer to various SIHFT transformations that we have developed. A brief overview of these options is provided in Table 2.1.
The final step in code generation is to use `gcc` for RISCV architectures to assemble and link the assembly code for a particular RISCV processor.

2.1.1 Example

For the sake of example, assume we want to protect the `crc` program from the MiBench suite. The compiled program is to run on the RISCV spike simulator. The C project contains the following C files:

- crc.h: header for CRC library functions
- crc.c: source for CRC library functions
- main.c: contains the `main` program to invoke crc library functions

Following commands are issued on bash shell to create the RISCV binary with

- NZDC, RASM protection on `crcSlow` function
- RASM protection on `crcFast` function
- NZDC, CFCSS protection on `main` function

```
$ cd <crc-project-source>

$ <clang> -emit-llvm -O2 -S --target=riscv64
    -march=rv64gc -mabi=lp64d
    -isystem <riscv64-gcc>/riscv64-unknown-elf/
    main.c -o main.ll
$ <clang> -emit-llvm -O2 -S --target=riscv64
    -march=rv64gc -mabi=lp64d
    -isystem <riscv64-gcc>/riscv64-unknown-elf/
    crc.c -o crc.ll

$ <llc> -O2 -march=riscv64 -mattr=+m,+a,+d,+c
    -NZDC=main -CFCSS=main
    main.ll -o main_hardened.s
$ <llc> -O2 -march=riscv64 -mattr=+m,+a,+d,+c
    -NZDC=crcSlow -RASM=crcSlow,crcFast
    crc.ll -o crc_hardened.s

$ <riscv64-gcc>/bin/riscv64-unknown-elf-gcc -O2
    -march=rv64gc -mabi=lp64d
    main_hardened.s crc_hardened.s -o crc_hardened.elf

$ spike -pk crc_hardened.elf
```

2.2 Configuration via TOML script