Qualcomm Technologies, Inc.

Snapdragon ARM LLVM Compiler for Android

User Guide

80-VB419-90 Rev. K

March 14, 2016

Questions or comments: developer.qualcomm.com/llvm-forum

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Tables

1 Introduction

1.1 Overview

This document describes C and C++ compilers for the ARM® processor architecture. The compilers are based on the LLVM compiler framework, and are collectively referred to as the LLVM compilers.

NOTE The LLVM compilers are commonly referred to as *Clang*.

It is highly recommended to try using the various LLVM code optimizations to improve the performance of your program. Using just the default optimization settings is likely to result in suboptimal performance.

1.2 Features

The LLVM compilers offer the following features:

- **ISO C conformance**
  Supports the International Standards Organization (ISO) C language standard
- **Compatibility**
  Supports LLVM extensions and most GCC extensions to simplify porting
- **System library**
  Supports standard libraries as provided in the Android NDK
- **Processor-specific libraries**
  Provides library routines that are optimized for the Qualcomm ARM architecture
- **Intrinsics**
  Provides a mechanism for emitting LLVM assembly instructions in C source code

1.3 Languages

The LLVM compilers support C, C++, and many dialects of those languages:

- C language: K&R C, ANSI C89, ISO C90, ISO C94 (C89+AMD1),
  ISO C99 (+TC1, TC2, TC3)
- C++ language: C++98, C++11

In addition to these base languages and their dialects, the LLVM compilers support a broad variety of language extensions. These extensions are provided for compatibility with the GCC, Microsoft, and other popular compilers, as well as to improve functionality through the addition of extensions unique to the LLVM compilers.

All language extensions are explicitly recognized as such by the LLVM compilers, and marked with extension diagnostics which can be mapped to warnings, errors, or simply ignored.

1.4 GCC compatibility

The LLVM compiler driver and language features are intentionally designed to be as compatible with the GNU GCC compiler as reasonably possible, easing migration from GCC to LLVM. In most cases, code "just works".

1.5 Processor versions

The LLVM compilers can generate code for all versions of the ARM processor architecture that are supported by the standard LLVM compiler.

However, full support is provided only for ARMv7 (which includes Krait) and ARMv8 (which is ARM's newest architecture).

ARMv8 supports two instruction set architectures (ISA):

- **AArch64** – the new 64-bit ISA, which supports a larger virtual and physical address space. All general purpose registers (and many of the system registers) are 64 bits. All instructions are encoded in 32 bits.
- **AArch32** – a 32-bit ISA which incorporates the ARMv7 ISA for both ARM and Thumb modes, and also includes many aspects of AArch64 (including support for cryptography and enhanced floating point).

For more information see the *ARMv8-A Reference Manual*.

1.6 LLVM versions

The LLVM compilers are based on LLVM 3.8, as defined at llvm.org.

1.7 Using the document

This document is designed as a reference for experienced C/C++ programmers. It describes the LLVM compilers and language implementations.

The document contains the following chapters:

- Chapter 1, *Introduction*, presents an overview of the compilers and the document.
- Chapter 2, *Getting Started*, explains how to compile and execute a simple C program.
- Chapter 3, *Using the Compilers*, describes the command line syntax, console messages, and input and output files.
- Chapter 4, *Code Optimization*, describes features for improving the size and speed of program code.
- Chapter 5, *Compiler Security Tools*, describes tools and features for improving the security and reliability of program code.
- Chapter 6, *Porting Code from GCC*, describes issues commonly encountered while porting GCC code to LLVM LLVM.
- Chapter 7, *Coding Practices*, describes recommended coding practices for ensuring the generation of efficient object code.
- Chapter 8, *Language Compatibility*, describes how the compilers implement the C language standard.
- Appendix A presents the LLVM license statements governing this document.

C language reference

This document does not describe the C or C++ languages. The suggested references are:

- *The C Programming Language (2nd Edition),* Brian Kernighan and Dennis Ritchie, Prentice Hall, 1988.
- *The C++ Programming Language (3rd Edition)*, Bjarne Stroustrup, Addison-Wesley, 1997.

Compiler references

This document does not provide detailed descriptions of the code optimizations performed by LLVM. Suggested compiler references are:

- *Compilers: Principles, Techniques, and Tools (2nd Edition)*, Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman, Prentice Hall, 2006
- *Engineering a Compiler (2nd Edition)*, Keith Cooper and Linda Torczon, Morgan Kaufmann, 2011

1.8 Notation

This document uses italics for terms and document names:

The C Programming Language (2nd Edition)

Courier font is used for computer text:

```
int main()
{
   printf("Hello world\n");
   return(0);
}
```

The following notation is used to define the syntax of functions and commands:

- Square brackets enclose optional items (e.g., `help [command]`).
- **Bold** is used to indicate literal symbols (e.g., the brackets in `array[index]`).
- The vertical bar character | is used to indicate a choice of items.
- Parentheses are used to enclose a choice of items (e.g., `(on|off)`).
- An ellipsis, `...`, follows items that can appear more than once.
- *Italics* are used for terms that represent categories of symbols.

Examples:

```
#define name(parameter1[, parameter2...]) definition
logging (on|off)
```

In the above examples `#define` is a preprocessor directive and `logging` is an interactive compiler command.

name represents the name of a defined symbol.

parameter1 and *parameter2* are macro parameters. The second parameter is optional since it is enclosed in square brackets. The ellipsis indicates that the macro accepts more than parameters.

`on` and **`off`** are bold to show that they are literal symbols. The vertical bar between them shows that they are alternative parameters of the `logging` command.

1.9 Feedback

If you have any comments or suggestions regarding the LLVM compilers (or this document), please send them to:

developer.qualcomm.com/llvm-forum

NOTE If you are a commercial licensee of Qualcomm, use your normal support channels for support.

2 Getting Started

2.1 Overview

This chapter shows how to build and execute a simple C program using the LLVM compiler.

The program is built in the Linux environment, and executed directly on ARMv7 or ARMv8 hardware running Linux.

NOTE The Android NDK is assumed to be already installed on your computer. This includes the tools required for assembling and linking a compiled program.

The commands shown in this chapter are for illustration only – for detailed information on building programs see Chapter 3.

2.2 Create source file

Create the following C source file:

```
#include <stdio.h>

int main()
{
     printf(“Hello world\n”);
     return(0);
}
```

Save the file as `hello.c`.

2.3 Compile program

Compile the program with the following command:

```
clang hello.c -o hello
```

This translates the C source file `hello.c` into the executable file `hello`.

2.4 Execute program

To execute the program, use the following command:

```
hello
```

The program outputs its message in the terminal:

```
Hello world
```

You have now compiled and executed a C program using the LLVM compiler. For more information on using the compiler see the following chapter.

3 Using the Compilers

3.1 Overview

The LLVM compilers translate C and C++ programs into LLVM processor code.

C and C++ programs are stored in source files, which are text files created with a text editor. LLVM processor code is stored in object files, which are executable binary files.

This chapter covers the following topics:

- Starting the compilers
- Input and output files
- Compiler options
- Warning and error messages
- Using GCC cross compile environments
- Built-in functions
- Compilation phases

3.2 Starting the compilers

To start the C compiler from a command line, type:

```
clang [options...] input_files...
```

To start the C++ compiler from a command line, type:

```
clang++ [options...] input_files...
```

The compilers accept one or more input files on the command line. Input files can be C/C++ source files or object files. For example:

```
clang hello.c mylib.c
```

Command switches are used to control various compiler options (Section 3.4). A switch consists of a dash character (‘`-`’) followed by a switch name and optional parameter.

Switches are case-sensitive and must be separated by at least one space. For example:

```
clang hello.c -o hello
```

To list the available command options, use the `--help` option:

```
clang --help
clang++ --help
```

This option causes the compiler to display the command line syntax, followed by a list of the available command options.

NOTE `clang` is the name of the front end driver for the LLVM compiler framework.

3.3 Input and output files

The LLVM compilers preprocess and compile one or more source files into object files. The compilers then invoke the linker to combine the object files into an executable file.

Table 3-1 lists the input file types and the tool that processes files of each type. The compilers use the file name extension to determine how to process the file.

Table 3-1 Compiler input files

Extension	Description	Tool
.c	C source file	C compiler
.i	C preprocessed file	
.h	C header file	
.cc .cp .cxx .cpp .CPP .c++ .C	C++ source file	C++ compiler
.ii	C++ preprocessed file	
.h .hh .H	C++ header file	
.bc .ll	LLVM intermediate representation (IR) file	C/C++ compiler
.s .S	Assembly source file	Assembler
other	Binary object file	Linker

NOTE All file name extensions are case-sensitive literal strings. Input files with unrecognized extensions are treated as object files.

For more information on LLVM IR files see llvm.org.

Table 3-2 lists the output file types and the tools used to generate each file type.

Compiler options (Section 3.4) are used to specify the output file type.

Table 3-2 Compiler output files

File Type	Default File Name	Input Files
Executable file	`a.out`	The specified source files are compiled and linked to a single executable file.
Object file	*`file`*`.o`	Each specified source file is compiled to a separate object file (where *file* is the source file name).
Assembly source file	*`file`*`.s`	Each specified source file is compiled to a separate assembly source file (where *file* is the source file name).
Preprocessed C/C++ source file	`stdout`	The preprocessor output is written to the standard output.

3.4 Compiler options

The LLVM compilers can be controlled by command-line options (Section 3.2). Many of the GCC options are supported, along with options that are LLVM-specific.

NOTE Many of the `-f`, `-m`, and `-W` options can be written in two ways: `-f<`*`option`*`>` to enable a binary option, or `-fno-<`*`option`*`>` to disable the option.

`-mllvm` is not a stand-alone option, but rather a standard prefix that appears in many LLVM-specific option names.

Display

See Section 3.4.1

```
-help
-v
```

Compilation

See Section 3.4.2

```
-###
-c  -cc1  -ccc-print-phases
-E  -S  -pipe
-o file
-Wp,arg[,arg...]
-Wa,arg[,arg...]
-Wl,arg[,arg...]
-x language
-Xclang arg
-no-canonical-prefixes
```

C dialect

See Section 3.4.3

```
-ansi  -fno-asm  -fblocks  -fgnu-runtime  -fgnu89-inline
-fsigned-bitfields  -fsigned-char  -funsigned-char
-no-integrated-cpp  -std=(c89|gnu89|c94|c99|gnu99)
-traditional  -Wpointer-sign
```

C++ dialect

See Section 3.4.4

```
-cxx-isystem dir
-ffor-scope  -fno-for-scope  -fno-gnu-keywords
-ftemplate-depth-n  -fvisibility-inlines-hidden
-fuse-cxa-atexit  -nobuiltininc  -nostdinc++
-Wc++0x-compat -Wno-deprecated
-Wnon-virtual-dtor -Woverloaded-virtual
-Wreorder
```

Warning and error messages

See Section 3.4.5

```
-ferror-limit=n  -ftemplate-backtrace-limit=n
-ferror-warn filename  -fsyntax-only  -pedantic
-pedantic-errors  -Q-unused-arguments
-w -Wfoo -Wno-foo  -Wall  -Warray-bounds
-Wcast-align  -Wchar-subscripts
-Wcomment  -Wconversion
-Wdeclaration-after-statement  -Wno-deprecated-declarations
-Wempty-body  -Wendif-labels  -Werror
-Werror=foo  -Wno-error=foo
-Werror-implicit-function-declaration
-Weverything  -Wextra  -Wfloat-equal
-Wformat  -Wformat=2  -Wno-format-extra-args
-Wformat-nonliteral  -Wformat-security
-Wignored-qualifiers
-Wimplicit  -Wimplicit-function-declaration  -Wimplicit-int
-Wno-invalid-offsetof  -Wlong-long  -Wmain
-Wmissing-braces -Wmissing-declarations
-Wmissing-noreturn  -Wmissing-prototypes  -Wno-multichar
-Wnonnull -Wpacked  -Wpadded  -Wparentheses  -Wpedantic
-Wpointer-arith -Wreturn-type  -Wshadow  -Wsign-compare
-Wswitch -Wswitch-enum  -Wsystem-headers
-Wtrigraphs  -Wundef  -Wuninitialized  -Wunknown-pragmas
-Wunreachable-code  -Wunused  -Wunused-function  -Wunused-label
-Wunused-parameter  -Wunused-value  -Wunused-variable
-Wno-vectorizer-no-neon  -Wwrite-strings
```

Debugging

See Section 3.4.6

```
-dumpmachine -dumpversion
-feliminate-unused-debug-symbols
-ftime-report
-g[level]  -gline-tables-only
-print-diagnostic-categories
-print-file-name=library  -print-libgcc-file-name
-print-multi-directory -print-multi-lib
-print-multi-os-directory  -print-prog-name=program
-print-search-dirs
-save-temps  -time
```

Diagnostic format

See Section 3.4.7

```
-fcaret-diagnostics  -fno-caret-diagnostics
-fdiagnostics-format=(clang|msvc|vi)
-fdiagnostics-show-option  -fno-diagnostics-show-option
-fdiagnostics-show-category=(none|id|name)
-fdiagnostics-print-source-range-info
-fno-diagnostics-print-source-range-info
-fdiagnostics-parseable-fixits
-fdiagnostics-show-note-include-stack
-fdiagnostics-show-template-tree
-fmessage-length=n
```

Individual warning groups

See Section 3.4.8

```
-Wextra-tokens  -Wambiguous-member-template
-Wbind-to-temporary-copy
```

Compiler crash diagnostics

See Section 3.4.9

```
-fno-crash-diagnostics
```

Linker

See Section 3.4.10

```
-fuse-ld=(gold|bfd|qcld)
```

Preprocessor

See Section 3.4.11

```
-A pred=ans  -A -pred=ans  -ansi  -C  -CC  -d(DMNU)
-D name  -D name=definition  -fexec-charset=charset
-finput-charset=charset  -fpch-deps  -fpreprocessed
-fstrict-overflow  -ftabstop=width  -fwide-exec-charset=charset
-fworking-directory  --help  -H  -I dir  -I-  -include file
-isystem prefix  -isystem-prefix prefix
-ino-system-prefix prefix
-M  -MD  -MF file  -MG  -MM  -MMD  -MP  -MQ target  -MT target
-nostdinc  -nostdinc++  -o file  -P  -remap  --target-help
-U name  -v  -version  --version  -w  -Wall  -Wcomment
-Wcomments  -Wendif-labels  -Werror  -Wimport
-Wsystem-headers  -Wtrigraphs  -Wundef  -Wunused-macros
-Xpreprocessor option
```

Assembling

See Section 3.4.12

```
-Xassembler option
-integrated-as  -no-integrated-as
```

Linking

See Section 3.4.13

```
object_file_name  -c  -dynamic  -E
-l library  -moslib=library
-nodefaultlibs  -nostartfiles  -nostdlib
-pie  -s  -S  -shared  -shared-libgcc
-static  -static-libgcc
-symbolic  -u symbol  -Xlinker option
```

Directory search

See Section 3.4.14

```
-Bprefix
-F dir  -I dir
--gcc-toolchain=prefix
-I-
-Ldir
--sysroot=prefix
```

Processor version

See Section 3.4.15

```
-target triple
-march=version
-mcpu=version
-mfpu=version
-mfloat-abi=(soft|softfp|hard)
```

Code generation

See Section 3.4.16

```
-fasynchronous-unwind-tables
-fchar-array-precise-tbaa  -fno-char-array-precise-tbaa
-femit-all-data  -femit-all-decls
-ffp-contract=(fast|on|off)
-fno-exceptions
-fmerge-functions
-fpic  -fPIC  -fpie  -fPIE
-fsanitize=address  -fno-sanitize=address
-fsanitize=memory  -fno-sanitize=memory
-fsanitize=event[,event...]  -fno-sanitize=event[,event...]
-fsanitize-blacklist=file  -fno-sanitize-blacklist
-fsanitize-messages  -fno-sanitize-messages
-fsanitize-opt-size  -fno-sanitize-opt-size
-fsanitize-source-loc  -fno-sanitize-source-loc
-fsanitize-use-embedded-rt
-fsanitize-memory-track-origins[=level]
-fshort-enums  -fno-short-enums
-fshort-wchar  -fshort-wchar
-ftrap-function=value  -ftrapv  -ftrapv-handler
-funwind-tables  -fverbose-asm
-fvisibility=[default|internal|hidden|protected]
-fwrapv
-mhwdiv=(arm|thumb|arm,thumb|none)
-mllvm -aarch64-disable-abs-reloc
-mllvm -aggressive-jt
-mllvm -arm-expand-memcpy-runtime
-mllvm -arm-memset-size-threshold
-mllvm -arm-memset-size-threshold-zeroval
-mllvm -arm-opt-memcpy
-mllvm -disable-thumb-scale-addressing
-mllvm -emit-cp-at-end
-mllvm -enable-android-compat
-mllvm -enable-arm-addressing-opt
-mllvm -enable-arm-peephole
-mllvm -enable-arm-zext-opt
-mllvm -enable-print-fp-zero-alias
-mllvm -enable-round-robin-RA
-mllvm -enable-select-to-intrinsics
-mllvm -favor-r0-7
-mllvm -force-div-attr
-mllvm -prefetch-locality-policy=(L1|L2|L3|stream)
-mrestrict-it  -mno-restrict-it
```

Vectorization

See Section 3.4.17

```
-fvectorize-loops  -ftree-vectorize
-fvectorize-loops-debug
-fprefetch-loop-arrays[=stride]  -fno-prefetch-loop-arrays
```

Parallelization

See Section 3.4.18

```
-fparallel
-fparallel-symphony
```

Optimization

See Section 3.4.19

```
-O  -O0 -O1 -O2 -O3 -O4 -Os -Oz
-Ofast  -Osize
```

Specific optimizations

See Section 3.4.20

```
-falign-functions[=n] -falign-jumps[=n]
-falign-labels[=n] -falign-loops[=n]
-falign-inner-loops  -fno-align-inner-loops
-falign-os  -fno-align-os
-fdata-sections  -ffunction-sections
-finline  -finline-functions
-floop-pragma  -fnomerge-all-constants
-fomit-frame-pointer  -foptimize-sibling-calls
-fstack-protector  -fstack-protector-all
-fstack-protector-strong  -fstrict-aliasing
-funit-at-a-time  -funroll-all-loops
-funroll-loops  -fno-zero-initialized-in-bss
--param ssp-buffer-size=size
```

Math optimization

See Section 3.4.21

```
-fassociative-math  -ffast-math  -ffinite-math-only
-fmath-errno  -fno-math-errno  -freciprocal-math
-fno-signed-zeros  -fno-trapping-math
-funsafe-math-optimizations
```

Link-time optimization

See Section 3.4.22

```
-flto
```

Profile-guided optimization

See Section 3.4.23

```
-fprofile-instr-generate[=filename]
-fprofile-instr-use=filename
-fprofile-sample-use=filename
```

Optimization reports

See Section 3.4.24

```
-fopt-reporter=(vectorizer|parallelizer|all)
-polly-max-pointer-aliasing-checks
-Rpass=loop-opt
-Rpass-missed=loop-opt
```

Compiler security

See Section 3.4.25

```
--analyze  -analyzer-checker=checker  -analyzer-checker-help
-analyzer-disable-checker=checker  --analyzer-output html
--analyzer-Werror  --compile-and-analyze dir
-ffcfi  -fno-fcfi
```

3.4.1 Display

`-help`
Display compiler command and option summary.

`-v`
Display compiler release version.

3.4.2 Compilation

`-###`
Print commands used to perform the compilation.

`-c`
Compile source file, but do not link it.

`-cc1`
Bypass the compiler driver and go directly to LLVM.

`-ccc-print-phases`
Print the compilation stages as they occur.

`-E`
Preprocess source file only, do not compile it.

`-S`
Compile source file, but do not assemble it.

`-pipe`
Communicate between compiler stages using pipes not temporary files.

`-o` *`file`*
Specify the name of the compiler output file.

`-Wp,`*`arg`*`[,`*`arg`*`...]`
Pass the specified arguments to the preprocessor.

`-Wa,`*`arg`*`[,`*`arg`*`...]`
Pass the specified arguments to the assembler.

`-Wl,``*arg*[,*arg*...]`
Pass the specified arguments to the linker.

`-x` *`language`*
Specify language of the subsequent source files specified on the command line.

`-Xclang` *`arg`*
Pass the specified argument to the compiler.

`-no-canonical-prefixes`
When processing a pathname:

- Do not expand any symbolic links.
- Do not resolve any references to “`/./`” or “`/../`”.
- Do not make relative prefixes absolute.

3.4.3 C dialect

`-ansi`
For C, support ISO C90. For C++, remove conflicting GNU extensions.

`-fno-asm`
Do not recognize `asm`, `inline`, or `typeof` as keywords.

`-fblocks`
Enable the Apple “blocks” extension.

`-fgnu-runtime`
Generate output compatible with the standard GNU Objective-C runtime.

`-fgnu89-inline`
Use the gnu89 inline semantics.

`-fsigned-bitfields`
Define bitfields as signed.

`-fsigned-char`
Define `char` type as signed.

`-funsigned-char`
Define `char` type as unsigned.

`-no-integrated-cpp`
Compile using separate preprocessing and compilation stages.

`-std=``(`**`c89`**`|`**`gnu89`**`|`**`c94`**`|`**`c99`**`|`**`gnu99`**`|`**`c11`**`)`
LLVM language mode. The default setting is `gnu99`.

`-traditional`
Support pre-standard C language.

`-Wpointer-sign`
Flag pointers when assigned or passed values with a differing sign.

3.4.4 C++ dialect

`-cxx-isystem` *`dir`*
Add specified directory to C++ SYSTEM include search path.

`-ffor-scope`
`-fno-for-scope`
Control whether the scope of a variable declared in a `for` statement is limited to the statement or to the scope enclosing the statement.

`-fno-gnu-keywords`
Disable recognizing `typeof` as a keyword.

`-ftemplate-depth-`*`n`*
Specify the maximum instantiation depth of a template class.

`-fvisibility-inlines-hidden`
Specify default visibility for inline C++ member functions.

`-fuse-cxa-atexit`
Register destructors with function `__cxa_atexit` (instead of `atexit`). This applies only to objects that have static storage duration.

`-nobuiltininc`
Disable builtin #include directories.

`-nostdinc++`
Disable standard #include directories for the C++ standard library.

`-Wc++0x-compat`
Generate warnings for C++ constructs with different semantics in ISO C++ 1998 and ISO C++ 200x.

`-Wno-deprecated`
Do not generate warnings when deprecated features are used.

`-Wnon-virtual-dtor`
Generate warning when a polymorphic class is declared with a non-virtual destructor.

`-Woverloaded-virtual`
Generate warning when a function hides virtual functions from a base class.

`-Wreorder`
Generate warning when member initializers do not appear in the code in the required execution order.

3.4.5 Warning and error messages

`-ferror-limit=`*`n`*
Stop emitting diagnostics after *`n`* errors have been produced. The default setting is 20. The error limit can be disabled with the option `-ferror-limit=0`.

`-ftemplate-backtrace-limit=`*`n`*
Only emit up to *`n`* template instantiation notes within the template instantiation backtrace for a single warning or error. The default setting is 10. The limit can be disabled with the option `-ftemplate-backtrace-limit=0`.

-ferror-warn *filename*

Convert the specified set of compiler warnings into errors.

The specified text file contains a list of warning names, with each warning name separated by whitespace in the file.

Warning names are based on the switch names of the corresponding compiler warning-message options. For example, to convert the warnings generated by the option `-Wunused-variable`, use the warning name `unused-variable`.

This option can be specified multiple times.

NOTE This option (and its associated file) can be integrated into a build system, and used to iteratively resolve the warning messages generated by a project.

-fsyntax-only

Check for syntax errors only.

-pedantic
-Wpedantic

Generate all warnings required by the ISO C and ISO C++ standards.

-pedantic-errors

Equivalent to `-pedantic`, but generate errors instead of warnings.

-Qunused-arguments

Do not generate warnings for unused driver arguments.

-w

Suppress all warnings.

-W*foo*

Enable the diagnostic*foo*.

-Wno-*foo*

Disable the diagnostic*foo*.

-Wall

Enable all `-W` options.

-Warray-bounds

Generate warning if array subscripts are out of bounds.

-Wcast-align

Generate warning if a pointer cast increases the required alignment of the target.

-Wchar-subscripts

Generate warning if array subscript is type `char`.

-Wcomment

Generate warning if a comment symbol appears inside a comment.

-Wconversion

Generate warning if an implicit conversion may alter a value.

-Wdeclaration-after-statement

Generate warning when a declaration appears in a block after a statement.

-Wno-deprecated-declarations
Do not generate warnings for functions, variables, or types assigned the attribute `deprecated`.

-Wempty-body
Generate warning if an `if`, `else`, or `do while` statement contains an empty body.

-Wendif-labels
Generate warning if an `#else` or `#endif` directive is followed by text.

-Werror
Convert all warnings into errors.

-Werror=*foo*
Convert the diagnostic *foo* into an error.

-Wno-error=*foo*
Keep the diagnostic *foo* as a warning, even if `-Werror` is used.

-Werror-implicit-function-declaration
Generate warning or error if a function is used before being declared.

-Weverything
Enable all warnings.

-Wextra
Enable selected warning options, and generate warnings for selected events.

-Wfloat-equal
Generate warning if two floating point values are compared for equality.

-Wformat
In calls to `printf`, `scanf`, and other functions with format strings, ensure that the arguments are compatible with the specified format string.

-Wformat=2
This option is equivalent to specifying the following options: "`-Wformat -Wformat-nonliteral -Wformat-security -Wformat-y2k`".

-Wno-format-extra-args
Do not generate warning for passing extra arguments to `printf` or `scanf`.

-Wformat-nonliteral
Generate warning if the format string is not a string literal, except if the format arguments are passed through `va_list`.

-Wformat-security
Generate warning for format function calls that may cause security risks.

-Wignored-qualifiers
Generate warning if a return type has a qualifier (for example, `const`).

-Wimplicit
Equivalent to `-Wimplicit-int` and `-Wimplicit-function-declaration`.

-Wimplicit-function-declaration
Generate warning if a function is used before it is declared.

-Wimplicit-int
Generate warning if a declaration does not specify a type.

-Wno-invalid-offsetof
Do not generate warning if macro `offsetof` is passed a non-POD type.

-Wlong-long
Generate warning if type`long long` is used.

-Wmain
Generate warning if the function `main()` has any suspicious properties.

-Wmissing-braces
Generate warning if an aggregate or union initializer is not properly bracketed.

-Wmissing-declarations
Generate warning if a global function is defined without being first declared.

-Wmissing-noreturn
Generate warning if a function does not include a `return` statement.

-Wmissing-prototypes
Generate warning if a global function is defined without a prototype.

-Wno-multichar
Do not generate warning if a multicharacter constant is used.

-Wnonnull
Generate warning if a null pointer is passed to an argument that is specified to require a non-null value (with the `nonnull` attribute).

-Wpacked
Generate warning if the memory layout of a structure is not affected after the structure is specified with the `packed` attribute.

-Wpadded
Generate warning if the memory layout of a structure includes padding.

-Wparentheses
Generate warning if the parentheses are omitted in certain cases.

-Wpedantic
See `-pedantic`.

-Wpointer-arith
Generate warning if any code depends on the size of `void` or a function type.

-Wreturn-type
Generate warning if a function returns a type that defaults to `int`, or a value incompatible with the defined return type.

-Wshadow
Generate warning if a local variable shadows another local variable, global variable, or parameter; or if a built-in function gets shadowed.

-Wsign-compare
Generate warning in a signed/unsigned compare if the result may be inaccurate due to the signed operand being converted to unsigned.

-Wswitch
-Wswitch-enum
Generate warning if a `switch` statement uses an enumeration type for the index, and does not specify a `case` for every possible enumeration value, or specifies a case with a value outside the enum range.

-Wsystem-headers
Generate warning for constructs declared in system header files.

-Wtrigraphs
Generate warning if a trigraph forms an escaped newline in a comment.

-Wundef
Generate warning if an undefined non-macro identifier appears in an `#if` directive.

-Wuninitialized
Generate warning if referencing an uninitialized automatic variable.

-Wunknown-pragmas
Generate warning if a `#pragma` directive is not recognized by the compiler.

-Wunreachable-code
Generate warning if code will never be executed.

-Wunused
Specifies all of the `-Wunused` options.

-Wunused-function
Generate warning if a static function is declared without being defined or used.

NOTE No warning is generated for functions declared or defined in header files.

-Wunused-label
Generate warning if a label is declared without being used.

-Wunused-parameter
Generate warning if a function argument is not used in its function.

-Wunused-value
Generate warning if the value of a statement is not subsequently used.

-Wunused-variable
Generate warning if a local or non-constant static variable is not used in its function.

-Wno-vectorizer-no-neon
Do not generate the warning “Vectorization flags ignored because armv7/armv8 and neon not set”.

Vectorization requires the target to be ARMv7 or ARMv8, and the NEON feature to be enabled. If the vectorization options are used without these required options, a warning is normally generated and the vectorization options are ignored.

-Wwrite-strings
For C, assign string constants the type `const char[`*length*`]` to ensure that a warning is generated if the string address gets copied to a non-`const char *` pointer. For C++, generate warning if converting a string constant to `char *`.

3.4.6 Debugging

`-dumpmachine`
Display the target machine name.

`-dumpversion`
Display the compiler version.

`-feliminate-unused-debug-symbols`
Generate debug information only for the symbols that are used. (Debug information is generated in STABS format.)

`-time`
`-ftime-report`
Display the elapsed time for each stage of the compilation.

`-g``[`*`level`*`]`
Generate complete source-level debug information.

`-gline-tables-only`
Generate source-level debug information with line number tables only.

`-print-diagnostic-categories`
Display mapping of diagnostic category names to category identifiers.

`-print-file-name=`*`library`*
Display the full library path of the specified file.

`-print-libgcc-file-name`
Display the library path for file `libgcc.a`.

`-print-multi-directory`
Display the directory names of the multi libraries specified by other compiler options in the current compilation.

`-print-multi-lib`
Display the directory names of the multi libraries paired with the compiler options that specified the libraries in the current compilation.

`-print-multi-os-directory`
Display the relative path that gets appended to the multilib search paths.

`-print-prog-name=`*`program`*
Display the absolute path of the specified program.

`-print-search-dirs`
Display the search paths used to locate libraries and programs during compilation.

`-save-temps`
Save the normally-temporary intermediate files generated during compilation.

3.4.7 Diagnostic format

The LLVM compilers aim to produce beautiful diagnostics by default, especially for new users just beginning to use LLVM. However, different users have different preferences, and sometimes LLVM may be driven by another program which needs the diagnostic output to be simple and consistent rather than user-friendly. For these cases, LLVM provides a wide range of options to control the output format of the diagnostics that it generates.

`-fcaret-diagnostics`
`-fno-caret-diagnostics`

Print source line and ranges from source code in diagnostic.

Control whether LLVM prints the source line, source ranges, and caret when emitting a diagnostic. The default setting is enabled. When enabled, LLVM will print something like:

```
test.c:28:8: warning: extra tokens at end of #endif directive
[-Wextra-tokens]
#endif bad
       ^
       //
```

`-fdiagnostics-format=(clang|msvc|vi)`

Change diagnostic output format to better match IDEs and command line tools.

This option controls the output format of the filename, line number, and column printed in diagnostic messages. The default setting is `clang`. The effect of the setting on the output format is shown below.

`clang`

```
t.c:3:11: warning: conversion specifies type 'char *' but the
argument has type 'int'
```

`msvc`

```
t.c(3,11) : warning: conversion specifies type 'char *' but the
argument has type 'int'
```

`vi`

```
t.c +3:11: warning: conversion specifies type 'char *' but the
argument has type 'int'
```

-fdiagnostics-show-option
-fno-diagnostics-show-option

Enable `[-W`*`option`*`]` information in diagnostic line.

Control whether LLVM prints the associated warning group option name (Section 3.5.3) when outputting a warning diagnostic. The default setting is disabled. For example, given the following diagnostic output:

```
test.c:28:8: warning: extra tokens at end of #endif directive
[-Wextra-tokens]
#endif bad
       ^
       //
```

In this case, specifying `-fno-diagnostics-show-option` prevents LLVM from printing the [`-Wextra-tokens`] information in the diagnostic output. This information indicates the option needed to enable or disable the diagnostic, either from the command line or by using the pragma `GCC diagnostic` (Section 3.5.5).

-fdiagnostics-show-category=(none|id|name)

Enable printing category information in diagnostic line.

This option controls whether LLVM prints the category associated with a diagnostic when emitting it. The default setting is `none`. The effect of the setting on the output format is shown below.

`none`

```
t.c:3:11: warning: conversion specifies type 'char *' but the
argument has type 'int' [-Wformat]
```

`id`

```
t.c:3:11: warning: conversion specifies type 'char *' but the
argument has type 'int' [-Wformat,1]
```

`name`

```
t.c:3:11: warning: conversion specifies type 'char *' but the
argument has type 'int' [-Wformat,Format String]
```

Each diagnostic may or may not have an associated category; if it has one, it is listed in the diagnostic category field of the diagnostic line (in the `[]`'s).

This option can be used to group diagnostics by category, so it should be a high-level category: the goal is get dozens of categories, not hundreds or thousands of them.

`-fdiagnostics-print-source-range-info`
`-fno-diagnostics-print-source-range-info`

Print machine-parseable information about source ranges.

This option controls whether LLVM prints information about source ranges in a machine-parseable format after the file/line/column number information. The default setting is disabled. The information is a simple sequence of brace-enclosed ranges, where each range lists the start and end line/column locations. For example, given the following output:

```
exprs.c:47:15:{47:8-47:14}{47:17-47:24}: error: invalid operands
to binary expression ('int *' and '_Complex float')
P = (P-42) + Gamma*4;
    ~~~~~~ ^ ~~~~~~~
```

In this case the {}'s are generated by `-fdiagnostics-print-source-range-info`.

The printed column numbers count bytes from the beginning of the line; take care if your source contains multibyte characters.

`-fdiagnostics-parseable-fixits`

Print Fix-Its in a machine-parseable format.

This option makes LLVM print available Fix-Its in a machine-parseable format at the end of diagnostics. The following example illustrates the format:

```
fix-it:"t.cpp":{7:25-7:29}:"Gamma"
```

In this case the range printed is half-open, so the characters from column 25 up to (but not including) column 29 on line 7 of file `t.cpp` should be replaced with the string `Gamma`. Either the range or replacement string can be empty (representing strict insertions and strict erasures, respectively). Both the file name and insertion string escape backslash (as "`\\`"), tabs (as "`\t`"), newlines (as "`\n`"), double quotes (as "`\"`"), and non-printable characters (as octal "`\xxx`").

The printed column numbers count bytes from the beginning of the line; take care if your source contains multibyte characters.

-fdiagnostics-show-template-tree

For large templated types, this option causes LLVM to display the templates as an indented text tree, with one argument per line, and any differences marked inline.

```
default
```

```
t.cc:4:5: note: candidate function not viable: no known conversion
from 'vector<map<[...], map<float, [...]>>>' to 'vector<map<[...],
map<double, [...]>>>' for 1st argument;
```

```
-fdiagnostics-show-template-tree
```

```
t.cc:4:5: note: candidate function not viable: no known conversion
for 1st argument;
  vector<
    map<
      [...],
      map<
        [float != float],
        [...]>>>
```

-fmessage-length=*n*

Format error messages to fit on lines with the specified number of characters.

3.4.8 Individual warning groups

-Wextra-tokens

Warn about excess tokens at the end of a preprocessor directive.

This option enables warnings about extra tokens at the end of preprocessor directives. The default setting is enabled. For example:

```
test.c:28:8: warning: extra tokens at end of #endif directive
[-Wextra-tokens]
#endif bad
       ^
```

These extra tokens are not strictly conforming, and are usually best handled by commenting them out.

-Wambiguous-member-template

Warn about unqualified uses of a member template whose name resolves to another template at the location of the use.

This option (which is enabled by default) generates a warning in the following code:

```
template<typename T> struct set{};
template<typename T> struct trait { typedef const T& type; };
struct Value {
  template<typename T> void set(typename trait<T>::type value){}
};

void foo() {
  Value v;
  v.set<double>(3.2);
}
```

C++ requires this to be an error, but because it is difficult to work around, LLVM downgrades it to a warning as an extension.

-Wbind-to-temporary-copy

Warn about an unusable copy constructor when binding a reference to a temporary.

This option enables warnings about binding a reference to a temporary when the temporary does not have a usable copy constructor. The default setting is enabled. For example:

```
struct NonCopyable {
  NonCopyable();
private:
  NonCopyable(const NonCopyable&);
};
void foo(const NonCopyable&);
void bar() {
  foo(NonCopyable());   // Disallowed in C++98; allowed in C++11.
}


struct NonCopyable2 {
  NonCopyable2();
  NonCopyable2(const NonCopyable2&);
};
void foo(const NonCopyable2&);
void bar() {
  foo(NonCopyable2());   // Disallowed in C++98; allowed in C++11.
}
```

NOTE If `NonCopyable2::NonCopyable2()` has a default argument whose instantiation produces a compile error, that error will still be a hard error in C++98 mode, even if this warning is disabled.

3.4.9 Compiler crash diagnostics

The LLVM compilers may crash once in a while. Generally, this only occurs when using the latest versions of LLVM.

LLVM goes to great lengths to assist you in filing a bug report. Specifically, after a crash it generates preprocessed source file(s) and associated run script(s). These files should be attached to a bug report to ease reproducibility of the failure. The following compiler option is used to control the crash diagnostics.

`-fno-crash-diagnostics`

Disable auto-generation of preprocessed source files during a LLVM crash.

This option can be helpful for speeding up the process of generating a delta reduced test case.

3.4.10 Linker

`-fuse-ld=``(`**`gold`**`|`**`bfd`**`|`**`qcld`**`)`

Specify an alternative linker to use in place of the default system linker.

Several mechanisms are provided for specifying the system linker that is used in the Snapdragon ARM LLVM toolchain:

- `-fuse-ld`: This option causes the toolchain to use the specified linker (see below for details).
- `--gcc-toolchain`: If `-fuse-ld` is not used, this option causes the toolchain to use whatever linker is found in the GCC toolchain option path.
- `--sysroot`: If neither `-fuse-ld` nor `--gcc-toolchain` are used, this option causes the toolchain to use whatever linker is found in the specified sysroot.
- If none of the above options are used, the toolchain uses the host linker by default. Note that this will result in errors during linking.

The `-fuse-ld` option can be used to specify the `gold`, `bfd`, or `qcld` linker as the system linker.

`gold` and `bfd` are typically included in the GNU GCC sysroots (version 4.7 and later). `gold` provides the plugin interface that is necessary to support link-time optimization (Section 4.7), while `bfd` does not.

`qcld` specifies the Snapdragon ARM LLVM linker. For more information see the *Snapdragon Arm LLVM Linker User Guide*.

NOTE When using link-time optimization, the default system linker changes to `qcld`. In this case either `qcld` or `gold` must be used as the system linker (otherwise, the optimization will fail).

For more information on sysroots see Section 3.6.

3.4.11 Preprocessor

`-A pred=`*`ans`*
Assert the predicate *`pred`* and answer *`ans`*.

`-A -pred=`*`ans`*
Cancel the specified assertion.

`-ansi`
Use C89 standard.

`-C`
Retain comments during preprocessing.

`-CC`
Retain comments during preprocessing, including during macro expansion.

`-d``(`**`DMNU`**`)`

D	Print macro definitions in -E mode in addition to normal output
M	Print macro definitions in -E mode instead of normal output
N	Print macro names in -E mode in addition to normal output
U	Print referenced macro definitions in -E mode in addition to normal output. Additionally print `#undef`s for macros that are undefined when referenced. Both are printed at the point they are referenced.

`-D` *`name`*
`-D name=`*`definition`*
Define the specified macro symbol.

`-fexec-charset=charset`
Specify the character set used to encode strings and character constants. The default character set is UTF-8.

`-finput-charset=charset`
Specify the character set used to encode the input files. The default is UTF-8.

`-fpch-deps`
Cause the dependency-output options to additionally list the files from a precompiled header's dependencies.

`-fpreprocessed`
Notify the preprocessor that the input file has already been preprocessed.

`-fstrict-overflow`
Enforce strict language semantics for pointer arithmetic and signed overflow.

`-ftabstop=`*`width`*
Specify the tab stop distance.

`-fwide-exec-charset=charset`
Specify the character set used to encode wide strings and character constants. The default character set is UTF-32 or UTF-16, depending on the size of `wchar_t`.

`-fworking-directory`
Generate line markers in the preprocessor output. The compiler uses this to determine what the current working directory was during preprocessing.

`--help`
Display the preprocessor release version.

`-H`
Display the header includes and nesting depth.

`-I` *`dir`*
Add the specified directory to the list of search directories for header files.

`-I-`
This option is deprecated.

`-include` *`file`*
Include the contents of the specified source file.

`-isystem` *`prefix`*
Treat an included file as a system header if it is found on the specified path (Section 3.5.6).

`-isystem-prefix` *`prefix`*
Treat an included file as a system header if it is found on the specified subpath of a defined include path (Section 3.5.6).

`-ino-system-prefix` *`prefix`*
Do not treat an included file as a system header if it is found on the specified subpath of a defined include path (Section 3.5.6).

`-M`
Output a `make` rule describing the dependencies of the main source file.

`-MD`
Equivalent to `-M -MF` *file*, except `-E` is not implied.

`-MF` *`file`*
Write dependencies to the specified file.

`-MG`
Add missing headers to the dependency list.

`-MM`
Equivalent to `-M`, except do not mention header files found in the system header directories.

`-MMD`
Equivalent to `-MD`, except only mention user header files, not system header files.

`-MP`
Create artificial target for each dependency.

`-MQ` *`target`*
Specify target to quote for dependency.

`-MT` *`target`*
Specify target for dependency.

`-nostdinc`
Omit searching for header files in the standard system directories.

`-nostdinc++`
Omit searching for header files in the C++-specific standard directories.

`-o` *`file`*
Specify the name of the preprocessor output file.

`-P`
Disable linemarker output when using `-E`.

`-remap`
Generate code for file systems that only support short file names.

`--target-help`
Display all command options and exit immediately.

`-traditional-cpp`
Emulate pre-standard C preprocessors.

`-trigraphs`
Preprocess trigraphs.

`-U` *`name`*
Cancel any previous definition of the specified macro symbol.

`-v`
Equivalent to `-help`.

`-version`
Display the preprocessor version during preprocessing.

`--version`
Display the preprocessor version and exit immediately.

`-w`
Suppress all preprocessor warnings.

`-Wall`
Enable all warnings.

`-Wcomment`
`-Wcomments`
Generate warning if a comment symbol appears inside a comment.

`-Wendif-labels`
Generate warning if an `#else` or `#endif` directive is followed by text.

`-Werror`
Convert all warnings into errors.

`-Wimport`
Generate warning when `#import` is used the first time.

`-Wsystem-headers`
Generate warning for constructs declared in system header files.

`-Wtrigraphs`
Generate warning if a trigraph forms an escaped newline in a comment.

`-Wundef`
Generate warning if an undefined non-macro identifier appears in an `#if` directive.

`-Wunused-macros`
Generate warning if a macro is defined without being used.

3.4.12 Assembling

`-integrated-as`
`-no-integrated-as`

Use the LLVM integrated assembler when compiling C and C++ source files.

`-no-integrated-as` explicitly disables the use of the integrated assembler.

By default, the integrated assembler is enabled.

NOTE If a program can potentially generate hardware divide instructions (`SDIV`, `UDIV`), it is strongly recommended to use the integrated assembler. Older GNU assemblers may not understand these instructions.

When directly assembling a `.s` source file, LLVM still invokes the external assembler because it cannot correctly translate all GNU assembly language constructions. As a result, not all GNU assembler options (which are passed with the `-Wa` option) will work with the integrated assembler.

The integrated assembler can process its own assembly-generated code, along with most hand-written assembly that conforms to the GNU assembly syntax.

`-Xassembler` *`arg`*

Pass the specified argument to the assembler.

3.4.13 Linking

Starting with the 3.7 LLVM release, Clang should be used as the driver for linking.

`object_file_name`

Linker input file.

`-c`

Do not perform linking. This option is used with spec strings.

`-dynamic`

Link with a shared library (instead of a static library).

`-E`

Do not perform linking. This option is used with spec strings.

`-l` *`library`*

Search the specified library file while linking.

`-moslib=`*`library`*

Search the RTOS-specific library named `lib`*library*`.a`. The search paths for the library and include files must be explicitly specified.

`-nodefaultlibs`

Do not use the standard system libraries when linking.

`-nostartfiles`

Do not use the standard system startup files when linking.

`-nostdlib`

Do not use the standard system startup files or libraries when linking.

`-pie`
: Generate a position-independent executable as the output file.

`-s`
: Delete all symbol table information and relocation information from the executable.

`-S`
: Do not perform linking. This option is used with spec strings.

`-shared`
: Generate a shared object as the output file. The resulting file can be subsequently linked with other object files to create an executable.

`-shared-libgcc`
: Link with the shared version of the library`libgcc`.

`-static`
: Do not link with the shared libraries. Only relevant when using dynamic libraries.

`-static-libgcc`
: Link with the static version of the library`libgcc`.

`-symbolic`
: Bind references to global symbols when building a shared object.

`-u` *`symbol`*
: Pretend the symbol *symbol* is undefined, to force linking of library modules to define it.

`-Xlinker` *`arg`*
: Pass the specified argument to the linker.

3.4.14 Directory search

`-B`*`prefix`*
: Specify the top-level directory of the compiler.

`-F` *`dir`*
: Add the specified directory to the search path for framework includes.

`--gcc-toolchain``=`*`prefix`*
: Equivalent to `-B` above.

`-I` *`dir`*
: Add the specified directory to the include file search path.

`-I-`
: This option is deprecated.

`-L`*`dir`*
: Add the specified directory to the list of directories searched by the `-l` option.

`--sysroot``=`*`prefix`*
: Specify the root directory of the system tools environment (Section 3.6).

3.4.15 Processor version

LLVM defines the options `-target`, `-march`, and `-mcpu` for specifying the ARM processor version to generate code for.

If none of these options are specified, the LLVM compilers by default generate code for the lowest ARMv4t instruction set architecture, in ARM mode, for the ARM7tdmi CPU.

If the ARMv7 or ARMv8 architecture is specified using the options `-march` or `-mcpu`, but ARM mode (i.e., 32-bit-only mode) is not specified on the command line, the LLVM compilers default to generating code in Thumb2 mode. To disable Thumb mode, use the options `-mno-thumb` or `-marm`.

`-target` *`triple`*

Specify the ARM architecture, operating system, and ABI for code generation.

The `triple` argument has the following format:

`arch-platform-abi`

For example, to generate code for the ARMv7a which runs on Linux and conforms to `gnueabi`, specify the following option:

```
clang -target armv7a-linux-gnueabi foo.c
```

The best way to specify the architecture version and CPU is by using the `-march` and `-mcpu` options respectively. Even though a target triple can be used to specify the architecture, it must match the GCC tools sysroot (Section 3.6). Thus, the above command can be alternately expressed as follows:

```
clang -target arm-linux-gnueabi -mcpu=cortex-a9 foo.c
```

... where `cortex-a9` indicates ARMv7a as the CPU.

Here are some commonly-used target triples:

`arm-linux-gnueabi`

`arm-none-linux-gnueabi` (equivalent to `arm-linux-gnueabi`)

`arm-linux-androideabi` (for code conforming to Android EABI)

`aarch64-linux-gnu` (for ARMv8 AArch64 mode)

`aarch64-linux-android` (for code conforming to Android EABI)

`armv8-linux-gnu` (for ARMv8 AArch32 mode)

`arm-none-eabi` (for ARM bare-metal executables)

NOTE In older versions of LLVM the `-target` option was named `-triple`.

-march=*version*

Specify the ARM architecture for code generation.

This option has the following possible values:

```
armv5e
armv6j
armv7
armv7-a
armv7-m
armv8
armv8-a
```

-mcpu=*version*

Specify the ARM CPU for code generation.

For a complete list of the values defined for this option, run the following command:

```
llvm-as | </dev/null | llc -march=arm -mcpu=help
```

Here are some commonly-used CPU values:

ARMv7:
```
cortex-a8
cortex-a9
cortex-a15
```

Qualcomm ARMv7:
```
scorpion
krait
```

ARMv8:
```
cortex-a53
cortex-a57
kryo
```

NOTE `-mcpu=krait2` is deprecated. Instead, use `-mcpu=krait`.

`-mcpu` automatically sets `-mfpu`.

-mfpu=*version*

Specify the ARM architecture extensions.

For a complete list of the values defined for this option, run the following command:

```
llvm-as | </dev/null | llc -march=arm -mcpu=help
```

Here are some commonly-used option values for `-mfpu`:

`neon`

Enable the NEON single instruction, multiple data (SIMD) architecture extension for the ARM Cortex-A (`cortex-a9`) or Qualcomm ARM v7 (`krait`) and ARMv8 processors.

`vfpv4`

Enable the VFPv4 architecture extensions. The VFPv4 extension enables code generation of the fused multiply add and subtract instructions (Section 3.4.16).

`neon-fp-armv8`

Enable NEON and ARMv8 FP extensions.

`crypto-neon-fp-armv8`

Enable Cryptography, NEON, and ARMv8 FP extensions.

Here are examples of valid `-mfpu` option values for ARM and AArch64:

```
vfp
vfpv2
vfpv3
vfpv3-fp16
vfpv3-d16
vfpv3-d16-fp16
vfpv3xd
vfpv3xd-fp16
vfpv4
vfpv4-d16
fpv4-sp-d16
fpv5-d16
fpv5-sp-d16
fp-armv8
neon
neon-fp16
neon-vfpv4
neon-fp-armv8
crypto-neon-fp-armv8
```

NOTE Using the -`mcpu` option automatically enables the default NEON and FP extensions for the specified CPU target. For example:

`-mcpu=krait`

Automatically enables the NEON and VFPv4 extensions.

`-mcpu=cortex-a9`

Automatically enables the NEON and VFPv3 extensions (including the half-precision extension).

`-mcpu=cortexa57`

Automatically enables the Cryptography, NEON, and ARMv8 FP extensions.

NOTE To disable a specific NEON or FP extension, use `-mcpu` along with `-mfpu`. But note that using `-mcpu`, `-march`, or `--target` with `-mfpu` will generate an error if the specified `-mfpu` option is invalid.

`-mfloat-abi=``(`**`soft`**`|`**`softfp`**`|`**`hard`**`)`
Specify the floating-point ABI.

NOTE ARMv8 mandates hardware floating point.

3.4.16 Code generation

`-fasynchronous-unwind-tables`

Generate unwind table. The table is stored in DWARF2 format.

`-fchar-array-precise-tbaa`
`-fno-char-array-precise-tbaa`

Prevent aliasing of `char` arrays by non-`char` pointers.

This option causes the compiler to assume that no pointer other than a pointer to `char` can reference an element in a `char` array.

The default is disabled.

NOTE **`-fchar-array-precise-tbaa`** is enabled by default at the `-Ofast` level.

In the example below, enabling **`-fchar-array-precise-tbaa`** results in the statement "`d = *p`" being hoisted out, because `p` is a pointer to `int`.

```
typedef struct {
  char a;
  char b[100];
  char c;
 } S;

int *p;
S x;

void func1 (char d) {
  for (int i = 0; i < 100; i++) {
    x.b[i] += 1;
    d = *p;
    x.a += d;
  }
}
```

`-femit-all-data`

Emit all data, even if unused.

`-femit-all-decls`

Emit all declarations, even if unused.

`-ffp-contract=(fast|on|off)`

Fused multiply add and subtract operations (VFMLA,VFMS) are more accurate than chained multiply add and subtract operations (VMLA, VMLS) because the chained operations perform rounding both after the multiply and before the add/subtract. While rounding itself introduces only a small error, cumulatively it can have a huge impact on the final result.

While fused operations are IEEE compliant, it is not IEEE compliant for the compiler to automatically replace a multiply followed by an add/subtract (or VMLA/VMLS) with the equivalent fused operation, since the numeric result can differ so much. However, if a programmer explicitly specifies the use of a fused operation, then the substitution is considered IEEE compliant.

Fused operations are explicitly specified with the `-ffp-contract` option. It has the following possible values:

`fast`

Enable fused operations throughout the program.

`on`

Enable fused operations according to the `FP_CONTRACT` pragma (default).

`off`

Disable fused operations throughout the program.

NOTE This option must be used with the `-mfpu=neon-vfpv4` option.

Enabling fused operations causes the compiler to relax IEEE compliance for floating point computation.

`-fno-exceptions`

Do not generate code for propagating exceptions.

`-finstrument-functions`

Generate instrumentation calls in function entries and exits.

`-fmerge-functions`
`-fno-merge-functions`

Attempt to merge functions that are equivalent, or differ by only a few instructions (Section 4.6). The default setting is disabled.

This option attempts to improve code size by merging similar functions. It uses a number of heuristics to determine whether it is worthwhile to merge a pair of functions. For instance, very small functions or functions with significant differences are usually not merged.

NOTE Because this option may have a negative impact on program performance, it is disabled by default, and becomes enabled only when it is specified explicitly.

`-fpic`

Generate position-independent code (PIC) for use in a shared library.

`-fPIC`

Generate position-independent code for dynamic linking, avoiding any limits on the size of the global offset table.

`-fpie`
`-fPIE`

Generate position-independent code (PIC) for linking into executables.

`-fsanitize=address`
`-fno-sanitize=address`

Generate instrumentation for the address sanitizer (Section 5.3).

`-fsanitize=memory`
`-fno-sanitize=memory`

Generate instrumentation for the memory sanitizer (Section 5.9).

`-fsanitize=``*event*[,*event*...]`
`-fno-sanitize=``*event*[,*event*...]`

Generate instrumentation for the undefined behavior sanitizer. One or more events can be specified.

This option accepts the following event values:

`alignment`

Misaligned pointers or creating a misaligned reference.

`bool`

Loading boolean values that are neither true nor false.

`bounds`

Out-of-bounds array indexes (when the bounds can be statically determined).

`enum`

Loading enum values that are out-of-range for an enum type.

`float-cast-overflow`

Floating-point conversion which would overflow the destination.

`float-divide-by-zero`

Floating-point division by zero.

`function`

Indirect function calls through a pointer of the wrong type (Linux and C++ only).

`integer-divide-by-zero`

Integer division by zero.

`nonnull-attribute`

Returning null pointer from a function declared to never return null.

`null`

Using a null pointer or creating a null reference.

`object-size`

Attempts to use bytes that the optimizer can determine are not part of the object being accessed. (Object sizes are determined with `__builtin_object_size`, so it may be possible to detect more problems at higher optimization levels.)

`return`

In C++, reaching the end of a value-returning function without returning a value.

`returns-nonnull-attribute`

Returning null pointer from a function declared to never return null.

`shift`

Shift operators where the amount shifted is less than zero, or greater than or equal to the promoted bit-width of the left hand side, or where the left hand side is negative. For a signed left shift, it also checks for signed overflow in C, and for unsigned overflow in C++.

`signed-integer-overflow`

Signed integer overflow, including all the checks added by `-ftrapv`, and checking for overflow in signed division (INT_MIN / -1).

`unreachable`

Program control flow reaches `__builtin_unreachable`.

`unsigned-integer-overflow`

Unsigned integer overflows.

`vla-bound`

Variable-length arrays whose bounds do not evaluate to a positive value.

`vptr`

Use of an object whose `vptr` indicates that it is of the wrong dynamic type, or that its lifetime has not begun or has ended. Incompatible with `-fno-rtti` and `-fsanitize-use-embedded-rt`.

NOTE Using this option requires user-defined diagnostic handler functions. For more information see the undefined behavior sanitizer (Section 5.11).

`-fsanitize=integer`

Generate instrumentation for the undefined behavior sanitizer for the following events (as defined above):

```
signed-integer-overflow
unsigned-integer-overflow
shift
integer-divide-by-zero
```

`-fsanitize=undefined`

Generate instrumentation for the undefined behavior sanitizer for the following events (as defined above):

```
alignment
bool
bounds
enum
float-cast-overflow
float-divide-by-zero
function
integer-divide-by-zero
nonnull-attribute
null
object-size
return
returns-nonnull-attribute
shift
signed-integer-overflow
unreachable
vla-bound
vptr
```

NOTE `vptr` is not included when this option is used with `-fsanitize-use-embedded-rt`.

`-fsanitize-blacklist=`*`file`*
`-fno-sanitize-blacklist`

Disable the generation of `-fsanitize` runtime checks in the specified functions or source code files (Section 5.3).

The specified option argument is a text file, with each line in the file specifying the name of a function or source file:

- Function names are prefixed with `fun:`
- File names are prefixed with `src:`

For example:

```
# Disable checks in function and source file
fun:my_func
src:my_file
```

Empty lines and lines starting with “#” are ignored.

File and function names can be specified using regular expressions, but note that“#” works as it does in shell wildcarding.

`-fsanitize-memory-track-origins[=`*`level`*`]`

Track the origin of uninitialized memory in the memory sanitizer (Section 5.9).

This option accepts the following level values:

`0`

Disable origin tracking.

`1`

Track and report where uninitialized values were allocated (default).

`2`

Track and report where uninitialized values were allocated, along with information on intermediate stores that the uninitialized values went through.

`-fsanitize-messages`
`-fno-sanitize-messages`

Control the generation of diagnostic messages for undefined behavior violations when using `-fsanitize-use-embedded-rt`. Enabled by default.

`-fsanitize-opt-size`
`-fno-sanitize-opt-size`

Reduce the code size of undefined behavior runtime checks when using `-fsanitize-use-embedded-rt`. Using this option may decrease program performance. Disabled by default.

`-fsanitize-source-loc`
`-fno-sanitize-source-loc`

Control the generation of file and line number information in messages for undefined behavior violations when using `-fsanitize-use-embedded-rt`. Enabled by default, except when used with `-fsanitize-opt-size`, then disabled by default.

`-fsanitize-use-embedded-rt`

Use alternate undefined behavior sanitizer instrumentation and runtime appropriate for embedded environments.

`-fshort-enums`
`-fno-short-enums`

Allocate to an enum type only as many bytes necessary for the declared range of possible values. The default is disabled.

`-fshort-wchar`
`-fno-short-wchar`

Force `wchar_t` to be `short unsigned int`. The default is disabled.

`-ftrap-function=`*`name`*

Issue a call to the specified function rather than a trap instruction.

`-ftrapv`

Trap on integer overflow.

`-ftrapv-handler=`*`name`*

Specify the function to be called in the case of an overflow.

`-funwind-tables`

Similar to `-fexceptions`, except that it only generates any necessary static data, without affecting the generated code in any other way.

`-fverbose-asm`

Add commentary information to the generated assembly code to improve code readability.

`-fvisibility=``[`**`default`**`|`**`internal`**`|`**`hidden`**`|`**`protected`**`]`

Set the default symbol visibility for all global declarations.

`-fwrapv`

Treat signed integer overflow as two's complement.

`-mhwdiv=``(`**`arm`**`|`**`thumb`**`|arm,`**`thumb`**`|`**`none`**`)`

Control the generation of hardware divide instructions in ARM or Thumb mode.

`arm`

Generate hardware divide instructions in Arm mode only.

`thumb`

Generate hardware divide instructions in Thumb mode only.

`arm,thumb`

Generate hardware divide instructions in ARM and Thumb modes.

`none`

Do not generate hardware divide instructions (default).

NOTE This option applies only to ARMv7 processors that support hardware divide.

`-mcpu=krait` automatically sets `-mhwdiv=arm,thumb`.

`-mllvm -aarch64-disable-abs-reloc`

Eliminate absolute relocation by changing all global variable references to be PC-relative.

This option is commonly used with `-mllvm -emit-cp-at-end`.

`-mllvm -aggressive-jt`

A jump table is an efficient method to optimize switch statements by replacing them with unconditional branch instructions and simple operations to transfer program flow to them.

This option enables switch statements with small ranges to be automatically converted to jump tables.

The default is disabled.

`-mllvm -arm-expand-memcpy-runtime`

Set a threshold of 8 or 16 bytes for expanding (inlining) memcpy calls.

This option enables the generation of runtime checks for copy sizes 8 or 16 bytes, and inlining of memcpy calls that have copy sizes smaller than or equal to 8 or 16 bytes. For any other copy size the memcpy function is invoked.

Enabling this option causes an LLVM IR-level transformation. The resultant code might be vectorized, if NEON is enabled.

This option is effective for optimization level equal or higher than `-O1`, `Os`, and `Oz`. Otherwise it is silently ignored.

`-mllvm -arm-memset-size-threshold`

Control the code generation for memset library calls using NEON vector stores.

This option specifies the maximum number of bytes of data in memset call that should be implemented with NEON vector stores. A memset call with data size above the specified threshold will not be compiled into vector store operations.

The default is 128.

`-mllvm -arm-memset-size-threshold-zeroval`

Control the code generation for memset library calls that write 0 value using NEON vector stores.

This option specifies the maximum number of bytes of data in memset call that writes 0 value that can be implemented with NEON vector stores. A memset call that writes 0 value with data size above the specified threshold will not be compiled into vector store operations.

The default is 32.

`-mllvm -arm-opt-memcpy`

The optimized libc for Krait targets includes two specialized memcpy functions for copy sizes greater than 8 and 16 bytes:

```
memcpyGT8(void*, const void*, size_t)
memcpyGT16(void*, const void*, size_t)
```

When this option is set in conjunction with `-mllvm -arm-expand-memcpy-runtime`, the compiler transforms the LLVM IR by replacing memcpy calls with the runtime checks for copy size less than or equal to 8 or 16 bytes and these specialized memcpy calls. Their implementation uses vector instructions and requires NEON to be enabled.

Note the user needs to additionally set the option for copy size threshold, `-mllvm -arm-expand-memcpy-runtime`.

This option has no effect if `-mllvm -arm-expand-memcpy-runtime` is disabled.

This option is effective for optimization level equal or higher than `-O1`, `Os` and `Oz`. Otherwise it is silently ignored.

The default is disabled.

`-mllvm -disable-thumb-scale-addressing`

Control the code generation of scaled immediate addressing in Thumb mode.

By default scaled immediate addressing is enabled in Thumb mode, unless `-mcpu=krait` is set in the command line.

To disable it, set `-mllvm -disable-thumb-scale-addressing=true`.

-mllvm -emit-cp-at-end

Place constant pool at the end of a function.

When this option is used in conjunction with **-mllvm -aarch64-disable-abs-reloc** (which changes all global variable references to be PC-relative), the compiler places the constant pool at the end of a function.

The default is disabled.

In the following example global variable "a" is loaded using the default relocation code:

```
movz x8, #:abs_g3:a
movk x8, #:abs_g2_nc:a
movk x8, #:abs_g1_nc:a
movk x8, #:abs_g0_nc:a
ldr w0, [x8]
```

Enabling this option with **-mllvm -aarch64-disable-abs-reloc** changes the code to the following:

```
ldr x8, .LCPI0_0
ldr w0, [x8]
ret
.LCPI0_0:
.xword a    // address of "a"
```

-mllvm -enable-android-compat

Control the generation of hardware divide instructions (Section 3.6).

-mllvm -enable-arm-addressing-opt

Promotes use of optimized address modes by merging ADD operations into the associated LOAD instruction.

The default is enabled.

`-mllvm -enable-arm-peephole`

Enable peephole optimizations to eliminate VMOV instructions, which can be an expensive operations.

This option controls two peephole optimizations:

- Eliminate vmovs from D to R to S

Eliminates excess VMOVs that result from copying a value from a D register to an S register.

There is no copy instruction from D to S, so the code generator inserts a VMOV from D to R and then another VMOV from R back to S. This peephole optimization eliminates the VMOVs by using D registers that alias S registers – registers D0-D15 are aliases as S0-S31. No copy is necessary to get to an S register from these D registers.

- Eliminate VMOVs from D to R for an ADD operation.

Eliminates excess VMOVs that result from an ADD instruction whose operands are defined by VMOVs from a D register. The ADD is replaced with a horizontal ADD using the VPADD instruction and a VMOV to get the result to the R register.

The default is enabled.

`-mllvm -enable-arm-zext-opt`

Removes redundant ZERO-EXTEND operations, for example, when preceded by a LOAD instruction that zero-extends the value to 32 bits as part of its operation.

The default is enabled.

`-mllvm -enable-print-fp-zero-alias`

When this option is used in conjunction with `-no-integrate-as`, the compiler prints FP compare-with-zero instructions using the alias format "fcmXY ..., #0" instead of the default LLVM format "fcmXY ..., #0.0" specified in the ARMv8 documentation.

This ensures assembly code compatibility between LLVM and GNU tools while the tools are out of sync (i.e., the 4.9 GNU assembler currently uses "#0" syntax).

`-mllvm -enable-round-robin-RA`

Enable a round-robin register allocation heuristic which selects registers avoiding back-to-back reuse to minimize false data dependency.

This heuristic works well for targets with limited register renaming capability, as in Krait targets.

The default is disabled, unless `-mcpu=krait` is specified.

`-mllvm -enable-select-to-intrinsics`

Expose more if-statements to be converted into LLVM IR's SELECT instruction which in turn can more easily be mapped to ARM HW instructions.

The default is disabled, unless `-mcpu=krait` is specified.

`-mllvm -favor-r0-7`

Enable a heuristic in the Greedy Register Allocator that better guides the assignment of high-order registers (R8-R15) which are currently avoided aggressively in the allocator. The allocator exploits the fact that a Thumb2 instruction that uses one of R8-15 registers must be encoded in 32 bits. So a candidate assigned to these registers has a very a high cost.

With this option, the allocator avoids this register assignment based on an additional cost, the candidate frequency in a function. The benefits are better code size reduction, better performance/power generated from better code density, and reduced spilling. This change impacts mostly Thumb code generation, but ARM code generation can also be affected because it disables R8-15 register avoidance.

The default is disabled.

NOTE Use `-falign-inner-loops` with `-favor-r0-7` to achieve the maximum benefit from loop alignment.

`-mllvm -force-div-attr`

Control the generation of hardware divide instructions (Section 3.6).

`-mllvm -prefetch-locality-policy=``(`**`L1`**`|`**`L2`**`|`**`L3`**`|`**`stream`**`)`

Configure data prefetch to be temporal or non-temporal.

`L1`

Temporal or retained prefetch allocated in L1 cache.

`L2`

Temporal or retained prefetch allocated in L2 cache.

`L3`

Temporal or retained prefetch allocated in L3 cache.

`stream`

Streaming or non-temporal prefetch.

The default is `L1`.

NOTE This option is available only for AArch64, and only when `-fprefetch-loop-arrays` is enabled.

`-mrestrict-it`
`-mno-restrict-it`

Control the code generation of IT blocks.

In the ARMv8 architecture (AArch32) IT blocks are deprecated in Thumb mode. They can only be one instruction long, and can only contain a subset of all 16-bit instructions.

`-mrestrict-it` disallows the generation of IT blocks that are deprecated in ARMv8.

`-mno-restrict-it` allows generation of legacy IT blocks (i.e., deprecated forms in ARMv7).

The default option setting is determined by the target architecture (ARMv8 or ARMv7). For ARMv8 (AArch32) Thumb mode, `-mrestrict-it` is enabled by default, while for other targets it is disabled by default.

3.4.17 Vectorization

`-fvectorize-loops`

Perform automatic vectorization of loop code (Section 4.4).

Vectorization is subject to the following constraints:

- On nested loops it is performed only on the innermost loop.
- It can be used at any code optimization level higher than `-O0`.
- It works only with the ARMv7 or ARMv8 processor architecture with the NEON extension. NEON is enabled either implicitly (by specifying a target processor such as Krait), or explicitly with `-mfpu=neon`.

NOTE `-fvectorize-loops` is enabled by default with `-O2`, `-O3`, `-O4`, and `-Ofast`.

`-ftree-vectorize`

Alias of `-fvectorize-loops`, provided for GCC compatibility.

`-fvectorize-loops-debug`

Equivalent to `-fvectorize-loops`, but also generates a report indicating which loops in the program were vectorized.

NOTE This option works best when used with the `-g` option to print out the precise location of the loops that get vectorized.

The GCC option `-ftree-vectorizer-verbose` is not supported in LLVM.

`-fprefetch-loop-arrays``[=`*`stride`*`]`
`-fno-prefetch-loop-arrays`

Control the automatic insertion of ARM `PLD` instructions into loops that are vectorized.

The argument *`stride`* specifies the distance that the `PLD` instruction attempts to load. If the argument is omitted, the compiler automatically chooses a value.

The default is disabled.

NOTE This option must be used with the `-fvectorize-loops` option.

3.4.18 Parallelization

`-fparallel`

Perform automatic parallelization of loop code (Section 4.5).

Parallelization is subject to the following restrictions:

- It must be specified (on the command line) when compiling each `.c` or `.cpp` file.
- It must additionally be specified on the command line that directs linking.
- It can be used only with `-O2`, `-O3`, `-O4`, or `-Ofast`.

`-fparallel-symphony`

Perform automatic parallelization of loop code at runtime using the SYMPHONY library (Section 4.5.1).

Parallelization using SYMPHONY is subject to the following restrictions:

- The `-fparallel-symphony` option must be specified when compiling each `.c` or `.cpp` file.
- The same option must also be specified when linking.
- SYMPHONY works only with dynamically-linked executables.

NOTE This option is an alternative to `-fparallel`, and must not be used with it.

3.4.19 Optimization

`-O0`

Do not optimize. This is the default optimization setting.

`-O`
`-O1`

Enable a small set of optimizations. This optimization level is not recommended for performance or code size.

`-O2`

Enable optimizations for performance, including automatic loop vectorization (Section 3.4.17). Optimizations enabled at `-O2` improve performance but may cause a small-to-moderate increase in compiled code size.

`-O3`

Enable aggressive optimizations for performance. Optimizations enabled at `-O3` improve performance but may cause a large increase in compiled code size.

`-O4`

Similar to `-Ofast`, but additionally enables advanced loop fusion and data layout optimizations for performance. Optimizations enabled at `-O4` improve performance but may cause a large increase in compiled code size.

NOTE The Qualcomm LLVM compilers define `-O4` differently from the standard LLVM compiler. In particular, the Qualcomm compilers do not enable link-time optimization (Section 4.7) in `-O4`, while the standard compiler does enable it in `-O4`, additionally mapping `-O4` to `-O3`.

`-Os`

Enable optimizations for code size. Optimizations enabled at `-Os` reduce code size at the cost of a small-to-moderate decrease in compiled code performance.

`-Ofast`

The following optimizations are enabled at the `-Ofast` level:

- All options enabled with `-O3` (including `-fvectorize-loops`)
- `-mllvm -switch-transpose=true`
- `-mllvm -unroll-allow-partial=true`
- `-mllvm -unroll-threshold=1000`
- `-mllvm -inline-threshold=375`
- `-ffast-math` and `-fmath-errno`
- If compiling for ARM mode:
  - `-mllvm -unroll-rt-prolog=false`
- If compiling for Thumb mode:
  - `-mllvm -unroll-rt-prolog=true`
  - `-mllvm -enable-lsr-nested=true`
  - `-mllvm -lsr-no-outer=false`
  - `-mllvm -favor-r0-7=true`
- If `-mllvm -favor-r0-7=true` is successfully set, then `-falign-inner-loops=8` option is also enabled.

For details on the `-mllvm` options listed above, please refer to the LLVM documentation. The LLVM options `-mllvm -favor-r0-7` and `-falign-inner-loops` are further described in this document.

If the user sets any of the above options in the command line, then the user setting prevails. For example, it the user sets `-mllvm -favor-r0-7=false` or `-fno-align-inner-loops`, then `-Ofast` will not enable favoring r0 to r7 registers nor inner loops alignment.

If `-Ofast` is combined with any other optimization level (`-Os`, `-O0` to `-O4`) in the command line, the last `-O` option prevails.

The following are the recommended options for performance and code size optimizations.

1. Performance optimizations

The LLVM compilers generate the best performing code with the `-Ofast` option.

The following combination of options are recommended for Krait cores:

`-Ofast -mcpu=krait` (Thumb mode and `-fvectorize-loops` are enabled by default)

`-Ofast -mcpu=krait -marm` (ARM mode and `-fvectorize-loops` are enabled by default)

For non-Krait cores the following options are recommended:

```
-O3 -mllvm -unroll-threshold=1000 -mllvm -unroll-allow-partial
-mllvm -inline-threshold=325
```

The `-unroll-threshold` and `-inline-threshold` options increase the limits of loop unrolling and inlining respectively to exploit superscalar architectures such as Krait. In general, more aggressive loop unrolling and function inlining contributes to better performance.

2. Code-size optimizations

Currently, LLVM generates smallest code when compiled for Thumb2 mode. The following are the options recommended for generating compact code:

```
-Os -mthumb
```

`-Osize`

Enable `-Os` level optimizations and some additional options that trade off performance for best code size.

If `-Osize` is combined with any other optimization level (`-Os`, `-Ofast`, `-O0` to `-O4`) in the command line, the last `-O` option prevails.

NOTE This option has been tuned for ARMv7 targets only. It has not been tuned for ARMv8 targets (AArch32 and AArch64) and therefore should not be used with them.

`-Oz`

Enable optimizations for code size at the expense of performance. Optimizations enabled with `-Oz` reduce code size at the cost of a potentially significant decrease in compiled code performance.

3.4.20 Specific optimizations

`-falign-functions``[=`*`n`*`]`

Control function alignment.

Setting `-falign-functions=1` and `-fno-align-functions` are equivalent, resulting in disabling function alignment.

Setting `-falign-functions=0` or `-falign-functions` (with no value specified) enables function alignment using the target's default alignment value.

Setting `-falign-functions=`*`n`* enables function alignment using the next power-of-two greater than *`n`* as the alignment value, where *`n`* is the number of bytes.

The default is to not align functions.

To enable function alignment at the `-Os` level, an additional `falign-os` option must be set.

`-falign-jumps``[=`*`n`*`]`

Control jump alignment.

Setting `-falign-jumps=1` and `-fno-align-jumps` are equivalent, resulting in disabling jump alignment.

Setting `-falign-jumps=`*`n`* enables jump alignment using the next power-of-two greater than *`n`* as the alignment value, where *`n`* is the number of bytes.

The default is to not align jumps.

To enable jump alignment at the Os level, an additional `falign-os` option must be set.

`-falign-labels``[=`*`n`*`]`

Control label (branch target) alignment.

Setting `-falign-labels=1` and `-fno-align-labels` are equivalent, resulting in disabling label alignment.

Setting `-falign-labels =0` or `-falign-labels` (with no value specified) enables labels alignment using the target's default alignment value.

Setting `-falign-labels=`*`n`* enables label alignment using the next power-of-two greater than *`n`* as the alignment value, where *`n`* is the number of bytes.

The default is to not align labels.

To control the type of label that should be aligned, use the `-mllvm branch-target-align` option with strings "none" (do not align branch targets), "nocalls" (do not align function calls), "allcalls" (align after function calls).

To enable label alignment at the Os level, `-falign-os` must also be set.

`-falign-loops``[=`*`n`*`]`

Control loop alignment.

Setting `-falign-loops=1` and `-fno-align-loops` are equivalent, resulting in disabling loop alignment.

Setting `-falign-loops=0` or `-falign-loops` (with no value specified) enables loop alignment using the target's default alignment value.

Setting `-falign-loops=`*`n`* enables loop alignment using the next power-of-two greater than *`n`* as the alignment value, where *`n`* is the number of bytes.

The default is to not align loops.

To enable loop alignment at the Os level, an additional `falign-os` option must be set.

`-falign-inner-loops`
`-fno-align-inner-loops`

Control innermost loop alignment. When enabled, only the start basic block of innermost loops is aligned.

Setting `-falign-inner-loops=1` and `-fno-align-inner-loops` are equivalent, resulting in disabling innermost loop alignment.

Setting `-falign-inner-loops=0` or `-falign-inner-loops` (with no value specified) enables innermost loop alignment using the target's default alignment value.

Setting `-falign-inner-loops=`*`n`* enables innermost loop alignment using the next power-of-two greater than *`n`* as the alignment value, where *`n`* is the number of bytes.

`-falign-loops` and `-falign-inner-loops` are incompatible and cannot be set simultaneously, otherwise a compiler error is generated.

The default is to not align innermost loops.

To enable innermost loop alignment at the Os level, an additional `-falign-os` must be set.

`-falign-os`
`-fno-align-os`

Control alignment in Os level.

The default is to ignore alignment options in Os level.

When enabling alignment of loops, functions and labels in Os level set `-falign-os`, otherwise the compiler generates a warning of unused option. To disable it, set `-fno-align-os`.

`-fdata-sections`

Assign each data item to its own section.

`-ffunction-sections`

Assign each function item to its own section in the output file. The section is named after the function assigned to it.

`-finline`

Specify the `inline` keyword as active.

`-finline-functions`

Perform heuristically-selected inlining of functions.

`-floop-pragma`

Enable auto-parallelization and auto-vectorization when using loop pragmas.

`-fnomerge-all-constants`

Do not merge constants.

`-fomit-frame-pointer`

Do not store the stack frame pointer in a register if it is not required in a function.

`-foptimize-sibling-calls`

Optimize function sibling calls and tail-recursive calls.

`-fstack-protector`

Generate code which checks selected functions for buffer overflows.

`-fstack-protector-all`
Generate code which checks *all* functions for buffer overflows.

`-fstack-protector-strong`
Generate code which applies strong heuristic to check additional selected functions for buffer overflows.

Additional functions checked include those with local array definitions or references to local frame addresses.

`-fstrict-aliasing`
Enforce the strictest possible aliasing rules for the language being compiled.

`-funit-at-a-time`
Parse the entire compilation unit before beginning code generation.

`-funroll-all-loops`
Unroll *all* loops.

`-funroll-loops`
Unroll selected loops.

`-fno-zero-initialized-in-bss`
Assign all variables that are initialized to zero to the BSS section.

`--param ssp-buffer-size=`*`size`*
Specify the minimum size (in bytes) that a buffer must be in order to have buffer-overflow checks generated for it by the `-fstack-protector` options. The default value is 8.

3.4.21 Math optimization

`-fassociative-math`
Allow the operands in a sequence of floating-point operations to be re-associated.

Because this option may reorder floating-point operations, it should be used with caution when exact results are required (with no expectation of an error cutoff).

To use this option, both `-fno-signed-zeros` and `-fno-trapping-math` must be enabled, while `-frounding-math` must *not* be enabled.

NOTE This option enables additional features of parallelization (Section 4.5).

`-ffast-math`
Enable 'fast-math' mode in the compiler front-end. This has no effect on optimizations, but defines the preprocessor macro `__FAST_MATH__` which is the same as the GCC `-ffast-math` option.

`-ffinite-math-only`
Enable optimizations which assume that floating-point argument and result values are never NaNs nor +-Infs.

`-fno-math-errno`
Do not set `errno` after using single-instruction math functions.

`-freciprocal-math`
Enable optimizations which assume that the reciprocal of a value can be used instead of dividing by the value.

`-fno-signed-zeros`
Enable optimizations which ignore the sign of floating point zero values.

`-fno-trapping-math`
Enable optimizations which assume that floating-point operations cannot generate user-visible traps.

`-funsafe-math-optimizations`
Enable code optimizations which assume that the floating-point arguments and results are valid, and which may violate IEEE or ANSI standards.

This option enables `-fno-signed-zeros`, `-fno-trapping-math`, `-fassociative-math`, and `-freciprocal-math`.

3.4.22 Link-time optimization

`-flto`

Perform link-time optimization (Section 4.7).

This option can be used when the files in a program are compiled separately. In this case the option must be specified when compiling each source file, and again when the compiler is used to link the resulting object files.

When this option is used with `-c`, it produces a bitcode file which is used during link-time optimization (LTO).

NOTE All compile-time options must be passed to the linker command line so that LTO can generate code for the specified optimization level. To ensure that the options are passed correctly, it is strongly recommended to use `clang/clang++` to perform the linking.

NOTE When this option is used, the default system linker changes to the `qcld` linker, and either `qcld` or the `gold` linker must be used as the system linker. The `gold` linker can be specified with the `-fuse` option (Section 3.4.10).

The `gold` linker cannot be used with the Windows version of the LLVM compilers – in this case only the `qcld` linker can be used to perform LTO.

3.4.23 Profile-guided optimization

`-fprofile-instr-generate``[=`*`filename`*`]`

Specify the name and location of the raw profile data file to be created. The default name is `default.profraw`. The default location is "`/sdcard`" for Android applications, or the current directory for non-Android applications.

The raw profile data file is used in instrumentation-based profile-guided optimization (Section 4.8).

`-fprofile-instr-use=`*`filename`*

Use the specified instrumentation-generated profile data file to perform profile-guided optimization.

`-fprofile-sample-use=`*`filename`*

Use the specified sampling-generated profile data file to perform profile-guided optimization.

`--fprofile-instr-sync-interval=`*`interval`*

Periodically sync the collected profile data to the profile data file with the specified time interval (in milliseconds).

3.4.24 Optimization reports

`-fopt-reporter=(vectorizer|parallelizer|all)`
Request the specified type of optimization report data (Section 4.10).

`-polly-max-pointer-aliasing-checks`
Increase the number of runtime checks that are allowed to be inserted into a loop in order to disambiguate the pointers, thus enabling the loop to be vectorized

`-Rpass=loop-opt`
Output the line numbers of the loops that were auto-parallelized and/or vectorized.

`-Rpass-missed=loop-opt`
Output the line number and reason why a loop was not optimized.

3.4.25 Compiler security

`--analyze`
Invoke the static program analyzer (Section 5.14.1) on the specified input files.

`-analyzer-checker=`*`checker`*
Enable the specified checker or checker category in the static program analyzer.

The checker categories are `alpha`, `core`, `cplusplus`, `debug`, and `security`. Enabling a checker category enables all the checkers in that category.

For a complete list of checker names use `-analyzer-checker-help`.

NOTE `-analyzer-checker` must be prefixed with -`Xclang`

`-analyzer-checker-help`
List the complete set of checkers and their categories for use in `-analyzer-checker` and `-analyzer-checker-disable`.

NOTE `-analyzer-checker-help` must be prefixed with -`cc1`

`-analyzer-disable-checker=`*`checker`*
Disable the specified checker or checker category in the static program analyzer.

The checker categories are `alpha`, `core`, `cplusplus`, `debug`, and `security`. Disabling a checker category disables all the checkers in that category.

For a complete list of checker names use `-analyzer-checker-help`.

NOTE `-analyzer-disable-checker` must be prefixed with -`Xclang`

`--analyzer-output html`
Generate the static analyzer output report in HTML format.

The default report format is plist.

NOTE `--analyzer-output` and its argument must each be prefixed with `-Xclang`.

`--analyzer-Werror`
Convert all static analyzer warnings into errors.

`--compile-and-analyze` *`dir`*
Invoke static program analyzer on an entire program.

The analysis report files are written to the specified directory.

`-ffcfi`
Enable control-flow integrity checks (Section 5.13).

`-fno-fcfi`
Disable control-flow integrity checks.

3.5 Warning and error messages

LLVM provides a number of ways to control which code constructs cause the compilers to emit errors and warning messages, and how the messages are displayed to the console.

3.5.1 Controlling how diagnostics are displayed

When LLVM emits a diagnostic, it includes rich information in the output, and gives you fine-grain control over which information is printed. LLVM has the ability to print this information. The following options are used to control the information:

- A file/line/column indicator which shows exactly where the diagnostic occurs in your code.
- A categorization of the diagnostic as a note, warning, error, or fatal error.
- A text string describing the problem.
- An option indicating how to control the diagnostic (for diagnostics that support it) [`-fdiagnostics-show-option`].
- A high-level category for the diagnostic for clients that want to group diagnostics by class (for diagnostics that support it) [`-fdiagnostics-show-category`].
- The line of source code that the issue occurs on, along with a caret and ranges indicating the important locations [`-fcaret-diagnostics`].
- "FixIt" information, which is a concise explanation of how to fix the problem (when LLVM is certain it knows) [`-fdiagnostics-fixit-info`].
- A machine-parseable representation of the ranges involved (disabled by default) [`-fdiagnostics-print-source-range-info`].

For more information on these options see Section 3.4.7.

3.5.2 Diagnostic mappings

All diagnostics are mapped into one of the following classes:

- Ignored
- Note
- Warning
- Error
- Fatal

3.5.3 Diagnostic categories

Though not shown by default, diagnostics can each be associated with a high-level category. This category is intended to make it possible to triage builds which generate a large number of errors or warnings in a grouped way.

Categories are not shown by default, but they can be turned on with the `-fdiagnostics-show-category` option (Section 3.4.7). When this option is set to "`name`", the category is printed textually in the diagnostic output. When set to "`id`", a category number is printed.

NOTE The mapping of category names to category identifiers can be obtained by invoking LLVM with the option `-print-diagnostic-categories`.

3.5.4 Controlling diagnostics with compiler options

LLVM can control which diagnostics are enabled through the use of options specified on the command line.

The `-W` options are used to enable warning diagnostics for specific conditions in a program. For instance, `-Wmain` will generate a warning if the compiler detects anything unusual in the declaration of function `main()`.

`-Wall` enables *all* the warnings defined by LLVM. `-w` disables all of them.

Warnings for a specific condition can be disabled by specifying the corresponding `-W`*`cond`* option as `-Wno-`*`cond`*. For instance, `-Wno-main` disables the warning normally enabled by `-Wmain`.

`-Werror=`*`cond`* changes the specified warning to an error (Section 3.5.2). `-Werror` specified without a condition changes *all* the warnings to errors. `-ferror-warn` changes just the warnings that are listed in the specified text file.

`-pedantic` and `-pedantic-errors` enable diagnostics that are required by the ISO C and ISO C++ standards.

3.5.5 Controlling diagnostics with pragmas

LLVM can also control which diagnostics are enabled through the use of pragmas in the source code. This is useful for disabling specific warnings in a section of source code. LLVM supports GCC's pragma for compatibility with existing source code, as well as several extensions.

The pragma may control any warning that can be used from the command line. Warnings can be set to ignored, warning, error, or fatal. The following example instructs LLVM or GCC to ignore the `-Wall` warnings:

```
#pragma GCC diagnostic ignored "-Wall"
```

In addition to all the functionality provided by GCC's pragma, LLVM also enables you to push and pop the current warning state. This is particularly useful when writing a header file that will be compiled by other people, because you don't know what warning flags they build with.

In the below example `-Wmultichar` is ignored for only a single line of code, after which the diagnostics return to whatever state had previously existed:

```
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wmultichar"

char b = 'df'; // no warning.

#pragma clang diagnostic pop
```

The `push` and `pop` pragmas save and restore the full diagnostic state of the compiler, regardless of how it was set. That means that it is possible to use push and pop around GCC-compatible diagnostics, and LLVM will push and pop them appropriately, while GCC will ignore the pushes and pops as unknown pragmas.

NOTE While LLVM supports the GCC pragma, LLVM and GCC do not support the same set of warnings. Thus even when using GCC-compatible pragmas there is no guarantee that they will have identical behavior on both compilers.

3.5.6 Controlling diagnostics in system headers

Warnings are suppressed when they occur in system headers. By default, an included file is treated as a system header if it is found in an include path specified by `-isystem`, but this can be overridden in several ways.

The `system_header` pragma can be used to mark the current file as being a system header. No warnings will be produced from the location of the pragma onwards within the same file.

```
char a = 'xy'; // warning

#pragma clang system_header

char b = 'ab'; // no warning
```

The options `-isystem-prefix` and `-ino-system-prefix` can be used to override whether subsets of an include path are treated as system headers. When the name in a `#include` directive is found within a header search path and starts with a system prefix, the header is treated as a system header. The last prefix on the command-line which matches the specified header name takes precedence. For example:

```
$ clang -Ifoo -isystem bar -isystem-prefix x/
    -ino-system-prefix x/y/
```

Here, `#include "x/a.h"` is treated as including a system header, even if the header is found in `foo`, and `#include "x/y/b.h"` is treated as not including a system header, even if the header is found in `bar`.

An `#include` directive which finds a file relative to the current directory is treated as including a system header if the including file is treated as a system header.

3.5.7 Enabling all warnings

In addition to the traditional `-W` flags, *all* warnings can be enabled by specifying the option `-Weverything`.

`-Weverything` works as expected with `-Werror`, and also includes the warnings from `-pedantic`.

NOTE When this option is used with `-w` (which disables all warnings), `-w` takes priority.

3.6 Using GCC cross compile environments

The LLVM compilers are stand-alone compilers which rely on an existing system tools "root" environment – also known as *sysroot* – for accessing include files and libraries (as well as an ARM cross linker). The compilers are prebuilt to work with a GCC sysroot environment: to include header files and libraries in the build, they assume a predefined directory structure anchored by a GCC system root directory.

For example, the GCC sysroot for the ARMv8 AArch64 toolchain has the following structure:

```
/aarch64-linux-gnu/
/bin/
/debug-root/
/include/
/include/c++/4.8.2/
/backward/
/lib/
/libc/
/usr/
   /include/
```

The top level of the GCC tools directory must have a subdirectory that matches the target triple specified on the compiler command line (Section 3.4.15). The target triple directory typically contains a `libc` directory which mimics a host compilation environment by storing the following items:

- The library files in `*GCC-top*/*target-triple*/libc/lib`
- The include files in `*GCC-top*/*target-triple*/libc/usr/include`

Thus the sysroot location is `*GCC-top*/*target-triple*/libc`.

The sysroot location is specified with the compile option `--sysroot`.

The LLVM compilers additionally require the location of the GNU linker (and also an assembler, if not using the LLVM integrated assembler). This location is specified with the option `-B` or `-gcc-toolchain`, and must point to the top of the GCC toolchain directory.

For example, the following two LLVM commands compile a source file and generate code for the ARMv8 AArch64 ISA:

```
clang -target aarch64-linux-gnu --sysroot=GCC-top/aarch64-
                                  linux-gnu/libc -BGCC-top foo.c

clang -target aarch64-linux-gnu --sysroot=GCC-top/aarch64-linux-
                          gnu/libc --gcc-toolchain=GCC-top foo.c
```

With C++, it may be necessary to add a set of C++ include directories so the LLVM compilers can correctly search for the header files. Note that this is required only with certain GCC toolchain sysroots – in such cases the following directories should be added to the LLVM compiler command using the `-isystem` option:

```
-isystem GCC-top/include/c++/GCC-version
-isystem GCC-top/include/c++/GCC-version/triple
-isystem GCC-top/include/c++/GCC-version/backward
```

... where *`GCC-version`* indicates the version number of the GCC toolchain (4.6, 4.8.1, etc.).

NOTE The target triple specified above may differ from the target triple used on the compiler command line.

3.7 Using LLVM with GNU Assembler

Snapdragon LLVM includes support for using the GNU Assembler (GAS) as the system assembler. The options "`-mllvm -enable-android-compat`" and "`-mllvm -force-div-attr`" (Section 3.4.16) are used to control the generation of hardware divide instruction so it is compatible with various versions of GAS.

By default the LLVM compiler only emits the DIV attribute when the `-mhwdiv` option is specified. Depending on the GAS version you are using, it may be necessary to change this default behavior. Use the following guideline:

- GCC 4.6 and older releases
  - The DIV attribute is not emitted by GCC/GAS compiler/assembler.

    When using LLVM with this GNU version, you must specify the option "`-mllvm -enable-android-compat`".
- GCC 4.6 / 4.7 releases
  - The DIV attribute is emitted whether or not the option `-mhwdiv` is specified. It is given a different value depending on the target architecture specified:
    - 0 - Allow hardware division if supported in the target architecture, or if no information exists.
    - 1 - Disallow hardware division.
    - 2 - Allows hardware division as an optional extension above the base target architecture hardware features.

    When using LLVM with this GNU version, you must specify the option "`-mllvm -force-div-attr`".
- Post GCC 4.7 releases
  - The DIV attribute is emitted only when the option `-mhwdiv` is specified.

3.8 Built-in functions

```
__builtin_neon_memcpy_1024(void*, const void*, size_t)
```

The header file `arm_memcpy_bias.h` contains the declaration of a specialized ARM `memcpy` builtin for copy size of 1024.

Using this built-in function in the source will result in generated code with a runtime check for copy size of 1024 and the inlining of the `memcpy` specialized implementation for copy size 1024 using vector instructions.

NOTE NEON must be enabled to use this built-in.

3.9 Compilation phases

The LLVM compiler consists of a driver program (named `cc1`) which in turn invokes a set of tools that perform the various phases of the overall compilation process.

View phases

To view these phases during compilation, invoke the compiler using the option `-ccc-print-phases` (Section 3.4.2). For example:

```
clang -ccc-print-phases test.c
```

This option prints the following information during compilation:

```
0: input, "test.c", c
1: preprocessor, {0}, cpp-output
2: compiler, {1}, assembler
3: assembler, {2}, object
4: linker, {3}, image
```

This option is useful when paired with options that control compilation. For example:

```
clang -c -ccc-print-phases test.c
0: input, "test.c", c
1: preprocessor, {0}, cpp-output
2: compiler, {1}, assembler
3: assembler, {2}, object

clang --analyze -ccc-print-phases test.c
0: input, "test.c", c
1: preprocessor, {0}, cpp-output
2: analyzer, {1}, plist
```

View phase commands

To view the actual tool commands performed by the driver program at each compilation phase, invoke the compiler using the option `-###` (Section 3.4.2). For example:

```
clang -### test.c --sysroot=path_to_aarch64_android_sysroot
                  --gcc-toolchain=path_to_aarch64_android_tools
                  --target=aarch64-linux-android
```

This option prints the following command information:

Preprocessor and compiler:

```
clang-3.8" "-cc1" "-triple" "armv4t--linux-androideabi"
"-emit-obj"
...
"-o" "/tmp/t-871e8e.o" "-x" "c" "t.c"
```

Linker:

```
"ld" "--sysroot=..."
...
"-o" "a.out"
...
"/tmp/t-2e40e1.o"
```

Specify phase options

To pass a command option to the tool that performs a specific compilation phase, invoke the compiler using the option `-X` (Section 3.4.2). For example:

```
clang -Xlinker --print-map test.c
```

In this example, `-X` is used to pass the option `--print-map` to the linker.

`-X` specifies the tool that the option will be passed to:

`-Xclang`	Compiler
`-Xassembler`	Assembler
`-Xlinker`	Linker
`-Xanalyzer`	Static analyzer

If the option to be passed contains one or more arguments that are separated from the option name by spaces, then you will need to use `-X` multiple times in order to pass the option. For example:

```
clang --analyze -Xclang -analyzer-output -Xclang html
                                              -o dir test.c
```

In this example, `-X` is used twice to pass the option “`-analyzer-output html`” to the compiler.

4 Code Optimization

4.1 Overview

The LLVM compilers provide many tools and features for improving the size or speed of the generated object code.

This chapter covers the following topics:

- Optimizing for performance
- Optimizing for code size
- Automatic vectorization
- Automatic parallelization
- Merging functions
- Profile-guided optimization
- Loop optimization pragmas
- Optimization reports

NOTE It is highly recommended to try using the various code optimizations to improve the performance of your program. Using just the default optimization settings is likely to result in suboptimal performance.

4.2 Optimizing for performance

LLVM currently generates the fastest code when compiling for ARM mode.

Table 4-1 lists the options to use for optimizing code performance.

Table 4-1 Optimizing for performance

Core	Options
ARMv7	`-Ofast -mcpu=krait`
ARMv8 (AArch32)	`-Ofast -mcpu=cortex-a57`
ARMv8 (AArch64)	

For more information on `-Ofast` see Section 3.4.19.

4.3 Optimizing for code size

LLVM currently generates the smallest code when compiling for Thumb2 mode.

NOTE Thumb2 is available only on ARMv7 and AArch32.

Table 4-2 lists the options to use for optimizing code size.

Table 4-2 Optimizing for code size

Core	Options
ARMv7	`-Osize -mthumb`
ARMv8 (AArch32)	
ARMv8 (AArch64)	`-Os -mcpu=cortex-a57`

For ARMv7, the `-Osize` option is preferred over `-Os` because it enables additional code-size optimizations.

For more information on `-Osize` see Section 3.4.19.

4.4 Automatic vectorization

LLVM includes support for automatic code vectorization. By default the vectorizer is enabled at code optimization level `-O2` or higher. To enable it at lower optimization levels use the `-fvectorize-loops` option (Section 3.4.17).

Vectorization can be used at any code optimization level higher than `-O0`.

To see which loops in a program get vectorized, use the following option:

```
-fvectorize-loops-debug
```

Vectorization works only with the ARMv7 or ARMv8 processor architecture with the NEON extension. NEON is enabled either implicitly (by specifying a target processor such as `-mcpu=krait`), or explicitly with `-mfpu=neon`.

The following is an example of a loop that can be vectorized with `-fvectorize-loops`:

```
void foo(int * restrict A, int N) {
  for (int i = 0; i < N; i++)
    A[i] = A[i] + 1;
}
```

For vectorization of floating point computation, the GCC option `-ffast-math` should be specified. Because floating point vectorizations (reductions in particular) are not IEEE compliant, the fast math option is required to ensure maximum vectorization of floating point computations.

NOTE The vectorizer can also be enabled using the option `-ftree-vectorize`, which is an alias for `-fvectorize-loops`.

The GCC option `-ftree-vectorizer-verbose` (for printing out verbose information on a vectorized loop) is not supported. Instead, use `-fvectorize-loops-debug`.

The vectorizer currently operates only on the innermost loop of a nested loop.

4.5 Automatic parallelization

The Qualcomm LLVM compilers include support for automatic code parallelization. By default parallelization is disabled – to enable it use the `-fparallel` option (Section 3.4.18).

Parallelization can be used only with code optimization level `-O2`, `-O3`, `-O4`, or `-Ofast`.

Automatic code parallelization enables selected loops to be executed in parallel for faster performance. During parallelization, if a loop is determined to be free of any data, control, or memory dependencies, it is then split into multiple loops, each of which performs part of the work from the original loop. The resulting loops are dispatched to work queues on separate cores so they can be executed in parallel.

Parallelization requires a runtime component which is linked into the final executable image. The purpose of the component is to initialize a new thread at program initialization time, and subsequently manage the work queues during parallel execution.

While automatic code parallelization can significantly improve overall performance by distributing work across multiple cores, it accomplishes this by putting otherwise underutilized cores to use. Because other cores get used, performance becomes a function of the entire system, and is not fully determinable at compile time. Thus it is possible for performance to improve, but also for the net performance to decline. Although the threads maintain the cores in a power-saving mode when they are not working, the additional work that is done in parallel can increase the overall power usage.

For this reason automatic code parallelization is not enabled by default in the compiler, and its use must be evaluated on a case-by-case basis.

4.5.1 Auto-parallelization using SYMPHONY library

The Qualcomm LLVM compilers use a library named SYMPHONY to manage loop auto-parallelization at runtime in multicore asynchronous runtime environments. SYMPHONY uses work stealing and adaptive scheduling to provide more opportunities for speeding up parallel loops when using auto-parallelization.

Using SYMPHONY

To perform auto-parallelization with SYMPHONY, use the `-fparallel-symphony` option (Section 3.4.18).

`-fparallel-symphony` is used in place of `-fparallel`, and can be used with the other auto-parallelization options (such as `-fparallel-num-workloads` to control loop chunking).

It is recommended to use `-fparallel-symphony` only with the highest optimization level (`-Ofast`). However, it can be used with lower optimization levels.

NOTE For full documentation on SYMPHONY (including instructions on how to download the SYMPHONY System Manager SDK), see: https://developer.qualcomm.com/software/symphony-system-manager-sdk

SYMPHONY can be used only with Android applications.

SYMPHONY library

To perform auto-parallelization with SYMPHONY, the SYMPHONY dynamic library must be available on the target Android device. If this library is not found on the device, the program will generate an error message indicating that it is unable to load SYMPHONY.

The SYMPHONY library file `libsymphony-1.0.0.so` should be stored in the following location:

- `/system/vendor/lib64` (Android 64-bit devices)
- `/system/vendor/lib` (Android 32-bit devices)

To obtain the proper version of the SYMPHONY library for your Android device, download it from:

https://developer.qualcomm.com/software/symphony-system-manager-sdk

Table 4-3 lists the SYMPHONY library versions for the supported development platforms.

Table 4-3 SYMPHONY library versions

Platform		SYMPHONY Library File
Windows	64-bit	`C:\Program Files (x86)\Qualcomm\Symphony SDK\1.0.0\aarch64-linux-android\lib`
	32-bit	`C:\Program Files (x86)\Qualcomm\Symphony SDK\1.0.0\arm-linux-androideabi\lib`
Linux	64-bit	`/opt/Qualcomm/Symphony/1.0.0/aarch64-linux-android\lib`
	32-bit	`/opt/Qualcomm/Symphony/1.0.0/arm-linux-androideabi\lib`

The Android `adb` utility can be used to push the library file to the Android file system:

- `adb push libsymphony-1.0.0.so /system/vendor/lib64/` (64-bit)
- `adb push libsymphony-1.0.0.so /system/vendor/lib/` (32-bit)

Command line example

The following example shows the commands necessary to compile, link, and run a program using auto-parallelization with the SYMPHONY library.

Compile:

```
$ clang --target aarch64-linux-android
--sysroot=<AArch64_Android_Sysroot>
--gcc-toolchain=<AArch64_Android_Toolchain>
-Ofast -fparallel-symphony -c /tmp/test.c
```

Link:

```
$ clang --target aarch64-linux-android
--sysroot=<AArch64_Android_Sysroot>
--gcc-toolchain=<AArch64_Android_Toolchain>
-Ofast -fparallel-symphony /tmp/test.o -o a.out
```

Run:

```
$ adb push a.out /data/data/
$ adb shell chmod 755 /data/data/a.out
$ adb shell /data/data/a.out
```

The executable file `a.out` will run without problems as long as the SYMPHONY library is available in the specified location, and the application can be parallelized.

4.6 Merging functions

LLVM includes support for function merging. By default this optimization is disabled – to enable it use the `-fmerge-functions` option (Section 3.4.16).

Function merging attempts to improve code size by merging functions that are equivalent or differ in only a few instructions. The optimization uses a number of heuristics to determine whether it is worthwhile to merge a pair of functions. For instance, very small functions or functions with significant differences are usually not merged.

The following example shows how function merging works:

```
int f1(int a, int b) {           int f2(int a, int b) {
int x;                           int x;
x = a + 4;                       x = a + 10;
return x * b;                    return x * b;
}                                 }
```

Function merging determines that functions `f1` and `f2` are similar, and replaces them with the following functions:

```
int f1__merged(int a, int b, int choice) {
int x;
if (choice)
   x = a + 10;
else
   x = a + 4;
return x * b;
}

int f1(int a, int b) {
return f1__merged(a, b, 0);
}

int f2(int a, int b) {
return f1__merged(a, b, 1);
}
```

This example is for illustration purposes only. In practice, the optimizer would determine that functions `f1` and `f2` are too small to be worth merging.

NOTE Because function merging may have a negative impact on program performance, it is disabled by default, and becomes enabled only when it is specified explicitly.

4.7 Link-time optimization

Link-time optimization (LTO) comprises a set of powerful inter-modular optimizations which are performed during the linking stage of compilation.

LTO expands the scope of optimizations from individual modules to the entire program (or at least to all the modules visible at link time). This enables deeper compiler analysis (such as better alias analysis) and more effective code transformations (such as function inlining), which can result in improved performance and code size.

When used with `-c`, the `-flto` option produces a file containing the LLVM compiler's intermediate representation (also known as *bitcode*). This file can be subsequently used in a final link step which then performs inter-module code optimizations on the file contents.

LTO comprises the following elements:

- The link-time optimizer, a compiler feature (controlled with `-flto`) which performs the inter-modular optimizations while linking the files together.
- The LTO-specific attribute `lto_preserve`, which when applied to a C or C++ function or variable prevents it from being discarded by the link-time optimizer.

NOTE The Snapdragon LLVM ARM linker has been verified to support LTO on ARMv7 and ARMv8 targets, and Linux and Windows hosts. The GNU Gold linker may support LTO for ARMv8, depending on the GCC toolchain/sysroot version used. LTO is not supported on Windows using the Gold linker.

For more information on the Snapdragon ARM LLVM linker, see the *Snapdragon Arm LLVM Linker User Guide*.

For more information on the Gold linker see llvm.org/docs/GoldPlugin.html.

Link-time optimizer

The link-time optimizer is invoked with the following command:

```
clang -flto input_files...
```

The optimizer inputs several LLVM bitcode files or archives. It then links the specified files together, performs the specified inter-modular optimizations on them as a whole, and finally generates a single assembly file containing the optimized result.

An important optimization that the optimizer performs is the aggressive removal of any functions that it determines are not used. To provide the optimizer with a larger context for determining if a function is used, the list of filenames may include additional non-bitcode objects and archives. The optimizer will use the symbol information in these files to determine if a function should be preserved.

NOTE The optimizer requires archives to be homogeneous: the members of a given archive must be either all bitcode files or all object files.

4.8 Profile-guided optimization

Profile-guided optimization (PGO) is a two-step process:

- A program is first executed to collect profile information on it.
- The program is then recompiled, this time using the collected profile information to improve the code optimization that can be performed on the program.

The availability of accurate source code profile information enables the compiler to generate better optimized code: the compiler can focus on costly high-performance optimizations (in terms of code size or compile time) at the profile-identified hot spots, while limiting adverse code generation trade-offs to pathways that are relatively cold.

PGO can use two different kinds of profile information:

- Instrumentation-based profiling
- Sampling-based profiling

Each method offers distinct advantages and disadvantages when performing PGO. However, both provide the compiler with useful information for improving code optimization.

PGO uses the same compile options that are described here:

clang.llvm.org/docs/UsersManual.html#profile-guided-optimization

4.8.1 Instrumentation-based PGO

The instrumentation-based approach to PGO relies on a special build of the user's code, which inserts instrumentation that generates the appropriate profile information. The resulting information can be used for PGO during a subsequent build.

NOTE An instrumented binary has extra runtime overhead and executes more slowly than normal, but the generated profile information still accurately reflects the code's un-instrumented execution.

The following procedure explains how to perform instrumentation-based PGO:

Step 1: Build instrumented application

Compile and link your application code, using the compile option `-fprofile-instr-generate`. For example:

```
clang++ -O2 -fprofile-instr-generate source.cc -o application
```

NOTE `-fprofile-instr-generate` optionally accepts a filename argument which specifies the name and location of the raw profile data file to be created. Otherwise the file will be created with the default name and location.

Step 2: Generate profile information

Run the built application on your device to generate the profile information. For example, to run the above application on Android, perform the following commands:

```
HOST$: adb push application /data/local/tmp
HOST$: adb shell
DEVICE$ cd /data/local/tmp
DEVICE$ ./application
```

This command sequence creates the raw profile data file "`/sdcard/default.profraw`".

Step 3: Convert profile information

Profile information can be generated either by running the instrumented program once (which results in a single set of profile information), or by running the program several times with different input data (which results in several sets of profile information).

In either case, the collected "raw" profiles must be converted to a file format profile that is compatible with the Snapdragon LLVM version of PGO. To do this, use the LLVM tool `llvm-profdata` and its "merge" functionality. For example:

```
llvm-profdata merge -output=application.profile dataset-1.profraw
                                              dataset-2.profraw
```

The above example inputs two raw profile files (`dataset-1.profraw`, `dataset-2.profraw`), merges their contents, converts the merged profiles to a format usable in PGO, and writes the merged data to the file `application.profile`.

NOTE The "merge" step is required even if you only have a single profile file.

A raw profile data file can be merged with an existing merged profile data file, or with multiple profile data files that have already been merged.

Step 4: Rebuild application using PGO

Enable PGO in your application builds, using the profile data generated in the previous step. For example:

```
clang++ -O3 -fprofile-instr-use=application.profile source.cc
                                                 -o application
```

NOTE PGO profiles can be used at any code optimization level, and with any other compile option (Section 4.8.5).

4.8.2 Instrumentation-based profile gen with Android apps

In instrumentation-based PGO the collected profile data is normally written to a profile data file when the application exits. However, Android applications (APKs) typically do not have an exit mechanism. Therefore, to collect profile data while developing Android applications, use the compile option `-fprofile-instr-sync-interval` (along with the other profile- generation options).

This option directs the compiler to create a background writer thread which syncs the collected profile data to the file at a user-specified interval (expressed in milliseconds).

The following example directs the compiler to sync collected profile data to the file `/sdcard/default.profraw`, with a sync period of 1 second:

```
clang++ -O2 -fprofile-instr-generate=/sdcard/default.profraw
                    -fprofile-instr-sync-interval=1000 source.cc
```

NOTE At every sync event the collected profile data is appended to the raw profile output file. This causes the file to progressively grow in size. Raw profile files are compressed to their normal size after the usual post-processing is performed with the `llvm-profdata` tool.

Controlling profile generation

As an alternative to using `-fprofile-instr-sync-interval`, Snapdragon LLVM also provides APIs which can be used to limit profile generation to specific parts of a program.

The APIs (which must be added to the program source code) explicitly control syncing of the collected profile data to the profile data file:

- **Profile start**: `extern "C" int llvm_start_profile();`
- **Profile stop**: `extern "C" int lvm_stop_profile();`

`llvm_start_profile()` resets the profile data counters to zero, thus resetting the collected profile data.

`llvm_stop_profile()` syncs the currently-collected profile data to the file, and then resets the profile data counters to zero.

NOTE The APIs are intended for advanced users who need finer control over profile generation than is offered by `-fprofile-instr-sync-interval`.

The APIs return a value indicating success (0) or failure (-1). The most common source of failure is an inaccessible write location or disk full.

4.8.3 Sampling-based PGO

The sampling-based approach to PGO requires two external tools to set up the profile information:

- **Profile generator**: Linux `perf` profiler (perf.wiki.kernel.org)
- **Profile converter**: `autofdo` (github.com/google/autofdo)

The file format for sample-based profile information is described here:

clang.llvm.org/docs/UsersManual.html#sample-profile-format

Any profile generator or converter tool that can work with this file format can be used instead of the tools listed above.

NOTE Sample-based profiling has less runtime overhead than instrumentation-based profiling. However, its effectiveness tends to be directly proportional to the number of samples collected. Thus, obtaining more accurate sampled profile information requires collecting larger amounts of sampled profile data.

The following procedure explains how to use Linux `perf` and `autofdo` to perform sampling-based PGO:

Step 1: Build the application

Build the application code with the compile option `-gline-tables-only`. For example:

```
clang++ -gline-tables-only -O2 source.cc -o application
```

NOTE The application must be compiled with `-gline-tables-only` (or `-g`) to ensure that the profile information maps accurately back to the source code.

Step 2: Generate profile information

Use the profile generator `perf` to collect the profile information. For example:

```
perf record -e cycles -c 10000 ./application
```

This command generates a profile data file named `perf.data`.

NOTE On most commercial devices, installing `perf` requires root access.

Step 3: Convert profile information

Install the `autofdo` tool and convert the raw profiles into the required sample profile format. For example:

```
create_llvm_prof --binary=./application --out=application.profile
```

Step 4: Rebuild application using PGO

Enable PGO in your application build, using the profile data generated in the previous step. For example:

```
clang++ -O3 -gline-tables-only -fprofile-sample-use=
                  application.profile source.cc -o application
```

NOTE The application must be compiled with `-gline-tables-only` to ensure that the profile information maps accurately back to the source code.

Sample-based profile information can be used even as the user code changes over time (Section 4.8.5).

4.8.4 Sampling-based PGO on Snapdragon MDP

Snapdragon Mobile Development Platform (MDP) devices are targeted for application developers, and contain the latest Snapdragon processors and mobile features. MDP devices additionally include hardware and software features that specifically support application development.

Detailed information on Snapdragon MDP is presented here:

developer.qualcomm.com/mobile-development/development-devices/
mobile-development-platform-mdp

One of the MDP developer features is the collection of sample-based profiles. Normally a device must be rooted to collect sample data. However, MDP is preconfigured for this, and thus makes profile collection easy to perform using production applications.

NOTE The only additional step necessary is to add the location of `perf` to your PATH before using it.

The following procedure explains how to perform sampling-based PGO on a Snapdragon MDP:

Step 1: Build the application

Build the application code with the compile option `-gline-tables-only`:

```
clang++ -gline-tables-only -O2 source.cc -o application
```

After building the application, move the resulting binary file to the MDP.

Step 2: Generate profile information

`perf` is pre-installed on a Snapdragon MDP – you just need to add it to PATH:

```
export PATH=/data/data/com.qualcomm.qview/:$PATH
perf record -e cycles -c 10000 ./application
```

After running `perf`, move the generated profile data files back to the host.

Step 3: Convert profile information

Install the `autofdo` tool on the host and convert the raw profiles into the required sample profile format. For example:

```
create_llvm_prof --binary=./application --out=application.profile
```

Step 4: Rebuild application using PGO

Enable PGO in your application build, using the profile data generated in the previous step:

```
clang++ -O3 -gline-tables-only -fprofile-sample-use=
                  application.profile source.cc -o application
```

4.8.5 Profile resiliency

Profile information collected for PGO is associated back to the user's source code, and then used to perform PGO. As the user source code changes over time, LLVM will associate as much of the profile information with the code as it can. In cases where LLVM cannot associate the profiles back to source code, a warning message is generated and the unmappable profile information is ignored. The compiler then continues associating the profiles for the remaining parts of the user code.

LLVM profiles are thus quite resilient to changes in the source code. The user can reuse the collected application profiles over time, without needing to re-profile the application every time. LLVM will continue using the profiles as best as it can. Over time, as the user code evolves, the utility of these application profiles will degrade, and they will need to be refreshed. However, these profile refreshes are usually proportional to the scale of evolution of the application code.

4.8.6 PGO tips

- The benefits of using PGO are closely tied to the quality of the profiles collected. The profiles should reflect the workloads and user experience that you are trying to optimize performance for. Often, collecting profiles while running automated "correctness" tests for an application does not adequately exercise the hot loops. In this case, consider creating tests that specifically target what you are optimizing for. Improved performance of the final LLVM-generated binary is usually proportional to how relevant the input profiles are.
- Ensure that the profiles collected cover the different use cases and are collected over multiple runs of the same input data set (especially when using sampling-based PGO). The accuracy of sampling-based profilers tends to improve as the sample coverage increases.
- PGO has a greater impact on application performance when compiling at higher optimization levels, especially if PGO is combined with link-time optimization (LTO). With LTO profile-guided inlining is more powerful because it operates across module boundaries. With LTO profile-guided indirect call promotion is enabled. This optimization resolves the frequent targets for indirect or virtual calls, and thus improves the performance of applications with indirect or virtual calls.
- Sampling-based profiling requires using the options `-g` or `-gline-tables-only`. It helps LLVM accurately associate the generated profiles to source code.
- PGO is resilient to changes to the user's code. The profiles generated can be reused over time even as the application code changes. LLVM adjusts and uses the still-relevant profiles, while ignoring the profiles it deems outdated.
- When using instrumented PGO the linker option `-static` (which is used to build static executables) is not supported.
- Profile data generated with `-fprofile-instr-sync-interval` may include a final profile counter section which is truncated. This can result in warnings or errors while post-processing with `llvm-profdata`. In this case the messages can be ignored because all the preceding profile data sections were handled correctly by `llvm-profdata`. The post-processed output is thus valid and usable for PGO.
- When a program is compiled with `-fprofile-instr-generate`, `errno` may not be initially set to zero at the instrumented executable's startup.

4.9 Loop optimization pragmas

The compiler supports pragmas which can be used to selectively enable and disable the following loop transformations:

- Auto-vectorization

NOTE The compiler always verifies the correctness of any transformation, and will not vectorize a loop unless it can prove it is safe to do so.

4.9.1 Pragma syntax

The syntax used for loop pragmas follows the conventions used by the LLVM community.

To add a pragma to a loop, specify the pragma immediately before the target loop, using the following syntax:

```
#pragma clang loop pragma [...pragma]
```

Table 4-4 lists the supported loop pragmas – for detailed descriptions of these pragmas, see Section 4.9.3 and Section 4.9.4.

Table 4-4 Loop pragmas

Name	Description
Vectorization pragmas	
`vectorize(enable)`	Enable auto-vectorization for a loop.
`vectorize(disable)`	Disable auto-vectorization for a loop.
`vectorize_width(N)`	Enable auto-vectorization for a loop with the specified vector factor N. The vector factor is the number of iterations that will be executed in parallel. NOTE - The value N must be a power of 2.

4.9.2 Compile options

The loop pragmas for auto-vectorization take effect whenever the auto-vectorization transformations are enabled. These transformations can be enabled explicitly with a compile option (e.g., `-fvectorize-loops`) or implicitly with an optimization level (e.g., auto-vectorization is enabled at `-O3`).

As long as the corresponding transformation is enabled, no extra compile options are necessary to cause loop pragmas to take effect. To have a loop pragma take effect without enabling the transformation in general, specify the option `-floop-pragma`. For example, to vectorize only a specific loop, add the following pragma to the loop and compile the file with `-floop-pragma`:

```
#pragma clang loop vectorize(enable)
```

Table 4-5 lists the compile options that enable auto-vectorization.

Table 4-5 Loop pragma options

Name	Description
`-fvectorize-loops`	Enable auto-vectorization for all eligible loops.
`-floop-pragma`	Enable auto-vectorization for loops specified with an "enable" pragma.

The `-floop-pragma` option enables the compiler to vectorize loops with enable pragmas. Currently, `-floop-pragma` must be used to respect the enable pragmas when auto-vectorization is not otherwise enabled.

NOTE This restriction is expected to be lifted in the future so enable pragmas can be supported without the need for an additional compile option.

Table 4-6 lists the command option combinations that can enable auto-vectorization.

Table 4-6 Loop pragma option combinations

Combination	Description
`-fvectorize-loops -floop-pragma`	Enable auto-vectorization for all eligible loops.

4.9.3 Vectorization pragmas

The Snapdragon LLVM compiler supports the following vectorization pragmas:

- `#pragma clang loop vectorize(enable)`
- `#pragma clang loop vectorize(disable)`
- `#pragma clang loop vectorize_width(N)`

NOTE These are the same vectorization pragmas that are supported by the LLVM community compiler.

#pragma clang loop vectorize(enable)

Enable vectorization for a loop.

This pragma has two primary use cases:

1. Enable vectorization for a specific loop when auto-vectorization is not enabled in general.
2. Override the profitability heuristic of the auto-vectorizer.

Case 1 requires the use of the compile option `-floop-pragma` to enable the vectorizer to act on loops with enabling pragmas. Case 2 can be used to vectorize loops with constant upper bounds that would not normally be vectorized.

Safety conditions are always enforced by the compiler. The loop will not be vectorized unless the compiler can prove it is safe, regardless of the existence of the enable pragma.

NOTE Unlike the `threadify(enable)` pragma, this pragma does override the profitability conditions checked by the compiler.

#pragma clang loop vectorize(disable)

Disable vectorization for a loop.

This pragma is used to disable vectorization for a specific loop. It can be used to avoid vectorizing loops that are not profitable, or to work around bugs in the vectorizer by not vectorizing loops that are incorrectly vectorized.

#pragma clang loop vectorize_width(N)

Set vector factor used to vectorize a loop.

The vector factor determines how many iterations of a loop are done in parallel. The vector width must be a power of 2. Invalid vector widths are ignored. If the vector width is greater than the size of the vector register, the loop is unrolled until the specified vector width is reached.

For example, if the vector width is set to 16 and the vector register holds 4 elements, the loop is unrolled 4 times to achieve the requested vector width.

Setting the vector width to a value greater than 1 adds an implicit `vectorize(enable)` pragma to the loop. Setting the vector width to 1 is equivalent to using a `vectorize(disable)` pragma.

4.9.4 Reporting

The presence of a loop pragma can have an impact on what reports are generated for a loop. The compile option `-floop-pragma` has no impact on the reports generated by the auto-vectorizer when auto-vectorization is enabled. When auto-vectorization is disabled, `-floop-pragma` triggers reporting only for loops that have pragmas.

Table 4-7 shows the interaction between reporting, options, and loop pragmas. A checkmark indicates that the option is enabled (either from the command line or implicitly by the optimization level), while an `X` indicates that the option is disabled (either explicitly on the command line or by not appearing).

Table 4-7 Loop optimization reporting

-fvectorize-loops	-floop-pragma	Report Content
X	X	No reporting
X	✔	Report on vectorization results only for loops with enable pragmas
✔	X	Report vectorization results only
✔	✔	Report vectorization results for all loops
X	✔	Report vectorization results only for loops with enable pragmas
✔	X	Report vectorization results for all loops
✔	✔	Report vectorization results for all loops

Table 4-7 assumes that all report data is requested (`-fopt-reporter=all`). The reports can be further filtered using the usual mechanism of passing a specific transformation to the `-fopt-reporter` option.

A new report code has been added for loops that are explicitly disabled by a loop pragma. If the loop would otherwise be vectorized but has been disabled by a loop pragma, a "loop failed" report is generated with a "loop pragma disable" reason code.

4.9.5 Examples

This section presents a number of examples showing how to use pragmas and command options to perform loop vectorization. The examples are not exhaustive – they are intended to show how to achieve specific results.

4.9.5.1 Vectorize only a specific loop

This example demonstrates how to restrict auto-vectorization to only act on a specific loop.

Command line

```
clang -Os -floop-pragma
```

Pragma

```
#pragma clang loop vectorize(enable)
```

Example

Normally vectorization is disabled at `-Os`, but the pragma and `-floop-pragma` option ensure that the loop is vectorized.

```
void foo(int *A, int N) {
#pragma clang loop vectorize(enable)
for(int i = 0; i < N; ++i)
   A[i] += 1;
}
```

4.9.5.2 Disable vectorization of a specific loop

This example demonstrates how to disable auto-vectorization of a specific loop.

Command line

```
clang -mfpu=neon -mcpu=cortex-a57 -Ofast -fvectorize-loops
```

Pragma

```
#pragma clang loop vectorize(disable)
```

Example

The pragma ensures that the loop is not vectorized even though the `-fvectorize-loops` option is specified on the command line.

```
void foo(int *A, int N) {
#pragma clang loop vectorize(disable)
for(int i = 0; i < N; ++i)
   A[i] += 1;
}
```

4.9.5.3 Vectorize a "non-profitable" loop

The auto-vectorizer may decide that a loop is not profitable to vectorize, and disable vectorization of the loop. In this case a loop pragma can be used to specifically enable vectorization of the loop.

Command line

```
clang -mfpu=neon -mcpu=cortex-a57 -Ofast
-fvectorize-loops
```

Pragma

```
#pragma clang loop vectorize(enable)
```

Example

Enable vectorization for the inner loop. Without the option, the auto-vectorizer could decide that the loop is not profitable to vectorize.

```
void foo (int *A, int n) {
  for (int j = 0; j < n; j++) {
    int *p = A + 4*j;
#pragma clang loop vectorize(enable)
    for (int i = 0; i < 4; i++)
      p[i] += 1;
  }
}
```

4.9.5.4 Vectorize a loop with a different vector factor

The auto-vectorizer chooses a vector factor for the loop based on an internal heuristic. This can be overridden by using a loop pragma.

Command line

```
clang -mfpu=neon -mcpu=cortex-a57 -Ofast -fvectorize-loops
```

Pragma

```
#pragma clang loop vectorize_width(16)
```

Example

Auto-vectorize the loop in function `foo`, and enforce a vector factor of 16. Without the pragma, the vectorizer could choose a different vector factor.

```
void foo (int *A, int n) {
#pragma clang loop vectorize_width(16)
    for (int i = 0; i < n; i++)
      A[i] += 1;
}
```

4.10 Optimization reports

Optimization reports are a new compiler reporting mode which can be used to obtain information on why a loop is not auto-vectorized or auto-parallelized.

NOTE This feature is under development, and is subject to change in future releases. We encourage interested users to experiment with this feature and provide feedback on its usefulness.

The optimization report is a performance tool whose main purpose is to provide feedback to the user on why a loop could not be vectorized or parallelized. It is particularly useful when you have a loop you want to optimize, but the compiler optimizations are not working on the loop. Using optimization reports, you can learn why the compiler could not optimize the loop, and possibly take action to enable the desired optimization.

Using optimization reports to analyze a loop is an iterative process. There may be multiple reasons why a loop cannot be transformed. The compiler will only report the first problem it finds with the loop. After fixing the initial problem, there may be additional problems with the loop that will be reported (by recompiling the modified source code), and will need to be fixed before the loop is finally optimized.

The optimization report extends the community's LLVM optimization report for auto-vectorization and auto-parallelization optimizations. The standard LLVM options for enabling community optimization reports are described here:

clang.llvm.org/docs/UsersManual.html#options-to-emit-optimization-reports

To enable loop optimization reporting output from the compiler, specify the pass name as `loop-opt`. Two options are used to output the compiler remarks:

- `-Rpass=loop-opt` outputs the line number of the loops that were auto-parallelized and/or vectorized, depending on what optimization is enabled by the compile options.
- `-Rpass-missed=loop-opt` outputs the line number and the reason why the loop was not optimized.

4.10.1 Example output

Here is an example of an optimization report – it shows the messages a user will see when a loop is successfully vectorized.

```
$ cat t.c
void v1(int *A, int *B, int N) {
  for (int i = 0; i < N; ++i)
    A[i] += B[i];
}

$ clang -mfpu=neon -mcpu=krait -Ofast -c -g -Rpass=loop-opt  t.c
t.c:2:3: remark: Vectorized loop. [-Rpass=loop-opt]
  for (int i = 0; i < N; ++i)
  ^
```

4.10.2 Optimization report message details

This section describes the most common messages produced by the compiler. Each message description includes an example of what code triggers the message, along with potential actions a user can take to avoid the problem and vectorize the loop.

4.10.2.1 Unsupported control flow

The unsupported control flow message indicates that a loop contains control flow and cannot be vectorized. This is the most common message a user is likely to encounter. All outer and nested loops will be marked as invalid because of this reason (because they contain an inner loop, which is control flow). In many cases the control flow in an inner loop is unavoidable, but sometimes a user can rewrite the code slightly to make it friendlier for the vectorizer.

```
void foo(int *A, int *B, int N, int c, int d, int e) {
  for (int i = 0; i < N; ++i) {
    if (A[i] < c)
      B[i] += d;
    else if (A[i] > c)
      B[i] += e;
  }
}

t.c:2:8: remark: Loop body contains unsupported control flow [-
Rpass-missed=loop-opt]
  for (int i = 0; i < N; ++i) {
```

The control flow could be eliminated by the compiler if there was a store to B[i] in all cases. In this example an `else` clause can be added, which enables the compiler to remove the control flow and vectorize the loop:

```
void foo(int *A, int *B, int N, int c, int d, int e) {
  for (int i = 0; i < N; ++i) {
    if (A[i] < c)
      B[i] += d;
    else if (A[i] > c)
      B[i] += e;
    else
      B[i] = B[i];
  }
}

t.c:2:3: remark: Vectorized loop. [-Rpass=loop-opt]
  for (int i = 0; i < N; ++i) {
```

4.10.2.2 Non-affine loop bound

The loop optimizer requires all loop bounds to be affine, which is a linear function of the loop induction variable. If the loop bound is not affine – meaning that the number of iterations of the loop cannot be analyzed – then the loop is marked as invalid for optimization.

```
typedef struct S {
  int a;
  struct S *next;
} S;

int foo(S *s) {
  while (s->next != 0) {
    s->a += 1;
    s = s->next;
  }
  return 0;
}

t.c:8:5: remark: Failed to derive an affine function from the loop
bounds.
      [-Rpass-missed=loop-opt]
    s->a += 1;
    ^
```

The loop bound is non-affine because the compiler cannot analyze how many iterations the loop will execute ahead of time, because it depends on the length of the list of S structs. Contrast this case with a standard `for` loop (e.g., `for (int i = 0; i < N; ++i){...}`), where it is known that the loop will execute N times.

```
void foo(int *A, unsigned int N) {
  for (unsigned i = 0; i < N; i+=2) {
    A[i] += 1;
  }
}

t.c:3:5: remark: Failed to derive an affine function from the loop
bounds.
      [-Rpass-missed=loop-opt]
    A[i] += 1;
    ^
```

This example shows the problem of using unsigned variables for the loop index, with a non-unit step. On each iteration, the loop induction variable increases by two. Because the variable is unsigned, the C language requires that the value wrap if it reaches the max unsigned integer value. Because the variable may wrap, it is impossible for the compiler to compute how many iterations the loop may execute.

This problem can be fixed by using an int for the loop variable. Unlike unsigned ints, a plain int has undefined behavior when it wraps beyond the maximum value. The compiler can exploit this fact to assume that the value does not wrap, and compute how many times the loop executes (N/2 in this case).

```
void foo(int *A, unsigned int N) {
  for (int i = 0; i < N; i+=2) {
    A[i] += 1;
  }
}

t.c:2:3: remark: Vectorized loop. [-Rpass=loop-opt]
  for (int i = 0; i < N; i+=2) {
  ^
```

4.10.2.3 Unspecified error

This message is generated in cases where a problem cannot be easily described in terms of actionable error messages. One example of when this message is generated is from the complex control flow surrounding a loop.

```
int bar();
void foo(int *A, int N) {
  while(1) {
    while (*A < 10) {
      if (bar())
        (*A++) += 1;
      else
        break;
    }
    if (*A == 100)
      break;
  }
}
t.c:3:3: remark: Unspecified error. [-Rpass-missed=loop-opt]
  while(1) {
  ^
t.c:4:5: remark: Unspecified error. [-Rpass-missed=loop-opt]
    while (*A < 10) {
    ^
```

4.10.2.4 Non loop-invariant loop bound

This message is generated when the compiler cannot prove that the loop bound does not change during execution of the loop. The user can fix the problem by hoisting the loop bound computation out of the loop.

```
int bar(int);
void n3(int *A, int *B, int N) {
  for (int i = 0; i < bar(N); ++i)
    A[i] += B[i];
}

t.c:4:5: remark: Loop bound may change between two different loop
iterations.
      [-Rpass-missed=loop-opt]
    A[i] += B[i];
    ^
```

In this example the loop bound is computed as the return value from function bar(). The compiler cannot see the definition of bar(), so it assumes that it must be computed on each loop iteration. The fix is to hoist the call out of the loop.

```
int bar(int);
void n3(int *A, int *B, int N) {
  int Bound = bar(N);
  for (int i = 0; i < Bound; ++i)
    A[i] += B[i];
}

t.c:4:3: remark: Vectorized loop. [-Rpass=loop-opt]
  for (int i = 0; i < Bound; ++i)
  ^
```

4.10.2.5 Inst_FuncCall

This message is generated when the loop body contains a function call. You can work around the problem by inlining the function call into the loop body (if possible).

```
int inc(int);
void n5(int *A, int *B, int N) {
  for (int i = 0; i < N; ++i)
    A[i] = inc(B[i]);
}

t.c:4:12: remark: This function call cannot be handled. Try to
inline it.
      [-Rpass-missed=loop-opt]
    A[i] = inc(B[i]);
           ^
```

If the function body is known, you can either inline the definition into the loop, or add `__attribute__((always_inline))` to the function definition. Here it is assumed that `inc()` is a simple function which increments its arguments.

```
void n5(int *A, int *B, int N) {
  for (int i = 0; i < N; ++i)
    A[i] = B[i] + 1;
}

t.c:3:3: remark: Vectorized loop. [-Rpass=loop-opt]
  for (int i = 0; i < N; ++i)
  ^
```

4.10.2.6 Base pointer not loop invariant

This message indicates that a pointer used to access memory can potentially change during the execution of the loop. In order to successfully vectorize a loop, the compiler depends on having base values that do not move during the loop. The problem may not always be obvious when examining the source code, because it could be caused by potential aliasing of values in the loop.

```
typedef struct {
  int **b;
} S;
void foo(S *A, int N) {
  for (int i = 0; i < N; ++i)
    A->b[i] = 0;
}

t.c:6:5: remark: The base address of this array is not invariant
inside the loop
      [-Rpass-missed=loop-opt]
    A->b[i] = 0;
    ^
```

In this example the base value is loaded from the A struct at each iteration of the loop. The loop can be vectorized if the load of the base pointer is hoisted out of the loop.

```
typedef struct {
  int **b;
} S;
void foo(S *A, int N) {
  int **b = A->b;
  for (int i = 0; i < N; ++i)
    b[i] = 0;
}

t.c:6:3: remark: Vectorized loop. [-Rpass=loop-opt]
  for (int i = 0; i < N; ++i)
  ^
```

4.10.2.7 Non-affine memory access

This message indicates that a memory access in the loop is non-affine, meaning that it is not a linear function of the loop induction variable. Often, these accesses are the result of double indirections in the memory access, but they can also arise from non-linear arithmetic (e.g. A[i*i], A[i%n]).

```
void n4(int *A, int *B, int N) {
  for (int i = 0; i < N; ++i)
    A[B[i]] += 1;
}

t.c:3:5: remark: The array subscript of "A" is not affine [-Rpass-
missed=loop-opt]
    A[B[i]] += 1;
    ^
```

In this example the double indirection is the problem. The memory location accessed in the A array is read from the B array, which makes the access to A non-affine. If possible, the programmer should try to remove the double indirection in order to vectorize the loop.

4.10.2.8 Memory alias

This message indicates that the compiler was unable to vectorize the loop because of aliasing problems with pointers in the loop. Normally, the compiler will insert runtime checks to disambiguate the pointers to enable vectorization. However, if there are too many pointers the runtime checks will not be inserted because the checks themselves may be more costly than the benefit gained from vectorizing the loop.

The fix for this error is to increase the number of allowed runtime checks by using the option "`-mllvm -polly-max-pointer-aliasing-checks`", or by adding "`restrict`" to the pointer parameters that are passed to the function.

```
void n4(int *A, int *B, int *C, int *D, int *E, int N) {
  for (int i = 0; i < N; ++i)
    A[i] = B[i] + C[i] + D[i] + E[i] + 1;
}

t.c:3:5: remark: Accesses to the arrays "B", "C", "D", "E", "A" may
access the same memory.
      [-Rpass-missed=loop-opt]
    A[i] = B[i] + C[i] + D[i] + E[i] + 1;
    ^
```

The compiler reports an aliasing issue with the pointers in the loop. In this case the number of runtime checks can be increased using the option "`-mllvm -polly-max-pointer-aliasing-checks=5`" in order to vectorize the loop.

Alternatively, "`restrict`" could be added to the function parameters to tell the compiler that the pointers do not alias. Adding restrict is the preferred fix in this case because it avoids the overhead of runtime checks and leads to more efficient code.

```
void n4(int * restrict A, int * restrict B, int * restrict C, int *
restrict D, int * restrict E, int N) {
  for (int i = 0; i < N; ++i)
    A[i] = B[i] + C[i] + D[i] + E[i] + 1;
}

t.c:2:3: remark: Vectorized loop. [-Rpass=loop-opt]
```

5 Compiler Security Tools

5.1 Overview

The LLVM compilers support several tools and features for improving the security and reliability of program code.

This chapter covers the following topics:

- Sanitizer support
- Sanitizer special case lists
- Sanitizer usage on Android
- Sanitizer usage on Linux
- Address Sanitizer
- Data Flow Sanitizer
- Leak Sanitizer
- Memory Sanitizer
- Thread Sanitizer
- Undefined Behavior Sanitizer
- LLVM Symbolizer
- Control flow integrity
- Static program analysis

5.2 Sanitizer support

Not all sanitizers are supported on all targets. Table 5-1 shows which sanitizers are supported on which targets.

Table 5-1 Sanitizer support

Sanitizer	AARCH64-Linux	AARCH64-Android	ARM-Linux	ARM-Android
Address Sanitizer	✔	✔	✔	✔
Data flow Sanitizer	✔	X	X	X
Leak Sanitizer	✔	X	X	X
Memory Sanitizer	✔	X	X	X
Thread Sanitizer	✔	X	X	X
Undefined Behavior Sanitizer	✔	X	✔	X

5.3 Sanitizer special case lists

The behavior of the sanitizers can be controlled for certain source-level entities (such as functions) by providing a special file at compile-time. This file is called a *special case list*.

Special case lists are used to do the following things:

- Speed up time-critical functions that are already known to be correct
- Ignore functions that perform low-level operations (such as traversing thread stacks, which bypasses the stack frame boundaries)
- Ignore functions with known problems

To create a special case list, create a text file which lists the source-level entities to be ignored. Then pass this file to the compiler with the option `-fsanitize-blacklist` (Section 3.4.16).

Example case list:

```
# Disable checks in function and source file
fun:my_func
src:my_file
```

Each line in a special case list file has the following syntax:

`entity`:*`regexp`* [=*`category`*]

`entity` specifies the type of source-level entity. It has the following possible values:

- `src` – source file
- `fun` – function
- `global` – global variable (ASan only)
- `type` – class or struct type (ASan only)

`global` and `type` are specific to the Address Sanitizer. They are used to suppress error reports for out-of-bound accesses to the specified global symbols, or to instances of the specified class or struct type.

`regexp` specifies a regular expression which specifies the entity name.

`category` optionally specifies a category value to associate with the entity. Category values are specific to each sanitizer.

Empty lines and lines starting with # are ignored. The meaning of `*` in regular expression for entity names is different – it is treated as in shell wildcarding.

For example:

```
# Lines starting with # are ignored.
# Turn off checks for the source file (use absolute path or path
relative
# to the current working directory):
src:/path/to/source/file.c
# Turn off checks for a particular functions (use mangled names):
fun:MyFooBar
fun:_Z8MyFooBarv
# Extended regular expressions are supported:
fun:bad_(foo|bar)
src:bad_source[1-9].c
# Shell like usage of * is supported (* is treated as .*):
src:bad/sources/*
fun:*BadFunction*
# Specific sanitizer tools may introduce categories.
src:/special/path/*=special_sources
```

5.4 Sanitizer usage on Android

Generating an Android LLVM executable with sanitizer instrumentation requires the following items:

- The Android NDK (for its linker)
- sysroot (for building the executable)

Once the executable is built, push your executable, the sanitizer runtime library, and the LLVM Symbolizer (Section 5.12) to an Android device. The sanitizer runtime library is a shared object which must be preloaded into the executable when launched.

The shared object can be found under the LLVM release tools installation directory:

```
export INSTALL_PREFIX=LLVM_release_tools_install_dir
file $INSTALL_PREFIX/lib/clang/*/lib/linux/libclang_rt.xsan-
                                                    arm-android.so
```

NOTE *`xsan`* specifies a sanitizer library (`asan`, `msan`, etc.).

Example

Choose one of the examples that are provided in the individual sanitizer sections (for example, Section 5.6.1).

1. Build a C/C++ executable with the sanitizer instrumentation:

```
$ mkdir -p out
$ $INSTALL_PREFIX/bin/clang++ -target arm-linux-androideabi -g
-fsanitize=san_opt boom.cc -o out/boom
--sysroot=Android_ARM_sysroot
--gcc-toolchain=Android_NDK_toolchain
```

NOTE *san_opt* specifies a sanitizer option value (address, memory, etc.).

2. Push the executable, sanitizer runtime library, and symbolizer to Android device (Jellybean or later)

```
$ adb push out/boom /data/data/
$ adb push $INSTALL_PREFIX/lib/clang/*/lib/linux/
                                  libclang_rt.xsan-arm-android.so
/data/data/
$ adb push $INSTALL_PREFIX/arm-linux-androideabi/llvm-symbolizer
/data/data/
```

NOTE *xsan* specifies a sanitizer library (asan, msan, etc.).

3. Run the sanitizer-instrumented executable:

```
$ adb shell "san_path_SYMBOLIZER_PATH=/data/data/llvm-symbolizer
LD_PRELOAD=/data/data/libclang_rt.xsan-arm-android.so
/data/data/boom"
```

NOTE *san_path* and *xsan* specify a sanitizer path variable (ASAN, MSAN, etc.). and library (asan, msan, etc.).

Include the symbolizer in the argument string (as shown above) only if the sanitizer you are using requires a symbolizer to resolve the symbol names.

If the command line execution outputs the error “CANNOT LINK EXECUTABLE: could not load library”, try exporting the LD_LIBRARY_PATH:

```
adb shell "export LD_LIBRARY_PATH=/data/data/ ;
san_path_SYMBOLIZER_PATH=/data/data/llvm-symbolizer
LD_PRELOAD=/data/data/libclang_rt.xsan-arm-android.so
/data/data/boom"
```

5.5 Sanitizer usage on Linux

1. Build a C/C++ executable with sanitizer instrumentation:

```
$INSTALL_PREFIX/bin/clang++ -target arm-linux-gnueabi
--sysroot=Linux_ARM_sysroot
--gcc-toolchain=Linux_ARM_toolchain
-g
-fsanitize=san_opt boom.cc
-o boom
```

2. Run the ARM sanitizer-instrumented executable. You can run the executable on an ARM Linux system:

```
san_path_SYMBOLIZE_PATH=$INSTALL_PREFIX/arm-linux-gnueabi/
                                          llvm-symbolizer ./boom
```

NOTE *san_opt* and *san_path* specify a sanitizer option value (`address`, `memory`, etc.) and path variable (`ASAN`, `MSAN`, etc.).

Include the symbolizer (as shown above) only if the sanitizer you are using requires a symbolizer to resolve the symbol names.

5.6 Address Sanitizer

The LLVM compiler release includes a tool named *Address Sanitizer* (ASan) which can be used to detect memory errors in C and C++ code.

ASan controls checking for the following memory errors:

- Out-of-bounds accesses to heap, stack, and globals
- Use-after-free
- Use-after-return (to a certain extent)
- Double-free, invalid free
- Double-free, invalid free
- Memory leaks (experimental)

ASan is a runtime tool which requires compile-time instrumentation of the code, and a dedicated runtime library. If ASan encounters a bug during the execution of a program, it halts the execution and displays (on stderr) an error message and stack trace.

NOTE A program instrumented with ASan typically runs 2x slower.

5.6.1 Usage

To use ASan you must instrument your C/C++ code and generate an Android/Linux executable.

To instrument your C/C++ code with ASan, add the following options to both the compile and link options in LLVM:

```
-g -fsanitize=address
```

The ASan runtime library must be linked to the final executable – be sure to use `clang` (not `ld`) for the final link step.

When linking shared libraries, the ASan runtime is not linked, so `-Wl,-z,defs` may cause link errors (do not use it with ASan). To get a reasonable performance add `-O1` or higher. To get nicer stack traces in error messages, add `-fno-omit-frame-pointer`. To get perfect stack traces it may be necessary to disable inlining (just use `-O1`) and tail call elimination (`-fno-optimize-sibling-calls`).

For example:

```
% cat example_UseAfterFree.cc
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}

# Compile and link
% clang -O1 -g -fsanitize=address -fno-omit-frame-pointer
example_UseAfterFree.cc
```

Or:

```
# Compile
% clang -O1 -g -fsanitize=address -fno-omit-frame-pointer -c
example_UseAfterFree.cc
# Link
% clang -g -fsanitize=address example_UseAfterFree.o
```

If a bug is detected, the program will print an error message to stderr and exit with a non-zero exit code. ASan exits on the first detected error. This is by design:

- This approach enables ASan to produce faster and smaller generated code (both by approximately 5%).
- Fixing bugs becomes unavoidable. ASan does not produce false alarms. Once memory is corrupted the program is in an inconsistent state, which can lead to confusing results and potentially misleading subsequent reports.

5.6.2 Symbolizing the reports

To make ASan symbolize its output, you must set the `ASAN_SYMBOLIZER_PATH` environment variable to point to the `llvm-symbolizer` binary (or alternatively ensure that `llvm-symbolizer` is in your `$PATH`).

For example:

```
% ASAN_SYMBOLIZER_PATH=/usr/local/bin/llvm-symbolizer ./a.out
==9442== ERROR: AddressSanitizer heap-use-after-free on address
0x7f7ddab8c084 at pc 0x403c8c bp 0x7fff87fb82d0 sp 0x7fff87fb82c8
READ of size 4 at 0x7f7ddab8c084 thread T0
    #0 0x403c8c in main example_UseAfterFree.cc:4
    #1 0x7f7ddabcac4d in __libc_start_main ??:0
0x7f7ddab8c084 is located 4 bytes inside of 400-byte region
[0x7f7ddab8c080,0x7f7ddab8c210)
freed by thread T0 here:
    #0 0x404704 in operator delete[](void*) ??:0
    #1 0x403c53 in main example_UseAfterFree.cc:4
    #2 0x7f7ddabcac4d in __libc_start_main ??:0
previously allocated by thread T0 here:
    #0 0x404544 in operator new[](unsigned long) ??:0
    #1 0x403c43 in main example_UseAfterFree.cc:2
    #2 0x7f7ddabcac4d in __libc_start_main ??:0
==9442== ABORTING
```

5.6.3 Additional checks

ASan performs the following additional checks.

Initialization order checking

ASan can optionally detect dynamic initialization order problems, when initialization of globals defined in one translation unit uses globals defined in another translation unit. To enable this check at runtime, you should set environment variable `ASAN_OPTIONS=check_initialization_order=1`.

Memory leak detection

For more information on memory leak detection in ASan, see Section 5.8.

5.6.4 Issue suppression

ASan generally does not produce false positives, so if you see one, look again. Most likely it is a true positive.

Suppressing reports in external libraries

Runtime interposition allows ASan to find bugs in code that is not being recompiled. If you run into an issue in external libraries, we recommend immediately reporting it to the library maintainer so that it gets addressed. However, you can use the following suppression mechanism to unblock yourself and continue on with the testing. This suppression mechanism should only be used for suppressing issues in external code; it does not work on code recompiled with ASan. To suppress errors in external libraries, set the environment variable `ASAN_OPTIONS` to point to a suppression file. You can specify either the full path to the file, or the path of the file relative to the location of your executable.

For example:

```
ASAN_OPTIONS=suppressions=MyASan.supp
```

Use the following format to specify the names of the functions or libraries you want to suppress. You can see these in the error report. Remember that the narrower the scope of the suppression, the more bugs you will be able to catch.

```
interceptor_via_fun:NameOfCFunctionToSuppress
interceptor_via_fun:-[ClassName objCMethodToSuppress:]
interceptor_via_lib:NameOfTheLibraryToSuppress
```

__has_feature(address_sanitizer)

In some cases you may need to execute different code depending on whether ASan is enabled. The language extension `__has_feature` can be used for this purpose.

For example:

```
#if defined(__has_feature)
#  if __has_feature(address_sanitizer)
// code that builds only under AddressSanitizer
#  endif
#endif
```

`__has_feature` is a function-like macro which accepts a single identifier argument that is the name of a feature. It evaluates to 1 if the feature is both supported by Clang and standardized in the current language standard. If not, it evaluates to 0.

Another use of `__has_feature` is to check for compiler features not related to the language standard (such as ASan itself).

__attribute__((no_sanitize("address")))

Some code should not be instrumented by ASan. To disable the instrumentation of a particular function, use the following function attribute:

```
__attribute__((no_sanitize("address")))
```

NOTE The `no_sanitize` attribute has the deprecated synonyms `no_sanitize_address` and `no_address_safety_analysis`.

Blacklist

ASan supports the use of sanitizer special case lists to suppress error reports in the specified source files or functions (Section 5.3). Additionally, it defines the ASan-specific entity types `global` and `type` for suppressing error reports on any out-of-bound accesses to globals with certain names and types (you can only specify class or struct types).

ASan defines a the sanitizer-specific category `init`, which can be used in a case list to suppress error reports about initialization-order problems occurring in certain source files or with certain global variables.

For example:

```
# Suppress error reports for code in a file or in a function:
src:bad_file.cpp
# Ignore all functions with names containing MyFooBar:
fun:*MyFooBar*
# Disable out-of-bound checks for global:
global:bad_array
# Disable out-of-bound checks for global instances of a given class
...
type:Namespace::BadClassName
# ... or a given struct. Use wildcard to deal with anonymous
namespace.
type:Namespace2::*::BadStructName
# Disable initialization-order checks for globals:
global:bad_init_global=init
type:*BadInitClassSubstring*=init
src:bad/init/files/*=init
```

5.6.5 Suppressing memory leaks

If the Leak Sanitizer (Section 5.8) is run as part of ASan, any memory leak reports it generates can be suppressed by a separate file passed to the compiler using the environment variable `LSAN_OPTIONS`. For example:

```
LSAN_OPTIONS=suppressions=MyLSan.supp
```

The specified text file (in this case, `MyLSan.supp`) contains one or more lines of the following form:

```
leak:pattern
```

Memory leaks will be suppressed if the any of the patterns specified in this file match a function name, source file name, or library name in the symbolized stack trace of the leak report. For details see Section 5.8.

5.6.6 Limitations

- ASan uses more real memory than a native run. Exact overhead depends on the allocations sizes. The smaller the allocations you make the bigger the overhead is.
- ASan uses more stack memory. We have seen up to 3x increase.
- On 64-bit platforms ASan maps (but not reserves) 16+
- Terabytes of virtual address space. This means that tools like ulimit may not work as usually expected.
- Static linking is not supported.

5.6.7 Options

When you run your instrumented executable and ASan does not detect any errors, you will see no output. Conversely, when you see no output, it can mean either that no errors occurred, or that your executable was not instrumented with the ASan runtime.

To verify that your executable is instrumented with ASan, use the environment variable `ASAN_OPTIONS` and the flag “`verbosity=1`”. Doing this directs the ASan runtime to output a startup message when your executable is launched. For example (in Bash):

```
$ ASAN_OPTIONS=verbosity=1 ./myExe
```

If no output is generated while verbose mode is enabled, this implies your executable was not instrumented with ASan. Check that the option `-fsanitize=address` was passed to `clang` for *both* for the compilation step and the linking step.

ASan offers a variety of options for controlling the runtime behavior and enabling/disabling its functionality. For example, if you are running out of memory, set “`qualantine_size=0`”. This causes ASan to miss any use-after-free errors but still detect buffer-overflow errors. Similarly, if you are overflowing the stack, set “`redzone=0`” to save stack space. In this case you will miss buffer-overflow errors, but can still detect use-after-free errors.

You can specify multiple options by separating flags with a colon. For example:

```
$ ASAN_OPTIONS=log_path=my-asan-report:redzone=8
```

Table 5-2 lists the options supported in ASan.

Table 5-2 ASan options

Option	Default	Description
`verbosity`	0	Be more verbose (mostly for testing the tool itself)
`malloc_context_size`	30	Number of frames in malloc/free stack traces (0-256).
`redzone`	16	Size of minimal redzone.
`log_path`	stderr	Path to log files. If specified as `log_path=PATH`, every process will write error reports to `PATH.PID`.
`sleep_before_dying`	0	Sleep for the specified number of seconds before exiting the process on failure.
`quarantine_size`	256Mb	Size of quarantine (in bytes) for finding use-after-free errors. Lower values save memory but increase false negatives rate.
`exitcode`	1	Call `_exit(exitcode)` on error.
`abort_on_error`	0	If set to 1, on error call `abort()` instead of `_exit(exitcode).`
`strict_memcmp`	1	If set to 1 (default), treat `memcmp("foo", "bar", 100)` as a bug.
`alloc_dealloc_mismatch`	1	If set to 1, check for mismatches between `malloc()/new/new` and `free()/delete/delete`.
`handle_segv`	1	If set to 1, ASan installs its own handler for `SIGSEGV`.
`allow_user_segv_handler`	0	If set to 1, allows user to override `SIGSEGV` handler installed by ASan.
`check_initialization_order`	0	If set to 1, detect existing initialization order problems.
`strip_path_prefix`	""	If `strip_path_prefix=PREFIX`, remove the substring `.*PREFIX` from the reported file names.

5.6.8 Notes

The ASan runtime library does not yet demangle symbols, but the LLVM symbolizer can be used to demangle symbols (Section 5.12).

The ASan runtime library cannot be statically linked on Android. The linker does not load the `libc` symbols before any others, as it does on Linux systems. ASan relies on this feature to hijack symbols before any other shared objects are loaded. Therefore, on Android it is necessary to use the `LD_PRELOAD` trick.

For more information on ASan see:

clang.llvm.org/docs/AddressSanitizer.html

5.7 Data Flow Sanitizer

The LLVM compiler release includes a tool named *Data Flow Sanitizer* (DFSan) which can be used to perform generalized data flow analysis on C and C++ code.

Unlike the other sanitizers, DFSan is not designed to detect a specific class of bugs on its own. Instead, it provides a generic dynamic data flow analysis framework to be used by clients to help detect application-specific issues within their own code.

5.7.1 Usage

With no program changes, applying DFSan to a program will not alter its behavior. To use DFSan, the program uses API functions to apply tags to data to cause it to be tracked, and to check the tag of a specific data item. DFSan manages the propagation of tags through the program according to its data flow.

The APIs are defined in the header file `sanitizer/dfsan_interface.h`. For further information about each function, please refer to the header file.

5.7.2 ABI list

DFSan uses a list of functions known as an *ABI list* to decide whether a call to a specific function should use the operating system's native ABI, or whether it should use a variant of this ABI that also propagates labels through function parameters and return values.

The ABI list file also controls how labels are propagated in the former case. DFSan comes with a default ABI list which is intended to eventually cover the `glibc` library on Linux, but it may become necessary for users to extend the ABI list in cases where a particular library or function cannot be instrumented (for example, because it is implemented in assembly or another language that DFSan does not support) or a function is called from a library or function which cannot be instrumented.

DFSan's ABI list file uses the same format as a sanitizer special case list (Section 5.3). The pass treats every function in the uninstrumented category in the ABI list file as conforming to the native ABI. Unless the ABI list contains additional categories for those functions, a call to one of those functions will produce a warning message, as the labeling behavior of the function is unknown.

DFSan defines the sanitizer-specific categories `discard`, `functional`, and `custom` to control the sanitizer behavior:

- `discard` – To the extent that this function writes to (user-accessible) memory, it also updates labels in shadow memory (this condition is trivially satisfied for functions which do not write to user-accessible memory). Its return value is unlabelled.
- `functional` – Like discard, except that the label of its return value is the union of the label of its arguments.
- `custom` – Instead of calling the function, a custom wrapper `__dfsw_F` is called, where F is the name of the function. This function may wrap the original function or provide its own implementation. This category is generally used for uninstrumentable functions which write to user-accessible memory or which have more complex label propagation behavior. The signature of `__dfsw_F` is based on that of F with each argument having a label of type dfsan_label appended to the argument list. If F is of non-void return type a final argument of type `dfsan_label *` is appended to which the custom function can store the label for the return value.

For example:

```
void f(int x);
void __dfsw_f(int x, dfsan_label x_label);

void *memcpy(void *dest, const void *src, size_t n);
void *__dfsw_memcpy(void *dest, const void *src, size_t n,
                    dfsan_label dest_label, dfsan_label src_label,
                    dfsan_label n_label, dfsan_label *ret_label);
```

If a function defined in the translation unit being compiled belongs to the uninstrumented category, it will be compiled so as to conform to the native ABI. Its arguments will be assumed to be unlabeled, but it will propagate labels in shadow memory.

For example:

```
# main is called by the C runtime using the native ABI.
fun:main=uninstrumented
fun:main=discard
# malloc only writes to its internal data structures, not
# user-accessible memory.
fun:malloc=uninstrumented
fun:malloc=discard
# tolower is a pure function.
fun:tolower=uninstrumented
fun:tolower=functional
# memcpy needs to copy the shadow from the source to the
destination region.
# This is done in a custom function.
fun:memcpy=uninstrumented
fun:memcpy=custom
```

5.7.3 Example

The following program demonstrates label propagation by checking that the correct labels are propagated.

```
#include <sanitizer/dfsan_interface.h>
#include <assert.h>

int main(void) {
  int i = 1;
  dfsan_label i_label = dfsan_create_label("i", 0);
  dfsan_set_label(i_label, &i, sizeof(i));

  int j = 2;
  dfsan_label j_label = dfsan_create_label("j", 0);
  dfsan_set_label(j_label, &j, sizeof(j));

  int k = 3;
  dfsan_label k_label = dfsan_create_label("k", 0);
  dfsan_set_label(k_label, &k, sizeof(k));

  dfsan_label ij_label = dfsan_get_label(i + j);
  assert(dfsan_has_label(ij_label, i_label));
  assert(dfsan_has_label(ij_label, j_label));
  assert(!dfsan_has_label(ij_label, k_label));

  dfsan_label ijk_label = dfsan_get_label(i + j + k);
  assert(dfsan_has_label(ijk_label, i_label));
  assert(dfsan_has_label(ijk_label, j_label));
  assert(dfsan_has_label(ijk_label, k_label));

  return 0;
}
```

5.7.4 Notes

For more information on DFSan see:

clang.llvm.org/docs/DataFlowSanitizer.html

5.8 Leak Sanitizer

The LLVM compiler release includes a tool named *Leak Sanitizer* (LSan) which can be used to detect runtime memory leaks in C and C++ code.

LSan can be combined with the Address Sanitizer (Section 5.6) to enable both memory error and leak detection, or it can be used as a stand-alone tool.

NOTE LSan adds almost no performance overhead until the very end of the process, when an extra leak detection phase is performed.

5.8.1 Usage

To use LSan, simply build your program with the Address Sanitizer (Section 5.6).

For example:

```
$ cat memory-leak.c
#include <stdlib.h>
void *p;
int main() {
  p = malloc(7);
  p = 0; // The memory is leaked here.
  return 0;
}
% clang -fsanitize=address -g memory-leak.c ; ./a.out
==23646==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 7 byte(s) in 1 object(s) allocated from:
    #0 0x4af01b in __interceptor_malloc /projects/compiler-
rt/lib/asan/asan_malloc_linux.cc:52:3
    #1 0x4da26a in main memory-leak.c:4:7
    #2 0x7f076fd9cec4 in __libc_start_main libc-start.c:287
SUMMARY: AddressSanitizer: 7 byte(s) leaked in 1 allocation(s).
```

To use LSan in stand-alone mode, link your program with the option `-fsanitize=leak`. Be sure to use `clang` (not `ld`) for the link step, to ensure that the proper LSan runtime library is linked into the final executable.

5.8.2 Notes

For more information on LSan see:

clang.llvm.org/docs/LeakSanitizer.html

5.9 Memory Sanitizer

The LLVM compiler release includes a tool named *Memory Sanitizer* (MSan) which can be used to detect the use of uninitialized memory in C and C++ code.

MSan is a runtime tool which requires compile-time instrumentation of the code, and a dedicated runtime library. If MSan encounters a bug during the execution of a program, it halts the execution and displays (on stderr) an error message and stack trace. In addition, it may optionally display information on where the uninitialized memory was originally allocated.

5.9.1 Usage

To instrument your C/C++ code with MSan, add the following option to both the compile and link options in LLVM:

```
-fsanitize=memory
```

The MSan runtime library must be linked to the final executable – be sure to use `clang` (not `ld`) for the final link step.

When linking shared libraries, the MSan runtime is not linked, so `-Wl,-z,defs` may cause link errors (do not use it with MSan). For reasonable execution performance use `-O1` or higher. For meaningful stack traces in error messages use `-fno-omit-frame-pointer`. For perfect stack traces you may need to disable inlining (just use `-O1`) and tail call elimination (`-fno-optimize-sibling-calls`).

For example:

```
% cat umr.cc
#include <stdio.h>

int main(int argc, char** argv) {
  int* a = new int[10];
  a[5] = 0;
  if (a[argc])
    printf("xx\n");
  return 0;
}

% clang -fsanitize=memory -fno-omit-frame-pointer -g -O2 umr.cc
```

If a bug is detected, the program will print an error message to stderr and exit with a non-zero exit code:

```
% ./a.out
WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x7f45944b418a in main umr.cc:6
    #1 0x7f45938b676c in __libc_start_main libc-start.c:226
```

NOTE By default, MSan exits on the first detected error. If you find the error report hard to understand, try enabling origin tracking (Section 5.9.3).

__has_feature(memory_sanitizer)

In some cases you may need to execute different code depending on whether MSan is enabled. The language extension `__has_feature` can be used for this purpose.

For example:

```
#if defined(__has_feature)
#  if __has_feature(memory_sanitizer)
// code that builds only under MemorySanitizer
#  endif
#endif
```

`__has_feature` is a function-like macro which accepts a single identifier argument that is the name of a feature. It evaluates to 1 if the feature is both supported by Clang and standardized in the current language standard. If not, it evaluates to 0.

Another use of `__has_feature` is to check for compiler features not related to the language standard (such as MSan itself).

__attribute__((no_sanitize_memory))

Some code should not be instrumented by MSan. To disable the instrumentation of a particular function, use the following function attribute:

```
__attribute__((no_sanitize_memory))
```

To avoid false positives MSan may still instrument such functions.

Blacklist

MSan supports the use of sanitizer special case lists to suppress error reports in the specified source files or functions (Section 5.3). All "Use of uninitialized value" warnings are suppressed, and all values loaded from memory are considered fully initialized.

5.9.2 Report symbolization

MSan uses an external symbolizer to print files and line numbers in reports. Ensure that the `llvm-symbolizer` binary is in `PATH`, or set the environment variable `MSAN_SYMBOLIZER_PATH` to point to it.

5.9.3 Origin tracking

MSan can track origins of uninitialized values, similar to Valgrind's `-track-origins` option. This feature is enabled with the option `-fsanitize-memory-track-origins=2` (or simply `-fsanitize-memory-track-origins`).

Example of origin tracking (using the code from the preceding example):

```
% cat umr2.cc
#include <stdio.h>

int main(int argc, char** argv) {
  int* a = new int[10];
  a[5] = 0;
  volatile int b = a[argc];
  if (b)
    printf("xx\n");
  return 0;
}

% clang -fsanitize=memory -fsanitize-memory-track-origins=2 -fno-
omit-frame-

pointer -g -O2 umr2.cc
% ./a.out
WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x7f7893912f0b in main umr2.cc:7
    #1 0x7f789249b76c in __libc_start_main libc-start.c:226

  Uninitialized value was stored to memory at
    #0 0x7f78938b5c25 in __msan_chain_origin msan.cc:484
    #1 0x7f7893912ecd in main umr2.cc:6

  Uninitialized value was created by a heap allocation
    #0 0x7f7893901cbd in operator new[](unsigned long)
msan_new_delete.cc:44
    #1 0x7f7893912e06 in main umr2.cc:4
```

By default, MSan collects both the allocation points and all intermediate stores that the uninitialized value went through.

Origin tracking has proved to be very useful for debugging MSan reports. It slows down program execution by a factor of 1.5x-2x on top of the usual MSan slowdown, and increases memory overhead.

The option `-fsanitize-memory-track-origins=1` enables a slightly faster mode when MSan collects only allocation points and not intermediate stores.

5.9.4 Use-after-destruction detection

MSan supports use-after-destruction detection. After its destructor is invoked, an object is considered no longer readable, and using the underlying memory will lead to error reports in runtime.

To enable this feature at runtime, perform the following procedure:

1. During compilation, specify the option `-fsanitize-memory-use-after-dtor`.
2. Before running the program, set the environment variable `MSAN_OPTIONS=poison_in_dtor=1`.

NOTE This feature is experimental.

5.9.5 Handling external code

MSan requires all program code to be instrumented, including any libraries that the program depends on (even `libc`).

Failure to do this may result in the generation of false reports.

Full MSan instrumentation is very difficult to achieve. To make it easier, the MSan runtime library includes 70+ interceptors for the most common `libc` functions. This makes it possible to run MSan-instrumented programs linked with an uninstrumented version of `libc`.

5.9.6 Limitations

- MSan uses 2x more real memory than a native run, and 3x with origin tracking.
- MSan maps (but not reserves) 64 terabytes of virtual address space. This means that tools like `ulimit` may not work as expected.
- Static linking is not supported.
- Older versions of MSan (LLVM 3.7 and older) didn’t work with non-position-independent executables, and could fail on some Linux kernel versions with disabled ASLR. For more information see the LLVM documentation for older versions.

5.9.7 Notes

For more information on MSan see:

clang.llvm.org/docs/MemorySanitizer.html

5.10 Thread Sanitizer

The LLVM compiler release includes a tool named *Thread Sanitizer* (TSan) which can be used to detect data race conditions in C and C++ code.

TSan is a runtime tool which requires compile-time instrumentation of the code, and a dedicated runtime library.

If TSan encounters a bug during the execution of a program, it displays (on stderr) an error message.

TSan slows down program execution by a factor of 5x-15x, with a memory overhead of about 5x-10x.

NOTE Currently TSan symbolizes its error output using an external `addr2line` process (this will be fixed in the future).

5.10.1 Usage

To instrument your C/C++ code with TSan, add the following option to both the compile and link options in LLVM:

```
-fsanitize=thread
```

The TSan runtime library must be linked to the final executable – be sure to use `clang` (not `ld`) for the final link step.

For reasonable execution performance use `-O1` or higher. To include file names and line numbers in the generated error messages use `-g`.

For example:

```
% cat projects/compiler-rt/lib/tsan/lit_tests/tiny_race.c
#include <pthread.h>
int Global;
void *Thread1(void *x) {
  Global = 42;
  return x;
}
int main() {
  pthread_t t;
  pthread_create(&t, NULL, Thread1, NULL);
  Global = 43;
  pthread_join(t, NULL);
  return Global;
}

$ clang -fsanitize=thread -g -O1 tiny_race.c
```

If a data race is detected, the program will print an error message to stderr:

```
% ./a.out
WARNING: ThreadSanitizer: data race (pid=19219)
  Write of size 4 at 0x7fcf47b21bc0 by thread T1:
    #0 Thread1 tiny_race.c:4 (exe+0x00000000a360)

  Previous write of size 4 at 0x7fcf47b21bc0 by main thread:
    #0 main tiny_race.c:10 (exe+0x00000000a3b4)

  Thread T1 (running) created at:
    #0 pthread_create tsan_interceptors.cc:705 (exe+0x00000000c790)
    #1 main tiny_race.c:9 (exe+0x00000000a3a4)
```

__has_feature(thread_sanitizer)

In some cases you may need to execute different code depending on whether TSan is enabled. The language extension `__has_feature` can be used for this purpose.

For example:

```
#if defined(__has_feature)
#  if __has_feature(thread_sanitizer)
// code that builds only under ThreadSanitizer
#  endif
#endif
```

`__has_feature` is a function-like macro which accepts a single identifier argument that is the name of a feature. It evaluates to 1 if the feature is both supported by Clang and standardized in the current language standard. If not, it evaluates to 0.

Another use of `__has_feature` is to check for compiler features not related to the language standard (such as TSan itself).

__attribute__((no_sanitize_thread))

Some code should not be instrumented by TSan. To disable the instrumentation of a particular function, use the following function attribute:

```
__attribute__((no_sanitize_thread))
```

To avoid false positives and provide meaningful stack traces, TSan may still instrument such functions.

Blacklist

TSan supports the use of sanitizer special case lists to suppress data race reports in the specified source files or functions (Section 5.3).

NOTE Unlike functions marked with `no_sanitize_thread`, blacklisted functions are not instrumented at all. This can result in false positives due to missed synchronization via atomic operations, and missed stack frames in reports.

5.10.2 Limitations

- TSan uses more real memory than a native run. At the default settings the memory overhead is 5x plus 1Mb per thread. Settings with 3x (less accurate analysis) and 9x (more accurate analysis) overhead are also available.
- TSan maps (but does not reserve) a lot of virtual address space. This means that tools like ulimit may not work as usually expected.
- `libc/libstdc++` static linking is not supported.
- Non-position-independent executables are not supported. Therefore:
  - When compiling without `-fPIC`, `-fsanitize=thread` causes the compiler to act as though `-fPIE` had been specified.
  - When linking an executable, `-fsanitize=thread` causes the compiler to act as though `-pie` had been specified.

5.10.3 Notes

For more information on TSan see:

clang.llvm.org/docs/ThreadSanitizer.html

5.11 Undefined Behavior Sanitizer

The LLVM compiler release includes a tool named *Undefined Behavior Sanitizer* (UBSan) which can detect code whose behavior is undefined according to the C language specification.

UBSan can catch a wide variety of errors, including the following:

- Using misaligned or null pointers
- Signed integer overflow
- Conversions to, from, or between floating-point types which result in overflow

UBSan is a runtime tool which requires compile-time instrumentation of the code. It includes an optional runtime library which provides better error reporting.

If UBSan encounters code with undefined behavior during the execution of a program, it displays (on stderr) an error message, and then responds according to the type of program behavior:

- After a signed integer overflow, the program continues executing.
- After the invalid use of a null pointer, the program is halted.
- After the use of a misaligned pointer, a trap is generated.

5.11.1 Usage

To instrument your C/C++ code with UBSan, add the following option to both the compile and link options in LLVM:

```
-fsanitize=undefined
```

If you link the UBSan runtime library to the final executable, be sure to use `clang++` (not `ld`) for the final link step, to ensure that the executable is linked with the proper UBSan runtime libraries.

NOTE When using C code, you can link with `clang` instead of `clang++`.

For example:

```
% cat test.cc
int main(int argc, char **argv) {
  int k = 0x7fffffff;
  k += argc;
  return 0;
}
% clang++ -fsanitize=undefined test.cc
% ./a.out
test.cc:3:5: runtime error: signed integer overflow: 2147483647 + 1
cannot be represented in type 'int'
```

You can configure UBSan to change the following behavior:

- Enable only a subset of the regular UBSan checks.
- Define how UBSan responds to each type of undefined program behavior (either continue, halt, or trap).

For example:

```
% clang++ -fsanitize=signed-integer-overflow,null,alignment -fno-
sanitize-recover=null -fsanitize-trap=alignment
```

In this example the program will continue executing after a signed integer overflow, exit after the invalid use of a null pointer, and trap after the use of a misaligned pointer.

NOTE The trap option does not require UBSan runtime support.

5.11.2 Available checks

The checks performed by UBSan are individually controlled by option values passed to the `-fsanitize=`*`event`* option (Section 3.4.16).

Table 5-3 lists the individual checks and their option values.

Table 5-3 UBSan checks

Option Value	Description
`alignment`	Use of a misaligned pointer or creation of a misaligned reference.
`bool`	Load of a bool value which is neither TRUE nor FALSE.
`bounds`	Out-of-bounds array indexing, in cases where the array bound can be statically determined.
`enum`	Load of a value of an enumerated type which is not in the range of representable values for that enumerated type.
`float-cast-overflow`	Conversion to, from, or between floating-point types which would overflow the destination.
`float-divide-by-zero`	Floating point division by zero.
`function`	Indirect call of a function through a function pointer of the wrong type.
`integer-divide-by-zero`	Integer division by zero.
`nonnull-attribute`	Passing null pointer as a function parameter which is declared to never be null.
`null`	Use of a null pointer or creation of a null reference.
`object-size`	Attempt to use bytes which the optimizer can determine are not part of the object being accessed.
`return`	In C++, reaching the end of a value-returning function without returning a value.
`returns-nonnull-attribute`	Returning null pointer from a function which is declared to never return null.
`shift`	Shift operators where the amount shifted is greater or equal to the promoted bit-width of the left hand side or less than zero, or where the left hand side is negative. For a signed left shift, also checks for signed overflow in C, and for unsigned overflow in C++.

Table 5-3 UBSan checks (Continued)

Option Value	Description
`shift-base`	Check only left-hand side of a shift operation.
`shift-exponent`	Check only right-hand side of a shift operation.
`signed-integer-overflow`	Signed integer overflow, including all the checks added by -ftrapv, and checking for overflow in signed division (INT_MIN / -1).
`unreachable`	If control flow reaches `__builtin_unreachable`.
`unsigned-integer-overflow`	Unsigned integer overflows.
`unreachable`	If control flow reaches `__builtin_unreachable`.
`unsigned-integer-overflow`	Unsigned integer overflows.
`vla-bound`	A variable-length array whose bound does not evaluate to a positive value.
`vptr`	Use of an object whose vptr indicates that it is of the wrong dynamic type, or that its lifetime has not begun or has ended. Incompatible with **`-fno-rtti`**. Link must be performed by clang++, not clang, to make sure C++-specific parts of the runtime library and C++ standard libraries are present.
`undefined`	All of the checks listed above other than unsigned-integer-overflow.
`integer`	Checks for undefined or suspicious integer behavior (e.g. unsigned integer overflow).

5.11.3 Stack traces and report symbolization

To make UBSan print a symbolized stack trace for each error report, use the following procedure:

1. Compile with `-g` and `-fno-omit-frame-pointer` to get the proper debug information in your binary.
2. Run your program with the environment variable `UBSAN_OPTIONS=print_stacktrace=1`.
3. Ensure that the `llvm-symbolizer` binary is in `PATH`.

5.11.4 Issue suppression

UBSan generally does not produce false positives, so if you see one, look again. Most likely it is a true positive.

__attribute__((no_sanitize("undefined")))

Some code should not be instrumented by UBSan. To disable the instrumentation of a particular function, use the following function attribute:

```
__attribute__((no_sanitize_("undefined")))
```

All values of `-fsanitize=`*`event`* can be used in this attribute. For example, if your function deliberately contains possible signed integer overflow, you can use the following:

```
__attribute__((no_sanitize("signed-integer-overflow"))).
```

Blacklist

UBSan supports the use of sanitizer special case lists to suppress error reports in the specified source files or functions (Section 5.3).

Runtime suppressions

Sometimes you can suppress UBSan error reports for specific files, functions, or libraries without recompiling the code. You need to pass a path to suppression file in a `UBSAN_OPTIONS` environment variable.

```
UBSAN_OPTIONS=suppressions=MyUBSan.supp
```

You need to specify a check (Section 5.6.3) you are suppressing, along with the bug location. For example:

```
signed-integer-overflow:file-with-known-overflow.cpp
alignment:function_doing_unaligned_access
vptr:shared_object_with_vptr_failures.so
```

Several limitations apply:

- Sometimes your binary must have enough debug info and/or symbol table, so that the runtime could figure out source file or function name to match against the suppression.
- It is only possible to suppress recoverable checks. For the example above, you can additionally pass -fsanitize-recover=signed-integer-overflow,alignment,vptr, although most of UBSan checks are recoverable by default.
- Check groups (such as `undefined`) cannot be used in suppressions files. Only fine-grained checks are supported.

5.11.5 Notes

In the C language specification, *undefined behavior* is the result of performing certain erroneous operations that are not flagged with an error. Note that a single instance of undefined behavior causes *all* of a program's output to be considered unpredictable and therefore useless.

For more information on undefined behavior see:

blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

For more information on UBSan see:

clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

5.12 LLVM Symbolizer

The LLVM compiler release includes a tool named *LLVM Symbolizer*, which can be used to convert program addresses into source code locations.

Symbolizer is a command-line tool – it reads object file names and addresses from the standard input, and writes the corresponding source code locations to the standard output.

If an object file name is directly specified as a command-line argument, Symbolizer treats it as the name of the input object file, and reads only addresses from the standard input.

NOTE To perform its conversion, Symbolizer uses the symbol tables and debug info sections that are stored in the object files.

To start Symbolizer from a command line, type:

```
llvm-symbolizer options...
```

Command options are used to control the symbolizer (Section 5.12.2).

NOTE The Symbolizer normally returns 0 as a program return code. Any other code values indicate that an internal program error.

The Symbolizer is used with ASan (Section 5.6), MSan (Section 5.9), and UBSan (Section 5.11).

5.12.1 Usage

```
$ cat addr.txt
a.out 0x4004f4
/tmp/b.out 0x400528
/tmp/c.so 0x710
/tmp/mach_universal_binary:i386 0x1f84
/tmp/mach_universal_binary:x86_64 0x100000f24
$ llvm-symbolizer < addr.txt
main
/tmp/a.cc:4

f(int, int)
/tmp/b.cc:11

h_inlined_into_g
/tmp/header.h:2
g_inlined_into_f
/tmp/header.h:7
f_inlined_into_main
/tmp/source.cc:3
main
/tmp/source.cc:8
```

```
_main
/tmp/source_i386.cc:8

_main
/tmp/source_x86_64.cc:8
$ cat addr2.txt
0x4004f4
0x401000
$ llvm-symbolizer -obj=a.out < addr2.txt
main
/tmp/a.cc:4

foo(int)
/tmp/a.cc:12
```

5.12.2 Options

Symbolizer is controlled by command-line options.

Table 5-4 lists the options supported in Symbolizer.

Table 5-4 Symbolizer options

Option	Description
`-obj`	Path to object file to be symbolized.
`-functions=(none|short|linkage)`	Specify how function names are printed. `none` – Omit function name `short` – Print short function name `linkage` – Print full linkage name The default is `linkage`.
`-use-symbol-table`	Favor function names stored in the symbol table over function names in debug info sections. The default is enabled.
`-demangle`	Print demangled function names. The default is enabled.
`-inlining`	If a source code location is in an inlined function, prints all the inlined frames. The default is enabled.
`-default-arch` *`arch_name`*	If a binary contains object files for multiple architectures (e.g., it is a Mach-O universal binary), symbolize the object file for the specified architecture. The architecture name is specified as a string value. The default is an empty string. The architecture can alternatively be specified by passing the string "*`binary_name`*`:`*`arch_name`*" as part of the input (Section 5.12.1). NOTE - If an architecture is not specified in either way, addresses will not be symbolized.

5.13 Control flow integrity

Control flow integrity (CFI) is a compiler feature which is designed to abort the program upon detecting certain forms of undefined behavior that can potentially allow attackers to subvert the program's control flow.

When CFI is enabled, the program code is instrumented with fast checks for indirect calls, and hooks for a function to report violations of forward-edge control-flow integrity.

CFI is controlled with the compile options `-ffcfi` and `-fno-fcfi`. For example:

```
clang -S -emit-llvm -ffcfi -o foo.ll foo.c
```

5.13.1 Configuration

CFI must be configured to handle control-flow violations; otherwise, by default the violations are ignored.

It can also be configured to generate different types of code instrumentation. Different types of programs may execute more efficiently with different types of instrumentation.

CFI configuration is performed with the LLVM static compiler tool.

NOTE The static compiler is different from the normal LLVM compiler – it is invoked by the latter to translate LLVM bitcode into target native code.

To start the static compiler from a command line, type:

```
llc options...
```

Command options are used to control the static compiler (Section 5.13.3).

5.13.2 Usage

The following example shows how to use CFI.

1. Compile program files, enabling CFI and generating LLVM bitcode files:

```
clang -S -emit-llvm -ffcfi -o foo.ll foo.c
clang -S -emit-llvm -ffcfi -o bar.ll bar.c
```

2. Link bitcode files:

```
llvm-link -o prog.ll foo.ll bar.ll
```

3. Static-compile bitcode files, configuring CFI and generating relocatable object file:

```
llc -cfi-enforcing -cfi-type=sub -jump-table-type=simplified
                                 -filetype=obj -o prog.o prog.ll
```

4. Link relocatable object file into executable binary:

```
clang -o prog prog.o
```

5. Run CFI-instrumented executable binary:

```
./prog
```

5.13.3 Options

The LLVM static compiler is controlled by command-line options.

Table 5-4 lists the CFI options supported in the static compiler.

Table 5-5 Static compiler options

Option	Description
`-cfi-enforcing`	Enforce control-flow integrity. By default, integrity violations invoke a handler function specified with `-cfi-func-name`. If no function is specified the violation is ignored.
`-cfi-func-name=`*`name`*	Specify the handler function that is called when a CFI violation occurs. NOTE - This option is superseded by `-cfi-enforcing`.

`-cfi-type=(sub|ror|add)`	Specify the type of CFI checks to be performed.
	`sub`
	Subtract pointer from table base, then mask (default).
	`ror`
	Use rotate to check offset from table base.
	`add`
	Mask out high bits and add to aligned base.

`-jump-table-type=(single|arity|simplified|full)`	Specify the type of jump table to use for CFI instrumentation.
	`single`
	Create a single table for all functions (default).
	`arity`
	Group functions into tables by the number of arguments they receive.
	`simplified`
	Create one table per simplified function type.
	`full`
	Create one table per function type.
	
	NOTE - `simplified` is recommended, as it offers the best balance between security and robustness.

5.13.4 Handler functions

If a user-defined handler function is specified (with option `-cfi-func-name`), the function must accept two `char*` parameters.

The first parameter is a C string which will contain the name of the function where the control-flow integrity violation occurred.

The second parameter will contain the pointer that violated control-flow integrity.

The handler function must be defined as a linker symbol in order to be specified using `-cfi-func-name`.

5.13.5 Notes

For more information on CFI see:

clang.llvm.org/docs/ControlFlowIntegrity.html

For more information on the LLVM static compiler see:

llvm.org/docs/CommandGuide/llc.html

The current CFI implementation does not imply `-fsanitize` or `-flto`. Therefore you must compile each source file to bitcode using `-ffcfi`, and then compile and link the bitcode files into native code.

The LLVM Snapdragon compiler includes support for a particular CFI scheme known as *forward-edge control flow integrity*. It is supported on ARMv7 and ARMv8 (AArch32 and AArch64) targets. For more information on this scheme see:

www.pcc.me.uk/~peter/acad/usenix14.pdf

5.14 Static program analysis

The LLVM compiler release includes the following tools for performing static analysis on a program:

- Static analyzer
- Post processor
- Scan-build

The static analyzer is a source code analysis tool which finds potential bugs in C and C++ programs. It can be used to analyze individual files or entire programs.

The post processor creates a summary of the report that is generated by performing static analysis while compiling a program.

Scan-build is an additional tool for compiling and statically analyzing a program. It can be used with a Make-based build system.

5.14.1 Static analyzer

The static analyzer is a source code analysis tool which is integrated into the LLVM compiler. It analyzes a program for various types of potential bugs – including security threats, memory corruption, and garbage values – and generates a diagnostic report describing the potential bugs it detected.

The static analyzer has the following features:

- It supports more than 100 distinct *checkers* which are organized into the categories `alpha`, `core`, `cplusplus`, `debug`, and `security`
- Checkers can be selectively enabled or disabled from the command line
- Disabling a checker category disables all the checkers in that category
- Selected parts of the program code can be excluded from checking

5.14.1.1 Analyzing source files

To use the static analyzer on specific program source files, invoke the LLVM compiler on the files using the static analyzer options (Section 3.4.25). For example:

```
clang --analyze -Xclang --analyzer-output -Xclang html
                                                   -o dir files
```

`--analyze` causes the compiler to generate a static analyzer report instead of a program object file.

`"--analyzer-output html"` specifies that the report is generated in HTML format.

NOTE `-Xclang` must be used (twice) to pass the option `"--analyzer-output html"` to the compiler. For details see Section 3.4.2.

`-o` specifies the directory where the report files will be stored. (If the directory does not exist, the compiler automatically creates it.). The files are named `report*.html`.

`files` specifies the program source files to be analyzed.

Example of a diagnostic report entry:

```
// @file: test.cpp
int main() {
  int* p = new int();
  return* p;
}
warning: Potential leak of memory pointed to by 'p'
```

NOTE Each potential bug flagged in a report includes the path (i.e., control and data) needed for locating the bug in the program.

Static analyzer warnings can be converted to errors with the option `--analyzer-Werror`.

5.14.1.2 Analyzing programs

To use the static analyzer on an entire program, invoke the LLVM compiler on the program using the option `--compile-and-analyze` (Section 3.4.25). For example:

```
clang --compile-and-analyze dir input_files...
```

`--compile-and-analyze` specifies the directory where the static analyzer report will be stored. (If the directory does not exist, the compiler automatically creates it.). The report is automatically generated in HTML format. The files are named `report*.html`.

`input_files` specifies the program source files.

Statically analyzing an entire program at once (as opposed to selected source files) is recommended for the following reasons:

- The generated analysis report files are all stored in a single location.
- The command option can be passed from the build system, which helps perform the static analysis and compilation every time the program is built.
- Because build systems are good at tracking files that have changed, and compiling only the minimal set of required files, the overall turnaround time for static analysis is relatively small, making it reasonable to run the static analyzer with every build.

NOTE When using a build system, specifying the same directory name throughout the build will generate all the HTML report files in the specified directory.

The filenames generated for a report are based on hashing functions, so the report files will not be overwritten.

5.14.1.3 Managing checkers

The static analyzer supports more than 100 individual checkers which can analyze programs for various types of potential bugs. By default only a subset of these checkers is enabled, to minimize both the compile time and the generation of false positives.

To enable additional checkers, invoke the static analyzer using the option `-analyzer-checker` (Section 3.4.25). For example:

```
clang --analyze -Xclang -analyzer-output -Xclang html
                     -Xclang -analyzer-checker=NewDelete -o dir
```

`-analyzer-checker` specifies the checker to be enabled (in this case, `NewDelete`).

To disable individual checkers, invoke the static analyzer using the option `-analyzer-disable-checker` (Section 3.4.25). For example:

```
clang --analyze -Xclang -analyzer-output -Xclang html
         -Xclang -analyzer-disable-checker=NullDereference -o dir
```

NOTE `-Xclang` must be used to pass the options `-analyzer-output`, `-analyzer-checker`, and `-analyzer-disable-checker` to the compiler. For details see Section 3.4.2.

To list all the supported checkers, use the following command:

```
clang -cc1 -analyzer-checker-help
```

To list just the default checkers, use the option `-###` (Section 3.4.2).

Packages

The individual checkers are organized into the following categories:

- `alpha`
- `core`
- `cplusplus` (only for analyzing C++ programs)
- `debug`
- `security`
- `unix`

Each category (or *package*) is defined to include a number of checkers. For example, the checker `NullDereference` is a `core` checker, while `NewDelete` is a `cplusplus` checker. Organizing checkers into packages (and sub-packages) makes it easier to enable/disable specific sets of checkers.

Example of using the static analyzer with all `alpha` checkers enabled:

```
clang --analyze -Xclang -analyzer-output -Xclang html
                          -Xclang -analyzer-checker=alpha -o dir
```

Example of using the static analyzer with all `core.DivideZero` checkers disabled:

```
clang --analyze -Xclang -analyzer-output -Xclang html
    -Xclang -analyzer-disable-checker=core.DivideZero -o dir
```

NOTE The list of supported checkers includes all the supported packages and sub-packages.

Lists

To enable or disable multiple individual checkers, multiple checker and package names can be specified as a single comma-separated list. For example:

```
clang --analyze -Xclang -analyzer-output -Xclang html
                 -Xclang -analyzer-checker=alpha,core -o dir
```

5.14.1.4 Handling false positives

While checking a program for potential bugs, the static analyzer may report *false positives*, which are sections of code that the analyzer incorrectly flags as bugs.

To minimize false positives, the static analyzer by default enables a set of checkers that has been tested to identify a high percentage of actual program bugs (Section 5.14.1.3). And if necessary, additional checkers can be individually enabled.

However, despite the overall accuracy of the checkers, several cases still exist where false positives can be generated. For instance, if you enable the checker used to analyze dead code, the static analyzer will flag as a false positive any code that has been conditionally enabled for debugging purposes.

To handle such cases, the static analyzer supports several features for handling false positives:

- Special comment
- Preprocessor symbol
- Function attribute

NOTE Using comments or symbols to handle false positives is not recommended, as they make the code inaccessible to the analyzer. Instead, please report any false positives so the existing checkers can be improved to eliminate them.

For more information on false positives see clang-analyzer.llvm.org/faq.html.

Special comment

Individual lines of code can be excluded from checking by adding the following comment to the line:

```
// clang_sa_ignore [checker] [user_comment_text]
```

`checker` specifies a checker, package, or list (Section 5.14.1.3) that is excluded from being applied to the line of code. It must be enclosed in square brackets. For example:

```
g_ptr = new int(0); // clang_sa_ignore [deadcode.DeadStores]
g_ptr = new int(0); // clang_sa_ignore [alpha] my comment text
g_ptr = new int(0); // clang_sa_ignore [alpha,deadcode.DeadStores]
```

Preprocessor symbol

One or more lines of code can be conditionally excluded from all checking by using the preprocessor symbol `__clang_analyzer__`, which is automatically defined by the static analyzer. For example:

```
#ifndef __clang_analyzer__
   // Code excluded from checking
#endif
```

When using the preprocessor symbol with the static analyzer, the code must remain compilable, even though it does not need to be linkable or executable. For example, to exclude the body of a function from being analyzed, use the following conditional code:

```
#ifdef __clang_analyzer__
void noisyFunction(); // this version is for analysis only

#else // __clang_analyzer__
static void noisyFunction() {
  // function body is generating too many false positives
}

#endif // __clang_analyzer__
```

Function attribute

A common source of false positives is non-returning functions such as assert functions.

Although the static analyzer is aware of the standard library non-returning functions, if (for example) a program has its own implementation of asserts, it helps to mark them with the following function attribute:

```
__attribute__((__noreturn__))
```

Using this attribute greatly improves the static analysis diagnostics and lessens the number of false positives. For example:

```
void my_abort(const char* msg) __attribute__((__noreturn__)) {
printf("%s", msg);
exit(1);
}
```

5.14.2 Post processor

The post processor is a report generator which is implemented as a stand-alone script. It creates a summary of the report that is generated by using the option `-compile-and-analyze` (Section 5.14.1).

The post processor is invoked with the following command:

```
post-process --report-dir dir --html-title title
```

The post processor reads all the files from the directory specified by the option `--report-dir`, and writes in the same directory a summary report file named `index.html`.

The report title is specified with the option `--html-title`.

For more information on the post processor use the command `post-process --help`.

NOTE The post processor script is stored in the directory `$INSTALL_PREFIX/bin`.

In some cases the static analyzer may generate multiple report files for the same bug. The post processor cleans up after multiple report files. For this reason it should be run regularly to keep the report directory clean.

5.14.3 Scan-build

`Scan-build` is a stand-alone tool for compiling and statically analyzing a program. It can be used with a Make-based build system (though it is recommended to instead use the LLVM static analyzer whenever possible – see Section 5.14.1).

`Scan-build` enables a user to run the static analyzer as part of regular build process. Here are two examples of invoking `scan-build`:

```
scan-build clang++ -c test.cpp

scan-build -v -k -o out-dir -disable-checker deadcode
                     -use-c++=clang++ --use-c=clang make -j8
```

`Scan-build` works well with a Make-based build system.

For more information invoke `scan-build --help`.

NOTE The `Scan-build` script is stored in the directory `$INSTALL_PREFIX/bin`.

6 Porting Code from GCC

6.1 Overview

This chapter describes issues commonly encountered while porting to LLVM an application that was previously built only with GCC.

It covers the following topics:

- Command options
- Errors and warnings
- Function declarations
- Casting to incompatible types
- `aligned` attribute
- Reserved registers
- Inline versus extern inline

NOTE For more information on GCC compatibility see Chapter 8.

6.2 Command options

LLVM supports many but not all of the GCC command options. Unsupported options are either ignored or flagged with a warning or error message: most receive warning messages.

For more information see Section 3.4.5.

6.3 Errors and warnings

LLVM enforces strict conformance to the C99 language standard. As a result, you may encounter new errors and warnings when compiling GCC code.

To handle these messages when porting to LLVM, consider the following steps:

1. Remove the command option `-Werror` if it is being used (as it converts all warnings into errors).
2. Update the code to eliminate the remaining errors and warnings.

6.4 Function declarations

LLVM enforces the C99 rules for function declarations. In particular:

- A function declared with a non-`void` return type must return a value of that type.
- A function referenced before being declared is assumed to return a value of type `int`. If the function is subsequently declared to return some other type, it will be flagged with an error.
- A function declaration with the `inline` attribute assumes the existence of a separate definition for the function, which does not include the `inline` attribute. If no such definition appears in the program, a link-time error will occur.

To satisfy these restrictions when porting to LLVM, consider the following steps:

1. Use option `-Wreturn-type` to generate a warning whenever a function definition does not return a value of its declared type.
2. Use `-Wimplicit-function-declaration` to generate a warning whenever a function is used before being declared.
3. Update the code to eliminate the remaining errors and warnings.

For more information on inlining see http://clang.llvm.org/compatibility.html#inline.

A discussion of different inlining approaches can be found at http://www.greenend.org.uk/rjk/tech/inline.html.

6.5 Casting to incompatible types

LLVM enforces the C99 rules for *strict aliasing*.

In the C language, two pointers that reference the same memory location are said to *alias* one another. Because any store through an aliased pointer can potentially modify the data referenced by one of its pointer aliases, pointer aliases can limit the compiler's ability to generate optimized code.

In strict aliasing, pointers to different types are prevented from being aliased with one another. The compiler flags pointer aliases with an error message.

Note that strict aliasing has a few exceptions:

- Any pointer type can be cast to `char*` or `void*`.
- A `char*` or `void*` can be cast to any pointer type.
- Pointers to types that differ only by signedness (e.g., `int` versus `unsigned int`) can be aliased.

To satisfy strict aliasing when porting to LLVM, consider the following steps:

1. Use option `-Wcast-align` to generate a warning whenever a pointer alias is detected.
2. Update the code to eliminate the resulting warnings.

NOTE Dereferencing a pointer that is cast from a less strictly aligned type has undefined behavior.

6.6 aligned attribute

LLVM does not allow the `aligned` attribute to appear inside the `__alignof__` operator.

To satisfy this restriction when porting to LLVM, create a typedef with the `aligned` attribute. For example:

```
typedef unsigned char u8;
#ifdef __llvm__
    typedef u8 __attribute((aligned)) aligned_u8;
#endif
unsigned int foo()
{
#ifndef __llvm__
    return __alignof__(u8 __attribute__ ((aligned)));
#else
    return __alignof__(aligned_u8);
#endif
}
```

6.7 Reserved registers

LLVM does not support the GCC extension to place global variables in specific registers.

To satisfy this restriction when porting to LLVM, use the equivalent LLVM intrinsics whenever possible. For example:

```
#ifndef __llvm__
    register unsigned long current_frame_pointer asm("r11");
#endif
…
#ifndef __llvm__
    fp = current_frame_pointer;
#else
    fp = (unsigned long)__builtin_frame_address(0);
#endif
```

6.8 Inline versus extern inline

LLVM conforms to the C99 language standard, which defines different semantics for the `inline` keyword than GCC. For example, consider the following code:

```
inline int add(int i, int j) { return i + j; }

int main() {
  int i = add(4, 5);
  return i;
}
```

In C99 the function attribute `inline` specifies that a function's definition is provided only for inlining, and that another definition (without the `inline` attribute) is specified elsewhere in the program.

This implies that the above example is incomplete, because if `add()` is not inlined (for example, when compiling without optimization), then `main()` will include an unresolved reference to that other function definition. This will result in the following link-time error:

```
Undefined symbols:
  "_add", referenced from:   _main in cc-y1jXIr.o
```

By contrast, GCC's default behavior follows the GNU89 dialect, which is based on the C89 language standard. C89 does not support the `inline` keyword; however, GCC recognizes it as a language extension, and treats it as a hint to the optimizer.

There are several ways to fix this problem:

- Change `add()` to a static inline function. This is usually the right solution if only one translation unit needs to use the function. Static inline functions are always resolved within the translation unit, so it will not be necessary to add a non-inline definition of the function elsewhere in the program.
- Remove the `inline` keyword from this definition of `add()`. The `inline` keyword is not required for a function to be inlined, nor does it guarantee that it will be. Some compilers ignore it completely. LLVM treats it as a mild suggestion from the programmer.
- Provide an external (non-inline) definition of `add()` somewhere else in the program. Note that the two definitions *must* be equivalent.
- Compile with the GNU89 dialect by adding `-std=gnu89` to the set of LLVM options. This approach is not recommended.

7 Coding Practices

7.1 Overview

This chapter describes recommended coding practices for users of the LLVM compilers. These practices typically result in the compiler generating more optimized code.

This chapter covers the following topics:

- Use `int` types for loop counters
- Mark function arguments as `restrict` (if possible)
- Do not pass or return structures by value
- Avoid using inline assembly

7.2 Use int types for loop counters

Using an `int` type for loop counters is strongly recommended and results in the compiler generating more efficient code. If the code uses a non-`int` type, then the compiler will have to insert zero and sign-extensions to abide by C rules. For example, the following code is *not* recommended:

```
extern int A[30], B[30];
for (short int ctr = 0; ctr < 30; ++ctr) {
    A[ctr] = B[ctr] + 55;
}
```

Use this code instead:

```
extern int A[30], B[30];
for (int ctr = 0; ctr < 30; ++ctr) {
    A[ctr] = B[ctr] + 55;
}
```

7.3 Mark function arguments as restrict (if possible)

LLVM supports the `restrict` keyword for function arguments. Using `restrict` on a pointer passed in as a function argument indicates to the compiler that the pointer will be used exclusively to dereference the address it points at. This allows the compiler to enable more aggressive optimizations on memory accesses.

NOTE When using the `restrict` keyword, you must ensure that the restrict condition holds for all calls made to that function. If an argument is erroneously marked as `restrict`, the compiler may generate incorrect code.

7.4 Do not pass or return structs by value

It is strongly recommended that structs get passed to (and returned from) functions by reference and not by value.

If a struct is passed to a function by value, the compiler must generate code which makes a copy of the struct during application runtime. This can be extremely inefficient, and will reduce the performance of the compiled code. For this reason, it is recommended that structs be passed by pointer.

For instance, the following code is inefficient:

```
struct S {
   int z;
   int y[50];
   char *x;
   long int w[40];
};

int bar(struct S arg1) {
   …
}

int baz() {
  struct S s;
   …
   bar(s);
}
```

While this code is much more efficient:

```
struct S {
   int z;
   int y[50];
   char *x;
   long int w[40];
};

int bar(struct *S arg1) {
  /* Access z here using ‘arg1->z’ (instead of ‘arg1.z’) */
  …
}

int baz() {
  struct S s;
   …
   bar(&s);
}
```

Alternatively, in C++, the efficient code can be simplified by using reference parameters:

```
struct S {
   int z;
   int y[50];
   char *x;
   long int w[40];
};

int bar(struct &S arg1) {
   …
}

int baz() {
  struct S;
   … populate elements of S …
   bar(S);
}
```

7.5 Avoid using inline assembly

Using inline assembly snippets in C files is strongly discouraged for two reasons:

- Inline assembly snippets are extremely difficult to write correctly. For instance, omitting the input, output, or clobber parameters frequently leads to incorrect code. The resulting failure can be extremely difficult to debug.
- Inline assembly is not portable across processor versions. If you need to emit a specific assembly instruction, it is recommended to use a compiler intrinsic instead of inline assembly.

Intrinsics are easy to insert in a C file, and are portable across processor versions. If intrinsics are insufficient, then you should add a new function written in assembly which contains the desired functionality. The assembly function should be called from C code.

8 Language Compatibility

8.1 Overview

LLVM strives to both conform to current language standards, and to implement many widely-used extensions available in other compilers, so that most correct code will "just work" when compiled with LLVM. However, LLVM is more strict than other popular compilers, and may reject incorrect code that other compilers allow.

This chapter describes common compatibility and portability issues with LLVM to help you understand and fix the problem in your code when LLVM emits an error message.

It covers the following topics:

- C compatibility
- C++ compatibility

8.2 C compatibility

This section describes common compatibility and portability issues with LLVM C. It covers the following topics:

- Differences between various standard modes
- GCC extensions not implemented yet
- Intentionally unsupported GCC extensions
- Lvalue casts
- Jumps to within `__block` variable scope
- Non-initialization of `__block` variables
- Inline assembly

8.2.1 Differences between various standard modes

LLVM supports the `-std` option, which changes what language mode LLVM uses. The supported modes for C are c89, gnu89, c94, c99, gnu99, c11, and various aliases for those modes. If no `-std` option is specified, LLVM defaults to gnu99 mode.

The c* and gnu* modes have the following differences:

- c* modes define `__STRICT_ANSI__`.
- Target-specific defines not prefixed by underscores (such as “`linux`”) are defined in gnu* modes.
- Trigraphs default to being off in gnu* modes; they can be enabled by the `-trigraphs` option.
- The parser recognizes `asm` and `typeof` as keywords in gnu* modes; the variants `__asm__` and `__typeof__` are recognized in all modes.
- Arrays that are VLA's according to the standard, but which can be constant folded by the compiler front end are treated as fixed size arrays. This occurs for things such as “`int X[(1, 2)];`”, which is technically a VLA. c* modes are strictly compliant and treat these as VLAs.
- The Apple “blocks” extension is recognized by default in gnu* modes on some platforms. It can be enabled in any mode with the `-fblocks` option.

The *99 and *11 modes have the following differences:

- Warnings for use of C11 features are disabled.
- `__STDC_VERSION__` is defined to 201112L rather than 199901L.

The *89 and *99 modes have the following differences:

- The *99 modes default to implementing `inline` as specified in C99, while the *89 modes implement the GNU version. This can be overridden for individual functions with the `__gnu_inline__` attribute.
- Digraphs are not recognized in c89 mode.
- The scope of names defined in a `for`, `if`, `switch`, `while`, or `do` statement is different. (example: "`if ((struct x {int x;}*)0) {}`".)
- `__STDC_VERSION__` is not defined in *89 modes.
- `inline` is not recognized as a keyword in c89 mode.
- `restrict` is not recognized as a keyword in *89 modes.
- Commas are allowed in integer constant expressions in *99 modes.
- Arrays which are not lvalues are not implicitly promoted to pointers in *89 modes.
- Some warnings are different.

c94 mode is identical to c89 mode except that digraphs are enabled in c94 mode.

8.2.2 GCC extensions not implemented yet

LLVM tries to be compatible with GCC as much as possible, but the following GCC extensions are not yet implemented in LLVM:

- **`#pragma weak`** – This is likely to be implemented at some point in the future, at least partially.
- **Decimal floating (`_Decimal32`, etc.) and fixed-point types (`_Fract`, etc.)** – No one has expressed interest in these yet, so it is currently unclear when they will be implemented.
- **Nested functions** – This is a complex feature which is infrequently used, so it is unlikely to be implemented anytime soon.
- **Global register variables** – This is unlikely to be implemented soon as it requires additional LLVM back end support.
- **Static initialization of flexible array members** – This appears to be a rarely used extension, but could be implemented pending user demand.
- **`__builtin_va_arg_pack and __builtin_va_arg_pack_len`** – This is used rarely, but in some potentially interesting places such as the `glibc` headers, so it may be implemented pending user demand. Note that because LLVM pretends to be like GCC 4.2, and this extension was introduced in 4.3, the `glibc` headers will currently not try to use this extension with LLVM.
- **Forward-declaring function parameters** – This has not showed up in any real-world code yet, though, so it might never be implemented.

8.2.3 Intentionally unsupported GCC extensions

LLVM intentionally does not implement the following GCC extensions:

- **Variable-length arrays in structures** – This is not implemented for several reasons: it is tricky to implement, the extension is completely undocumented, and the extension appears to be rarely used. Note that LLVM *does* support flexible array members (arrays with a zero or unspecified size at the end of a structure).
- **An equivalent to GCC's "fold"** – This implies that LLVM does not accept some constructs GCC might accept in contexts where a constant expression is required, such as "x-x" where x is a variable.
- **`__builtin_apply` and related attributes** – This extension is extremely obscure and difficult to implement reliably.

8.2.4 Lvalue casts

Old versions of GCC permit casting the left-hand side of an assignment to a different type. LLVM produces an error for code like this:

```
lvalue.c:2:3: error: assignment to cast is illegal, lvalue casts
are not supported
(int*)addr = val;
^~~~~~~~~~ ~
```

To fix this problem, move the cast to the right-hand side. In this example, one could use:

```
 addr = (float *)val;
```

8.2.5 Jumps to within __block variable scope

LLVM disallows jumps into the scope of a `__block` variable. Variables marked with __block require special runtime initialization. A jump into the scope of a `__block` variable bypasses this initialization, leaving the variable's metadata in an invalid state.

Consider the following code fragment:

```
int fetch_object_state(struct MyObject *c) {
  if (!c->active) goto error;

  __block int result;
  run_specially_somehow(^{ result = c->state; });
  return result;

 error:
  fprintf(stderr, "error while fetching object state");
  return -1;
}
```

GCC accepts this code, but produces code that will usually crash when the result goes out of scope if the jump is taken. (It's possible for this bug to go undetected, because it often will not crash if the stack is fresh – i.e., is still zeroed.) Therefore, LLVM rejects this code with a hard error:

```
t.c:3:5: error: goto into protected scope
    goto error;
    ^
t.c:5:15: note: jump bypasses setup of __block variable
  __block int result;
              ^
```

The fix is to rewrite the code to not require jumping into a `__block` variable's scope; for example, by limiting that scope:

```
{
  __block int result;
  run_specially_somehow(^{ result = c->state; });
  return result;
}
```

8.2.6 Non-initialization of __block variables

In the following example code, the variable `x` is used before it is defined:

```
int f0() {
  __block int x;
  return ^(){ return x; }();
}
```

By an accident of implementation, GCC and `llvm-gcc` unintentionally always zero any initialized `__block` variables. However, any program that depends on this behavior is relying on unspecified compiler behavior. Programs must explicitly initialize all local block variables before they are used, as with other local variables.

LLVM does not zero-initialize local block variables – thus any programs that rely on such behavior will most likely break when built with LLVM.

8.2.7 Inline assembly

In general, LLVM is highly compatible with the GCC inline assembly extensions, allowing the same set of constraints, modifiers and operands as GCC inline assembly.

8.3 C++ compatibility

This section describes common compatibility and portability issues with LLVM C++. It covers the following topics:

- Deleted special member functions
- Variable-length arrays
- Unqualified lookup in templates
- Unqualified lookup into dependent bases of class templates
- Unqualified lookup into dependent bases of class templates
- Incomplete types in templates
- Templates with no valid instantiations
- Default initialization of const variable of a class type
- Parameter name lookup

8.3.1 Deleted special member functions

In C++11, the explicit declaration of a move constructor, or a move assignment operator within a class, deletes the implicit declaration of the copy constructor and copy assignment operator. This change occurred fairly late in the C++11 standardization process, so early implementations of C++11 (including LLVM before 3.0, GCC before 4.7, and Visual Studio 2010) do not implement this rule, leading them to accept the following ill-formed code:

```
struct X {
  X(X&&); // deletes implicit copy constructor:
  // X(const X&) = delete;
};

void f(X x);
void g(X x) {
  f(x); // error: X has a deleted copy constructor
}
```

This affects some early C++11 code, including Boost's popular `shared_ptr`, up to version 1.47.0. The fix for Boost's `shared_ptr` is described here:

svn.boost.org/trac/boost/changeset/73202

8.3.2 Variable-length arrays

GCC and C99 allow an array's size to be determined at run time. This extension is not permitted in standard C++. However, LLVM supports such variable length arrays in very limited circumstances for compatibility with GNU C and C99 programs:

- The element type of a variable length array must be a "plain old data" (POD) type, which means that it cannot have any user-declared constructors or destructors, any base classes, or any members of non-POD type. All C types are POD types.
- Variable length arrays cannot be used as the type of a non-type template parameter.

If your code uses variable length arrays in a manner that LLVM does not support, several ways are available to fix your code:

1. Replace the variable length array with a fixed-size array if you can determine a reasonable upper bound at compile time; sometimes this is as simple as changing `int size = ...;` to `const int size =  ...;` (if the initializer is a compile-time constant);
2. Use `std::vector` or some other suitable container type; or
3. Allocate the array on the heap instead using `new Type[]` – just remember to `delete[]` it.

8.3.3 Unqualified lookup in templates

Some versions of GCC accept the following invalid code:

```
template <typename T> T Squared(T x) {
  return Multiply(x, x);
}

int Multiply(int x, int y) {
  return x * y;
}

int main() {
  Squared(5);
}
```

LLVM flags this code with the following messages:

```
my_file.cpp:2:10: error: call to function 'Multiply' that is
neither visible in the template definition nor found by argument-
dependent lookup

   return Multiply(x, x);
          ^

my_file.cpp:10:3: note: in instantiation of function template
specialization 'Squared<int>' requested here

  Squared(5);
  ^

my_file.cpp:5:5: note: 'Multiply' should be declared prior to the
call site

  int Multiply(int x, int y) {
      ^
```

The C++ standard states that unqualified names such as “`Multiply`” are looked up in two ways:

- First, the compiler performs an *unqualified lookup* in the scope where the name was written. For a template, this means the lookup is done at the point where the template is defined, not where it's instantiated. Because `Multiply` has not been declared yet at this point, unqualified lookup will not find it.
- Second, if the name is called like a function, then the compiler also does *argument-dependent lookup* (ADL). In ADL the compiler looks at the types of all the arguments to the call. When it finds a class type, it looks up the name in that class's namespace; the result is all the declarations it finds in those namespaces, plus the declarations from unqualified lookup. However, the compiler does not do ADL until it knows all the argument types.

In the example code above, `Multiply` is called with dependent arguments, so ADL isn't done until the template is instantiated. At that point the arguments both have type `int`, which does not contain any class types, and so ADL does not look in any namespaces. Since neither form of lookup found the declaration of `Multiply`, the code does not compile.

Here's another example, this time using overloaded operators, which obey very similar rules.

```
#include <iostream>

template<typename T>

void Dump(const T& value) {
  std::cout << value << "\n";
}

namespace ns {
  struct Data {};
}

std::ostream& operator<<(std::ostream& out, ns::Data data) {
  return out << "Some data";
}

void Use() {
  Dump(ns::Data());
}
```

Again, LLVM flags this code with the following messages:

```
my_file2.cpp:5:13: error: call to function 'operator<<' that is
neither visible in the template definition nor found by argument-
dependent lookup

  std::cout << value << "\n";
            ^

my_file2.cpp:17:3: note: in instantiation of function template
specialization 'Dump<ns::Data>' requested here

  Dump(ns::Data());
  ^

my_file2.cpp:12:15: note: 'operator<<' should be declared prior to
the call site or in namespace 'ns'

  std::ostream& operator<<(std::ostream& out, ns::Data data) {
                ^
```

Just as before, unqualified lookup did not find any declarations with the name `operator<<`. Unlike before, the argument types both contain class types:

- One of them is an instance of the class template type `std::basic_ostream`
- The other is the type `ns::Data` that is declared in the example above

Therefore, ADL will look in the namespaces `std` and `ns` for an `operator<<`. Because one of the argument types was still dependent during the template definition, ADL is not done until the template is instantiated during `Use`, which means that the `operator<<` it should find has already been declared. Unfortunately, it was declared in the global namespace, not in either of the namespaces that ADL will look in!

Two ways exist to fix this problem:

1. Make sure the function you want to call is declared before the template that might call it. This is the only option if none of its argument types contain classes. You can do this either by moving the template definition, or by moving the function definition, or by adding a forward declaration of the function before the template.
2. Move the function into the same namespace as one of its arguments so that ADL applies.

8.3.4 Unqualified lookup into dependent bases of class templates

Some versions of GCC accept the following invalid code:

```
template <typename T> struct Base {
  void DoThis(T x) {}
  static void DoThat(T x) {}
};

template <typename T> struct Derived : public Base<T> {
  void Work(T x) {
    DoThis(x); // Invalid!
    DoThat(x); // Invalid!
  }
};
```

LLVM correctly rejects this code with the following errors (when `Derived` is eventually instantiated):

```
my_file.cpp:8:5: error: use of undeclared identifier 'DoThis'

  DoThis(x);
  ^
  this->

my_file.cpp:2:8: note: must qualify identifier to find this
declaration in dependent base class

void DoThis(T x) {}
     ^

my_file.cpp:9:5: error: use of undeclared identifier 'DoThat'

DoThat(x);
^
this->

my_file.cpp:3:15: note: must qualify identifier to find this
declaration in dependent base class

static void DoThat(T x) {}
```

As noted in Section 8.3.3, unqualified names such as `DoThis` and `DoThat` are looked up when the template `Derived` is defined, not when it's instantiated. When looking up a name used in a class, we usually look into the base classes. However, we can't look into the base class `Base<T>` because its type depends on the template argument `T`, so the standard says we should just ignore it.

The fix, as LLVM indicates, is to tell the compiler that we want a class member by prefixing the calls with `this->`:

```
void Work(T x) {
 this->DoThis(x);
 this->DoThat(x);
}
```

Alternatively, you can tell the compiler exactly where to look:

```
void Work(T x) {
  Base<T>::DoThis(x);
  Base<T>::DoThat(x);
}
```

This works whether the methods are static or not, but be careful: if `DoThis` is virtual, calling it this way will bypass virtual dispatch!

8.3.5 Incomplete types in templates

The following code is invalid, but compilers are allowed to accept it:

```
class IOOptions;

template <class T> bool read(T &value) {
  IOOptions opts;
  return read(opts, value);
}

class IOOptions { bool ForceReads; };
bool read(const IOOptions &opts, int &x);
template bool read<>(int &);
```

The standard says that types which don't depend on template parameters must be complete when a template is defined if they affect the program's behavior. However, the standard also says that compilers are free to not enforce this rule. Most compilers enforce it to some extent; for example, it would be an error in GCC to write `opts.ForceReads` in the code above. In LLVM, the decision to enforce the rule consistently provides a better experience, but unfortunately it also results in some code getting rejected that other compilers accept.

8.3.6 Templates with no valid instantiations

The following code contains a typo: the programmer meant `init()` but wrote `innit()` instead.

```
template <class T> class Processor {
  ...
  void init();
  ...
};

...

template <class T> void process() {
  Processor<T> processor;
  processor.innit();      // <-- should be 'init()'
  ...
}
```

Unfortunately, the compiler can't flag this mistake as soon as it detects it: inside a template, we're not allowed to make assumptions about "dependent types" such as `Processor<T>`. Suppose that later on in this file the programmer adds an explicit specialization of `Processor`, like so:

```
template <> class Processor<char*> {
  void innit();
};
```

Now the program will work – but only if the programmer ever instantiates `process()` with `T = char*`! This is why it's hard, and sometimes impossible, to diagnose mistakes in a template definition before it's instantiated.

The standard states that a template with no valid instantiations is ill-formed. LLVM tries to do as much checking as possible at definition-time instead of instantiation-time: not only does this produce clearer diagnostics, but it also substantially improves compile times when using pre-compiled headers. The downside to this philosophy is that LLVM sometimes fails to process files because they contain broken templates that are no longer used. The solution is simple: since the code is unused, just remove it.

8.3.7 Default initialization of const variable of a class type

The default initialization of a `const` variable of a class type requires a user-defined default constructor.

If a `class` or `struct` has no user-defined default constructor, C++ does not allow you to default-construct a `const` instance of it. For example:

```
class Foo {
public:
  // The compiler-supplied default constructor works fine, so we
  // don't bother with defining one.
  ...
}
void Bar() {
  const Foo foo; // Error!
  ...
}
```

To fix this, you can define a default constructor for the class:

```
class Foo {
public:
  Foo() {}
  ...
};

void Bar() {
  const Foo foo; // Now the compiler is happy.
  ...
}
```

8.3.8 Parameter name lookup

Due to a bug in its implementation, GCC allows the redeclaration of function parameter names within a function prototype in C++ code, e.g.

```
void f(int a, int a);
```

LLVM diagnoses this error (where the parameter name has been redeclared). To fix this problem, rename one of the parameters.

A Acknowledgements

We would like to thank the LLVM community for their many contributions to the LLVM Project.

This document includes content derived from the LLVM Project documentation under the terms of the LLVM Release License:

llvm.org/releases/3.8.0/LICENSE.TXT