Qualcomm Technologies, Inc.

Qualcomm® Snapdragon™ LLVM Arm Compiler

Internal User Guide

80-VB419-96 M

January 31, 2022

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

Revision history

Rev.	Date	Changes
A	Feb 2014	Initial revision
B	Aug 2014	Added Symbolizer and UBSan; numerous misc changes
C	Mar 2015	Added PGO, loop pragmas, optional reports, ASan, MSan, CFI, numerous miscellaneous changes
D	Sep 2015	Updated PGO, LTO, and auto-vectorization; added MARE library, adaptive execution, and resource analyzer; miscellaneous changes
E	Mar 2016	Updated to support LLVM 3.8; updated existing sanitizers; added new sanitizers; renamed MARE as SYMPHONY; added -O4; miscellaneous changes
F	Oct 2016	Updated to support LLVM 3.9; added LLVM 3.9 warning messages; updated compiler, assembler, and profiler options; added chapter for bare metal environment; added support for C++ compatibility; miscellaneous changes
G	Jun 2017	Updated to support LLVM 4; added Section 6.7 *Porting from ARM Compiler toolchain*; miscellaneous changes
H	Mar 2018	Added LLVM 6 warning messages and profile-guided optimizations; updated to support ThinLTO, bare metal environments, and static analyzer; updated Section 5.7.2 *Instrumentation-based profile generation with Android apps*; miscellaneous edits and changes
J	Dec 2018	Added LLVM 8 warning messages; updated to support some specific optimizations and vectorization loop pragmas; miscellaneous edits and changes
K	Nov 2019	Added LLVM 10 warning messages and Armv8.x extensions and features
L	Nov 2020	Added support for c++17 and gnu++17; added LLVM 12 warning messages; updated Armv8.x features; miscellaneous edits and changes
M	Jan 2022	Editorial changes

NOTE: There is no Rev. I, O, Q, S, X, or Z per Mil. standards.

Contents

Figures

Tables

1 Introduction

This document is a reference for C/C++ programmers. It 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*.

We strongly recommend that you try using the various LLVM code optimizations to improve the performance of your program. Using only the default optimization settings might result in suboptimal performance.

1.1 Conventions

`Courier` font is used for computer text and code samples, for example, `void foo()`.

The following notation is used to define command syntax:

- Square brackets enclose optional items, for example, `[`**`label`**`]`.
- **Bold** indicates literal symbols for example, `[`**`comment`**`]`.
- The vertical bar character, `|`, indicates a choice of items.
- Parentheses enclose a choice of items for example, `(`**`add`**`|`**`del`**`)`.
- An ellipsis, `. . .` , follows items that can appear more than once.

For example:

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

Where:

- `#define` is a preprocessor directive
- *`name`* is the name of a defined symbol.
- *`parameter1`* and *`parameter2`* are macro parameters. The square brackets indicate that the second parameter is optional. The ellipsis indicates that the macro accepts more than parameters.
- `logging` is an interactive compiler command.
- **`on`** and **`off`** are bold to show that they are literal symbols. The vertical bar indicates that they are alternative parameters of the `logging` command.

1.2 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to CreatePoint, register for access or send email to qualcomm.support@qti.qualcomm.com.

2 Functional overview

The C and C++ compilers are based on the LLVM compiler framework and are collectively referred to as the LLVM compilers.

2.1 Features

The LLVM compilers offer the following features:

- ISO C conformance

  Supports the International Standards Organization (ISO) C language standard

- Compatibility

  Supports Arm 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 Arm assembly instructions in C source code

2.2 Languages

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

- C language dialects:

  K&R C, ANSI C89, ISO C90, ISO C94 (C89+AMD1), ISO C99 (+TC1, TC2, TC3), ISO C11

- C++ language dialects:

  C++98, C++11, C++14, C++17

The LLVM compilers also 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. They are marked with extension diagnostics that can be mapped to warnings, errors, or simply ignored.

The following C++17 features are not supported by the compiler:

- *ISO/IEC TS 19570:2015 C++ Extensions for Parallelism* technical specification
- `constexpr std::hardware_constructive`
- `constexpr std::hardware_destructive`

2.3 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*.

2.4 Processor versions

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

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 that 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*.

NOTE: The LLVM compilers do not define command options specifically for V56, but they do support V56 if you use the V55 command options.

2.5 LLVM versions

The LLVM compilers are based on LLVM 4.0+, as defined at llvm.org.

2.6 Internal features

Table 2-1 Compiler features available for internal use only

Feature	Description
-fparallel-num-workloads	Section 4.3.21
Adaptive execution	Section 5.10

2.7 C language and compiler references

This document does not describe the following:

- C or C++ languages
- Detailed descriptions of the code optimizations performed by LLVM

For suggested references, see Appendix B.

3 Get started

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.

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

NOTE: The commands shown in this chapter are for illustration only. For detailed information on building programs, see Chapter 4.

3.1 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`.

3.2 Compile program

Compile the program with the following command:

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

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

NOTE: When compiling with a GCC sysroot, see Section 4.4 for additional details. When compiling with a bare metal sysroot, see Section 6-1 for additional details.

3.3 Execute program

To execute the program, use the following command:

```
hello
```

The program outputs its message to 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 Chapter 4.

4 Use the compilers

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

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

4.1 Start the compilers

To start the C compiler from the command line, enter:

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

To start the C++ compiler from the command line, enter:

```
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 4.3). 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. `clang` is the name of the front-end driver for the LLVM compiler framework.

4.2 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.

Following are the input file types and the tool that processes files of each type. The compilers use the filename extension to determine how to process the file.

Table 4-1 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

All filename 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.

Following are the output file types and the tools used to generate each file type. Compiler options (Section 4.3) are used to specify the output file type.

Table 4-2 Compiler output files

File type	Default filename	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 filename).
Assembly source file	`file.s`	Each specified source file is compiled to a separate assembly source file (where `file` is the source filename).
Preprocessed C/C++ source file	`stdout`	The preprocessor output is written to the standard output.

4.3 Compiler options

The LLVM compilers can be controlled by command-line options (Section 4.1). Many of the GCC options are supported, along with options that are LLVM-specific.

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 standalone option, but rather a standard prefix that appears in many LLVM-specific option names.

Following is a quick reference of the options.

Display (see Section 4.3.1)

```
-help
-v
```

Compilation (see Section 4.3.2)

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

C dialect (See Section 4.3.3)

```
-ansi
-fblocks
-fgnu-runtime
-fgnu89-inline
-fno-asm
-fsigned-bitfields
-fsigned-char
-funsigned-char
-no-integrated-cpp
-std=(c89|gnu89|c99|gnu99|c11|gnu11)
-traditional
-Wpointer-sign
```

C++ dialect (see Section 4.3.4)

```
-cxx-isystem dir
-ffor-scope  |  -fno-for-scope
-fno-gnu-keywords
```

```
-ftemplate-depth-n
-fvisibility-inlines-hidden
-fuse-cxa-atexit
-nobuiltininc
-nostdinc++
-std=(c++98|gnu++98|c++11|gnu++11|c++14|gnu++14|c++17|gnu++17)
-Wc++0x-compat
-Wno-deprecated
-Wnon-virtual-dtor
-Woverloaded-virtual
-Wreorder
```

Warning and error messages (see Section 4.3.5)

```
c++98|c++11|c++14|c++17-compat-pedantic
cast-calling-convention
comma
constant-conversion
expansion-to-defined
-ferror-limit=n
-ferror-warn filename
float-overflow-conversion
float-zero-conversion
-ftemplate-backtrace-limit=n
-fsyntax-only
incompatible-sysroot
nonportable-include-path
nonportable-system-include-path
null-dereference
openmp-target
-pedantic  |  -Wpedantic
pedantic-core-features
-pedantic-errors
-Qunused-arguments
shadow-field-in-constructor-modified
shadow-field-in-constructor
undefined-func-template
undefined-var-template
unguarded-availability
unknown-argument
unsupported-cb
varargs
-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
-Wformat-nonliteral
-Wformat-security
-Wignored-qualifiers
-Wimplicit
-Wimplicit-function-declaration
-Wimplicit-int
-Wno-format-extra-args
-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 4.3.6)

```
-dumpmachine
-dumpversion
-feliminate-unused-debug-symbols
-time  |  -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
```

Diagnostic format (see Section 4.3.7)

```
-fcaret-diagnostics       |  -fno-caret-diagnostics
-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-template-tree
-fmessage-length=n
```

Individual warning groups (see Section 4.3.8)

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

Compiler crash diagnostics (see Section 4.3.9)

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

Linker (see Section 4.3.10)

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

Preprocessor (see Section 4.3.11)

```
-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
-H
--help
-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
-traditional-cpp
-trigraphs
-U name
-v
-version
--version
-w
-Wall
-Wcomment  |  -Wcomments
-Wendif-labels
-Werror
-Wimport
-Wsystem-headers
-Wtrigraphs
-Wundef
-Wunused-macros
```

Assembling (see Section 4.3.12)

```
-fintegrated-as  |  -fno-integrated-as
-Xassembler arg
```

Linking (see Section 4.3.13)

```
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 arg
```

Directory search (see Section 4.3.14)

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

Processor version (see Section 4.3.15)

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

Code generation (see Section 4.3.19)

```
-fchar-array-precise-tbaa  |  -fno-char-array-precise-tbaa
-femit-all-data
-femit-all-decls
-ffp-contract=(fast|on|off)
-finstrument-functions
-fmerge-functions  |  -fno-merge-functions
-fno-exceptions
-fpic
-fPIC
```

```
-fpie  |  -fPIE
-fsanitize=address           |  -fno-sanitize=address
-fsanitize=memory            |  -fno-sanitize=memory
-fsanitize=event[,event...]  |  -fno-sanitize=event[,event...]
-fsanitize=integer
-fsanitize=undefined
-fsanitize-blacklist=file    |  -fno-sanitize-blacklist
-fsanitize-memory-track-origins[=level]
-fsanitize-messages          | -fno-sanitize-messages
-fsanitize-opt-size          |  -fno-sanitize-opt-size
-fsanitize-source-loc        |  -fno-sanitize-source-loc
-fsanitize-use-embedded-rt
-fshort-enums  |  -fno-short-enums
-fshort-wchar  |  -fno-short-wchar
-ftrap-function=name
-ftrapv
-ftrapv-handler=name
-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 4.3.20)

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

Parallelization (see Section 4.3.21)

```
-fparallel
-fparallel-num-workloads=n
-fparallel-symphony
```

Optimization (see Section 4.3.22)

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

Specific optimizations (see Section 4.3.23)

```
-faggressive-unroll
-fdata-sections
-ffp-contract=[off|on|fast]
-ffunction-sections
-finline
-finline-functions
-floop-pragma
-fno-zero-initialized-in-bss
-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
-maggressive-size-opts
--param ssp-buffer-size=size
```

Math optimization (see Section 4.3.24)

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

Link-time optimization (compiler and linker) (see Section 4.3.25)

```
-flto
```

Profile-guided optimization (see Section 4.3.26)

```
-fprofile-instr-append-file
-fprofile-instr-file-sync
-fprofile-instr-file-sync-thread
```

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

Optimization reports (see Section 4.3.27)

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

Adaptive execution (see Section 4.3.28)

```
-mae  |  -mno-ae
```

Compiler security (see Section 4.3.29)

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

4.3.1 Display

-help

Displays compiler command and option summary.

-v

Displays compiler release version.

4.3.2 Compilation

-###

Prints commands used to perform the compilation.

-c

Compiles the source file, but does not link it.

-cc1

Bypasses the compiler driver and go directly to LLVM.

-ccc-print-phases

Prints the compilation stages as they occur.

-E

Preprocesses the source file only; does not compile it.

-no-canonical-prefixes

When processing a pathname:

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

-pipe

Communicates between compiler stages using pipes not temporary files.

-o *file*

Specify the name of the compiler output file.

-S

Compiles the source file, but does not assemble it.

-Wa,*arg***[,***arg***...]**

Passes the specified arguments to the assembler.

-Wl,*arg***[,***arg***...]**

Passes the specified arguments to the linker.

-Wp,*arg***[,***arg***...]**

Passes the specified arguments to the preprocessor.

-x *language*

Specifies the language of the subsequent source files specified on the command line.

-Xclang *arg*

Passes the specified argument to the compiler.

4.3.3 C dialect

-ansi

For C, supports ISO C90.

For C++, removes conflicting GNU extensions.

-fblocks

Enables the Apple block*s* extension.

-fgnu-runtime

Generates output compatible with the standard GNU Objective-C runtime.

-fgnu89-inline

Uses the gnu89 inline semantics.

-fno-asm

Does not recognize `asm`, `inline`, or `typeof` as keywords.

-fsigned-bitfields

Defines bit fields as signed.

-fsigned-char

Defines the char type as signed.

-funsigned-char

Defines the char type as unsigned.

-no-integrated-cpp

Compiles using separate preprocessing and compilation stages.

-std=(c89|gnu89|c99|gnu99|c11|gnu11)

LLVM C language mode. The default setting is `gnu11`.

-traditional

Supports pre-standard C language.

-Wpointer-sign

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

4.3.4 C++ dialect

-cxx-isystem *dir*

Adds a 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

Disables recognizing `typeof` as a keyword.

-ftemplate-depth-n

Specifies the maximum instantiation depth of a template class.

-fvisibility-inlines-hidden

Specifies the default visibility for inline C++ member functions.

-fuse-cxa-atexit

Registers destructors with function `__cxa_atexit` (instead of `atexit`). This option applies only to objects that have static storage duration.

-nobuiltininc

Disables built-in `#include` directories.

-nostdinc++

Disables standard `#include` directories for the C++ standard library.

-std=(c++98|gnu++98|c++11|gnu++11|c++14|gnu++14|c++17|gnu++17)

LLVM C++ language mode. The default setting is `gnu++14`.

-Wc++0x-compat

Generates warnings for C++ constructs with different semantics in ISO C++ 1998 and ISO C++ 200x.

-Wno-deprecated

Does not generate warnings when deprecated features are used.

-Wnon-virtual-dtor

Generates a warning when a polymorphic class is declared with a non-virtual destructor.

-Woverloaded-virtual

Generates a warning when a function hides virtual functions from a base class.

-Wreorder

Generates a warning when member initializers do not appear in the code in the required execution order.

4.3.5 Warning and error messages

c++98|c++11|c++14|c++17-compat-pedantic

Hexadecimal floating literals are incompatible with C++ standards before C++1z.

Warn if the compiled code uses experimental features added to language standards more recent than C++17.

cast-calling-convention

Cast between incompatible %0 and %1 calling conventions. Calls through this pointer might abort at runtime.

comma

Possible misuse of comma operator here.

constant-conversion

Implicit conversion from %2 to %3 changes the value from %0 to %1.

expansion-to-defined

Macro expansion producing defined has undefined behavior.

-ferror-limit=*n*

Stops emitting diagnostics after *n* errors have been produced. The default setting is 20. Disable the error limit with `-ferror-limit=0`.

-ferror-warn *filename*

Converts 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 `-Wunused-variable`, use the warning name, `unused-variable`.

You can specify this option multiple times.

NOTE: You can integrate this option (and its associated file) into a build system and use it to iteratively resolve the warning messages generated by a project.

float-overflow-conversion

Implicit conversion of out of range value from %0 to %1 changes the value from %2 to %3.

float-zero-conversion

Implicit conversion from %0 to %1 changes the non-zero value from %2 to %3.

-ftemplate-backtrace-limit=*n*

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

-fsyntax-only

Checks for syntax errors only.

incompatible-sysroot

Uses sysroot for %0 but targets %1.

nonportable-include-path

Non-portable path to file %0. The specified path differs in case from the filename on the disk.

nonportable-system-include-path

Non-portable path to file %0. The specified path differs in case from the filename on the disk.

null-dereference

Binding dereferenced NULL pointer to reference has undefined behavior.

openmp-target

Declaration is not declared in any declare target region.

-pedantic
-Wpedantic

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

pedantic-core-features

OpenCL extension %0 is a core feature or a supported optional core feature; ignoring.

-pedantic-errors

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

-Qunused-arguments

Does not generate warnings for unused driver arguments.

shadow-field-in-constructor-modified

Modifying constructor parameter %0 that shadows a field of %1.

shadow-field-in-constructor

Constructor parameter %0 shadows the field %1 of %2.

undefined-func-template

Instantiation of function %q0 is required, but no definition is available.

undefined-var-template

Instantiation of variable %q0 is required, but no definition is available.

unguarded-availability

Using * case here; platform %0 is not accounted for.

unknown-argument

Unknown argument is ignored in clang-cl: %0.

unsupported-cb

Ignoring `-mcompact-branches=` option because the %0 architecture does not support it.

varargs

Passing an object that undergoes a default argument promotion to `va_start` has undefined behavior.

-w

Suppresses all warnings.

-W*foo*

Enables the diagnostic *`foo`*.

-Wno-*foo*

Disables the diagnostic *`foo`*.

-Wall

Enables all `-W` options.

-Warray-bounds

Generates a warning if array subscripts are out of bounds.

-Wcast-align

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

-Wchar-subscripts

Generates a warning if array subscript is type `char`.

-Wcomment

Generates a warning if a comment symbol appears inside a comment.

-Wconversion

Generates a warning if an implicit conversion might alter a value.

-Wdeclaration-after-statement

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

-Wno-deprecated-declarations

Does not generate warnings for functions, variables, or types assigned the attribute `deprecated`.

-Wempty-body

Generates a warning if an `if`, `else`, or `do while` statement contains an empty body.

-Wendif-labels

Generates a warning if an `#else` or `#endif` directive is followed by text.

-Werror

Converts all warnings into errors.

-Werror=*foo*

Converts the diagnostic *foo* into an error.

-Wno-error=*foo*

Keeps the diagnostic *foo* as a warning, even if `-Werror` is used.

-Werror-implicit-function-declaration

Generates a warning or error if a function is used before being declared.

-Weverything

Enables all warnings.

-Wextra

Enables selected warning options, and generate warnings for selected events.

-Wfloat-equal

Generates a warning if two floating-point values are compared for equality.

-Wformat

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

-Wformat=2

Equivalent to specifying the following options: `-Wformat -Wformat-nonliteral -Wformat-security -Wformat-y2k`.

-Wformat-nonliteral

Generates a warning if the format string is not a string literal, except if the format arguments are passed through `va_list`.

-Wformat-security

Generates a warning for format function calls that might cause security risks.

-Wignored-qualifiers

Generates a warning if a return type has a qualifier (such as `const`).

-Wimplicit

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

-Wimplicit-function-declaration

Generates a warning if a function is used before it is declared.

-Wimplicit-int

Generates a warning if a declaration does not specify a type.

-Wno-format-extra-args

Does not generate a warning for passing extra arguments to `printf` or `scanf`.

-Wno-invalid-offsetof

Does not generate a warning if a non-POD type is passed to the `offsetof` macro.

-Wlong-long

Generates a warning if type `long long` is used.

-Wmain

Generates a warning if the `main()` function has any suspicious properties.

-Wmissing-braces

Generates a warning if an aggregate or union initializer is not properly bracketed.

-Wmissing-declarations

Generates a warning if a global function is defined without being first declared.

-Wmissing-noreturn

Generates a warning if a function does not include a return statement.

-Wmissing-prototypes

Generates a warning if a global function is defined without a prototype.

-Wno-multichar

Does not generate a warning if a multiple-character constant is used.

-Wnonnull

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

-Wpacked

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

-Wpadded

Generates a warning if the memory layout of a structure includes padding.

-Wparentheses

Generates a warning if the parentheses are omitted in certain cases.

-Wpedantic

See `-pedantic`.

-Wpointer-arith

Generates a warning if any code depends on the size of `void` or a function type.

-Wreturn-type

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

-Wshadow

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

-Wsign-compare

Generates a warning in a signed/unsigned compare operation if the result might be inaccurate due to the signed operand being converted to unsigned.

-Wswitch
-Wswitch-enum

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

-Wsystem-headers

Generates a warning for constructs declared in system header files.

-Wtrigraphs

Generates a warning if a trigraph forms an escaped newline in a comment.

-Wundef

Generates a warning if an undefined non-macro identifier appears in an `#if` directive.

-Wuninitialized

Generates a warning if referencing an uninitialized automatic variable.

-Wunknown-pragmas

Generates a warning if a `#pragma` directive is not recognized by the compiler.

-Wunreachable-code

Generates a warning if code will never be executed.

-Wunused

Specifies all of the `-Wunused` options.

-Wunused-function

Generates a 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

Generates a warning if a label is declared without being used.

-Wunused-parameter

Generates a warning if a function argument is not used in its function.

-Wunused-value

Generates a warning if the value of a statement is not subsequently used.

-Wunused-variable

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

-Wno-vectorizer-no-neon

Does not generate the warning, `Vectorization flags ignored because armv7/armv8 and neon not set`.

Vectorization requires an Armvv7 or Armvv8 target, and the NEON feature must 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, assigns 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++, generates a warning that states a string constant is being converted to `char *`.

Warning—Argument unused during compilation: -mfpu=%0

Clang generates a warning when setting `-mfpu` in addition to specifying AArch64. To avoid this warning, do not specify the `-mfpu` option when specifying AArch64.

4.3.5.1 LLVM 4.0 release

Table 4-3 LLVM 4.0 release: new warnings

Category	Description
-Wunused-command-line-argument	-fdiagnostics-show-hotness argument requires profile-guided optimization information.
-Wslash-u-filename	/UA treated as the /U option.
-Wunable-to-open-stats-file	Unable to open statistics output file A: B.
-Wprivate-module	Top-level module A in a private module map; expected a submodule of B.
-Wempty-decomposition	ISO C++1z does not allow a decomposition group to be empty.
-Wc++1z-extensions	Use of multiple declarators in a single using declaration is a C++1z extension.
-Wc++98-c++11-c++14-compat	Initialization statements are incompatible with C++ standards before C++1z.
-Wignored-pragma-intrinsic	A is not a recognized built-in intrinsic; consider including <intrin.h> to access non-built-in intrinsics.
-Wignored-pragmas	Ignored. Expected *enable*, *disable*, *begin*, or *end*.
-Wmax-unsigned-zero	Call to function without interrupt attribute could clobber VFP registers of the interruptee.
-Wextra	Call to function without interrupt attribute could clobber VFP registers of the interruptee.
-Wunused-lambda-capture	Variable explicitly captured by a lambda is not used in the body of the lambda.
-Wshadow-uncaptured-local	Declaration shadows a local variable.
-Wdynamic-exception-spec	ISO C++1z does not allow dynamic exception specifications.
-Wc++1z-compat	Mangled name of A will change in C++17 due to non-throwing exception specification in function signature.
-Wreturn-type	Control might reach end of non-void coroutine.
-Wmain	Boolean literal returned from *main*.
-Wincompatible-exception-spec	Exception specifications of return type differ.
-Winconsistent-missing-destructor-override	A overrides a destructor but is not marked as *override*.
-Walloca-with-align-alignof	Second argument to __builtin_alloca_with_align is supposed to be in bits.
-Wsigned-enum-bitfield	Enums in the Microsoft ABI are signed integers by default. Consider giving the enum A an unsigned underlying type to make this code portable.
-Wunguarded-availability	A is only available on B, C, or newer.
-Winvalid-partial-specialization	Class/variable template partial specialization is not more specialized than the primary template.

Table 4-3 LLVM 4.0 release: new warnings (cont.)

Category	Description
`-Wstrict-prototypes`	Function declaration is not a prototype.
`-Wblock-capture-autoreleasing`	Block captures an auto-releasing out-parameter, which might result in use-after-free bugs.
`-Waddress-of-packed-member`	Taking address of packed member A of class or structure B might result in an unaligned pointer value.
`-Wambiguous-delete`	Multiple suitable A functions for B; no *operator delete* function will be invoked if initialization throws an exception.
`-Wincompatible-function-pointer-types`	Incompatible function pointer types used.
`-Wformat`	Using A format specifier annotation outside of `os_log()`/`os_trace()`.
`-Wspir-compat`	Sampler initializer has invalid A bits.
`-Wnullability-completeness-on-arrays`	Array parameter is missing a NULL type specifier (`_Nonnull`, `_Nullable`, or `_Null_unspecified`).
`-Wgcc-compat`	`__final` is a GNU extension. Consider using C++11 final.

4.3.5.2 LLVM 6.0 release

The following warning flags are new for LLVM 6.0:

```
-Wbinary-literal
-Wbinding-in-condition
-Wbitfield-enum-conversion
-Wc++17-compat
-Wc++17-compat-mangling
-Wc++17-compat-pedantic
-Wc++17-extensions
-Wc++1z-compat-mangling
-Wc++2a-compat
-Wc++2a-compat-pedantic
-Wc++2a-extensions
-Wc++98-c++11-compat-binary-literal
-Wcoroutine-missing-unhandled-exception
-Wcpp
-Wenum-compare-switch
-Wmissing-noescape
-Wmsvc-not-found
-Wnsconsumed-mismatch
-Wnsreturns-mismatch
-Wnull-pointer-arithmetic
-Wobjc-flexible-array
-Wobjc-messaging-id
-Wout-of-scope-function
-Wpragma-clang-attribute
-Wpragma-pack
-Wpragma-pack-suspicious-include
-Wprofile-instr-missing
-Wredundant-parens
-Wtautological-constant-compare
-Wtautological-unsigned-enum-zero-compare
-Wtautological-unsigned-zero-compare
-Wundefined-internal-type
-Wunguarded-availability-new
-Wunicode-homoglyph
-Wunsupported-availability-guard
-Wunused-template
-Wzero-as-null-pointer-constant
```

For details, visit the llvm.org page:

releases.llvm.org/6.0.0/tools/clang/docs/DiagnosticsReference.html

4.3.5.3 LLVM 8.0 release

The following warning flags are new for LLVM 8.0:

```
-Wargument-outside-range
-Watimport-in-framework-header
-Watomic-alignment
-Watomic-implicit-seq-cst
-Wc++98-compat-extra-semi
-Wcast-qual-unrelated
-Wdangling
-Wdefaulted-function-deleted
-Wdeprecated-this-capture
-Wfixed-enum-extension
-Wframework-include-private-from-public
-Wfunction-multiversion
-Wignored-pragma-optimize
-Wimplicit-float-conversion
-Wimplicit-int-conversion
-Wincomplete-framework-module-declaration
-Wmemset-transposed-args
-Wnontrivial-memaccess
-Wobjc-property-assign-on-object-type
-Wordered-compare-function-pointers
-Wquoted-include-in-framework-header
-Wreturn-std-move
-Wreturn-std-move-in-c++11
-Wself-assign-overloaded
-Wstdlibcxx-not-found
-Wsuspicious-bzero
-Wsuspicious-memaccess
-Wunicode-zero-width
-Wunsupported-target-opt
```

For details, visit the llvm.org page:

https:\\clang.llvm.org/docs/DiagnosticsReference.html

4.3.5.4 LLVM 10.0 release

The following warning flags are new for LLVM 10.0:

```
-Walloca
-Wavr-rtlib-linking-quirks
-Wc++2a-designator
-Wc99-designator
-Wcall-to-pure-virtual-from-ctor-dtor
-Wconcepts-ts-compat
-Wctad-maybe-unsupported
-Wctu
-Wdangling-gsl
-Wdarwin-sdk-settings
-Wdelete-abstract-non-virtual-dtor
-Wdelete-non-abstract-non-virtual-dtor
-Wdeprecated-comma-subscript
-Wempty-init-stmt
-Wexport-unnamed
-Wexport-using-directive
-Wextra-semi-stmt
-Wfinal-dtor-non-final-class
-Wfortify-source
-Wimplicit-fixed-point-conversion
-Wimplicit-int-float-conversion
-Wincomplete-setjmp-declaration
-Winitializer-overrides
-Wint-in-bool-context
-Wmicrosoft-drectve-section
-Wmisexpect
-Wmissing-constinit
-Wmodule-import
-Wnoderef
-Wobjc-bool-constant-conversion
-Wobjc-boxing
-Wobjc-signed-char-bool
-Wobjc-signed-char-bool-implicit-float-conversion
-Wobjc-signed-char-bool-implicit-int-conversion
-Woverride-init
-Wpointer-compare
-Wpointer-integer-compare
-Wpoison-system-directories
-Wreorder-ctor
-Wreorder-init-list
-Wsigned-unsigned-wchar
-Wsizeof-array-div
-Wsizeof-pointer-div
-Wstack-exhausted
-Wtautological-objc-bool-compare
-Wunderaligned-exception-object
-Wxor-used-as-pow
```

For details, visit the llvm.org page:

https://clang.llvm.org/docs/DiagnosticsReference.html

4.3.5.5 LLVM 12.0 release

To review the warning flags that are new for LLVM 12.0, visit the following internal Wiki:

https://qwiki.qualcomm.com/quic-llvm-arm/SDLLVM_12_0_Warnings

4.3.6 Debugging

-dumpmachine

Displays the target machine name.

-dumpversion

Displays the compiler version.

-feliminate-unused-debug-symbols

Generates debug information only for the symbols that are used. (Debug information is generated in STABS format.)

-time
-ftime-report

Displays the elapsed time for each stage of the compilation.

-g[*level***]**

Generates complete source-level debug information.

-gline-tables-only

Generates source-level debug information with line number tables only.

-print-diagnostic-categories

Displays mapping of diagnostic category names to category identifiers.

-print-file-name=*library*

Displays the full library path of the specified file.

-print-libgcc-file-name

Displays the library path for the `libgcc.a` file.

-print-multi-directory

Displays the directory names of the multiple libraries specified by other compiler options in the current compilation.

-print-multi-lib

Displays the directory names of the multiple libraries paired with the compiler options that specified the libraries in the current compilation.

-print-multi-os-directory

Displays the relative path that is appended to the `multi-lib` search paths.

-print-prog-name=*program*

Displays the absolute path of the specified program.

-print-search-dirs

Displays the search paths used to locate libraries and programs during compilation.

-save-temps

Saves the normally-temporary intermediate files generated during compilation.

4.3.7 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 is driven by another program that requires 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 format.

This option controls whether LLVM prints the source line, source ranges, and caret when emitting a diagnostic. The default setting is `enabled`. When enabled, LLVM prints information like the following example:

```
test.c:28:8: warning: extra tokens at end of #endif directive
[-Wextra-tokens]
#endif bad
       ^
       //
-fdiagnostics-format=(clang|msvc|vi)
```

Changes 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 the output format is shown in the following examples.

- `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 `[-Woption]` information in diagnostic line.

This option controls whether LLVM prints the associated warning group option name (Section 4.3.31.3) 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 that is required to enable or disable the diagnostic, either from the command line or by using the `GCC diagnostic` pragma (Section 4.3.31.5).

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

Enables 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 the output format is shown in the following examples.

`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 can have an associated category. If it has one, it is listed in the diagnostic category field of the diagnostic line (in the `[]` brackets).

This option can be used to group diagnostics by category, so it should be a high-level category; the goal is to have 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, `-fdiagnostics-print-source-range-info` generates the braces, `{}`.

The printed column numbers count bytes from the beginning of the line; take care if the source contains multiple-byte characters.

-fdiagnostics-parseable-fixits

Prints 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 filename 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 the source contains multiple-byte 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*

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

4.3.8 Individual warning groups

-Wextra-tokens

Warns 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

Warns 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

Warns 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.

4.3.9 Compiler crash diagnostics

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

LLVM goes to great lengths to provide assistance in filing a bug report. Specifically, after a crash, it generates preprocessed source files and associated run scripts. Attach these files to a bug report to ease reproducibility of the failure. The following compiler option is used to control the crash diagnostics.

`-fno-crash-diagnostics`

Disables auto-generation of preprocessed source files during an LLVM crash.

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

4.3.10 Linker

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

Specifies 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 LLVM Arm toolchain:

-fuse-ld

Causes the toolchain to use the specified linker as the system linker. For example, `gold`, `bfd`, or `qcld`.

--gcc-toolchain

If `-fuse-ld` is not used, 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, causes the toolchain to use whatever linker is found in the specified sysroot.

If none of the these options are used, the toolchain uses the host linker by default. However, this results in errors during linking.

The `gold` and `bfd` options 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 5.6), while `bfd` does not.

The `qcld` option specifies the Snapdragon LLVM Arm linker. For more information, see the *Qualcomm Snapdragon LLVM Arm Linker User Guide* (80-VB419-102).

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).

NOTE: For more information on sysroots, see Section 4.4.

4.3.11 Preprocessor

-A pred=*ans*

Asserts the predicate *pred* and answer *ans*.

-A -pred=*ans*

Cancels the specified assertion.

-ansi

Uses C89 standard.

-C

Retains comments during preprocessing.

-CC

Retains comments during preprocessing, including during macro expansion.

-d(DMNU)

Prints macro definitions or names based on the following arguments:

D

Prints macro definitions in -E mode in addition to normal output.

M

Prints macro definitions in -E mode instead of normal output.

N

Prints macro names in -E mode in addition to normal output.

U

Prints referenced macro definitions in -E mode in addition to normal output. It also prints `#undefs` for macros that are undefined when referenced.

Both are printed at the point they are referenced.

-D *name*
-D *name*=*definition*

Defines the specified macro symbol.

-fexec-charset=charset

Specifies the character set used to encode strings and character constants. The default character set is `UTF-8`.

-finput-charset=charset

Specifies the character set used to encode the input files. The default setting is `UTF-8`.

-fpch-deps

Causes the dependency-output options to additionally list the files from a precompiled header's dependencies.

-fpreprocessed

Notifies the preprocessor that the input file has already been preprocessed.

-fstrict-overflow

Enforces strict language semantics for pointer arithmetic and signed overflow.

-ftabstop=*width*

Specifies the tab stop distance.

-fwide-exec-charset=charset

Specifies 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

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

-H

Displays the header includes and nesting depth.

--help

Displays the preprocessor release version.

-I *dir*

Adds the specified directory to the list of search directories for header files.

-I-

Deprecated.

-include *file*

Includes the contents of the specified source file.

-isystem *prefix*

Treats an included file as a system header if it is found on the specified path (Section 4.3.31.6).

-isystem-prefix *prefix*

Treats an included file as a system header if it is found on the specified sub-path of a defined include path (Section 4.3.31.6).

-ino-system-prefix *prefix*

Does not treat an included file as a system header if it is found on the specified sub-path of a defined include path (Section 4.3.31.6).

-M

Outputs a `make` rule describing the dependencies of the main source file.

-MD

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

-MF *file*

Writes dependencies to the specified file.

-MG

Adds missing headers to the dependency list.

-MM

Equivalent to `-M`, except this operation does not mention header files found in the system header directories.

-MMD

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

-MP

Creates artificial target for each dependency.

-MQ *target*

Specify a target to quote for dependency.

-MT *target*

Specify target for dependency.

-nostdinc

Omits searching for header files in the standard system directories.

-nostdinc++

Omits searching for header files in the C++-specific standard directories.

-o *file*

Specifies the name of the preprocessor output file.

-P

Disables line marker output when using `-E`.

-remap

Generates code for file systems that only support short filenames.

--target-help

Displays all command options and exit immediately.

-traditional-cpp

Emulates pre-standard C preprocessors.

-trigraphs

Preprocesses trigraphs.

-U *name*

Cancels any previous definition of the specified macro symbol.

-v

Equivalent to `-help`.

-version

Displays the preprocessor version during preprocessing.

--version

Displays the preprocessor version and exit immediately.

-w

Suppresses all preprocessor warnings.

-Wall

Enables all warnings.

-Wcomment
-Wcomments

Generates a warning if a comment symbol appears inside a comment.

-Wendif-labels

Generates a warning if an `#else` or `#endif` directive is followed by text.

-Werror

Converts all warnings into errors.

-Wimport

Generates a warning when `#import` is used the first time.

-Wsystem-headers

Generates a warning for constructs declared in system header files.

-Wtrigraphs

Generates a warning if a trigraph forms an escaped newline in a comment.

-Wundef

Generates a warning if an undefined non-macro identifier appears in an `#if` directive.

-Wunused-macros

Generates a warning if a macro is defined without being used.

4.3.12 Assembling

-fintegrated-as
-fno-integrated-as

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

`-fno-integrated-as` explicitly disables the use of the integrated assembler.

By default, the integrated assembler is enabled.

If a program can potentially generate hardware divide instructions (`SDIV`, `UDIV`), we strongly recommend using the integrated assembler. Older GNU assemblers might not understand these instructions.

When directly assembling a`*.s` source file, the 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*

Passes the specified argument to the assembler.

4.3.13 Linking

Starting with the LLVM 3.7 release, use clang as the driver for linking.

object_file_name

Linker input file.

-c

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

-dynamic

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

-E

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

-l *library*

Searches the specified library file while linking.

-moslib=*library*

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

-nodefaultlibs

Does not use the standard system libraries when linking.

-nostartfiles

Does not use the standard system startup files when linking.

-nostdlib

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

-pie

Generates a position-independent executable as the output file.

-s

Deletes all symbol table information and relocation information from the executable.

-S

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

-shared

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

-shared-libgcc

Links with the shared version of the `libgcc` library.

-static

Does not link with the shared libraries. Only relevant when using dynamic libraries.

-static-libgcc

Links with the static version of the `libgcc` library.

-symbolic

Binds references to global symbols when building a shared object.

-u *symbol*

Pretends the *symbol* is undefined to force linking of library modules to define *symbol*.

-Xlinker *arg*

Passes the specified argument to the linker.

4.3.14 Directory search

-B*prefix*

Specifies the top-level directory of the compiler.

-F *dir*

Adds the specified directory to the search path for framework includes.

--gcc-toolchain=*prefix*

Equivalent to `-B`*prefix*.

-I *dir*

Adds the specified directory to the include file search path.

-I-

Deprecated.

-L*dir*

Adds the specified directory to the list of directories searched by the `-l` option.

--sysroot=*prefix*

Specifies the root directory of the system tools environment (Section 4.4).

4.3.15 Processor version

LLVM defines the `-target`, `-march`, and `-mcpu` options that are used to specify the Arm processor version for which code is generated. 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 Armv7tdmi CPU.

If the Armv7 or Armv8 architecture is specified using the `-march` or `-mcpu` options, but Arm mode (that is, 32-bit-only mode) is not specified on the command line, the LLVM compilers default to generating code in Thumb-2 mode. To disable Thumb mode, use the `-mno-thumb` or `-marm` options.

-target *triple*

Specifies 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 Armv7-A architecture, 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 4.4).

Thus, the example command can be alternately expressed as follows:

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

Where `cortex-a9` indicates the Armv7-A processor as the CPU.

Following 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`*

Specifies the Arm architecture for code generation.

This option has the following possible values:

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

For more information on architecture versions, see Section 4.3.15.

`-mcpu=`*`version`*

Specifies 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
```

Following are some commonly used CPU values:

Armv7 CPU:	Qualcomm Armv7 CPU:	Armv8 CPU:
`cortex-a8`	`scorpion`	`cortex-a53`
`cortex-a9`	`krait`	`cortex-a57`
`cortex-a15`		`kryo`

NOTE: `-mcpu` automatically sets `-mfpu`.

-mfpu=*version*

Specifies 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
```

Commonly used option values for -mfpu

neon

Enable the NEON single instruction, multiple data (SIMD) architecture extension for the Arm Cortex-A or Qualcomm Armv7 and Armv8 processors.

vfpv4

Enable the VFPv4 architecture extensions. The VFPv4 extension enables code generation of the fused multiply add and subtract instructions (Section 4.3.19).

neon-fp-armv8

Enable NEON and Armv8 FP extensions.

crypto-neon-fp-armv8

Enable Cryptography, NEON, and Armv8 FP extensions.

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
```

Automatic use of -mfpu

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`. However, using `-mcpu`, `-march`, or `--target` with `-mfpu` generates an error if the specified `-mfpu` option is invalid.

-mfloat-abi=(soft|softfp|hard)

Specifies the floating-point ABI.

The Armv8 processor mandates a hardware floating point.

4.3.16 Armv8.x extensions and features

New features have been added in each of the Armv8.x-A extensions. Some features are limited to the AArch64 state, and other features are available in both the AArch32 and AArch64 states.

When using clang, you can enable all the features available in a certain architecture extension version by setting the -march flag on the clang command line. For example:

`-march=armv8.1-a`	Enable Armv8.1a instructions
`-march=armv8.2-a`	Enable Armv8.2a instructions
`-march=armv8.3-a`	Enable Armv8.3a instructions
`-march=armv8.4-a`	Enable Armv8.4a instructions
`-march=armv8.5-a`	Enable Armv8.5a instructions

Alternatively, you can enable or disable an individual feature by setting the flag that corresponds to the feature, preceded by + or -. For example:

`-march=armv8+lse`	Enable atomic memory access instructions from `armv8.1-a`
`-march=armv8.1-a-lse`	Disable atomic memory access instructions from `armv8.1-a`

The most important features for AArch32 and AArch64 are listed in the following sections. For more details, go to the Arm developer website:

https://developer.arm.com/architectures/learn-the-architecture/understanding-the-armv8.x-extensions/armv8.x-extensions-and-features

4.3.16.1 AArch64 state

Table 4-4 lists the features enabled by default with each architecture extension version. The features are cumulative, for example, `armv8.2-a` enables all `armv8.1-a` features plus some new features.

Table 4-4 AArch64 features enabled by default

Extension version	Enabled features
-march=armv8.1-a	crc, lor, lse, pan, rdm, vh
-march=armv8.2-a	All armv8.1-a features plus ccpp, pan-rvw, ras, uaops
-march=armv8.3-a	All armv8.2-a features plus ccidx, complxnum, cpc, jsconv, pa
-march=armv8.4-a	All armv8.3-a features plus am, dit, dotprod, fmi, mpam, nv, rasv8_4, rcpc-immo, sel2, tlb-rmi, tracev8.4
-march=armv8.5-a	All armv8.4-a features plus altnzcv, bti, ccdp, fptoint, predres, sb, specrestrict, ssbs

Table 4-5 lists all the AArch64 features per extension.

Table 4-5 AArch64 extension-enabled features

aes	Enable AES support
altnzcv	Enable Armv8.5 alternative NZCV format for floating-point comparisons
am	Enable Armv8.4-A Activity Monitors extension
bti	Enable Armv8.5 branch target identification
ccdp	Enable Armv8.5 cache clean to point of deep persistence
ccidx	Enable Armv8.3-A extension of the CCSIDR number of sets
ccpp	Enable Armv8.2 data cache clean to point of persistence
complxnum	Enable Armv8.3-A floating-point complex number support
crc	Enable Armv8 CRC-32 checksum instructions
crypto	Enable cryptographic instructions
dit	Enable Armv8.4-A data independent timing instructions
dotprod	Enable dot product support
fmi	Enable Armv8.4-A flag manipulation instructions
fp-armv8	Enable Armv8 FP
fp16fml	Enable FP16 FML instructions
fptoint	Enable Armv8.5 FRInt[32|64][Z|X] instructions that round a floating-point number to an integer (in FP format), forcing it to fit into a 32- or 64-bit integer
fullfp16	Enable Full FP16 instructions
jsconv	Enable Armv8.3-A JavaScript FP conversion enhancement
lor	Enable Armv8.1 Limited Ordering Regions extension
lse	Enable Armv8.1 Large System Extension (LSE) atomic instructions

Table 4-5 AArch64 extension-enabled features (cont.)

mpam	Enable Armv8.4-A Memory System Partitioning and Monitoring extension
neon	Enable advanced SIMD instructions
nv	Enable Armv8.4-A nested virtualization enhancement
pa	Enable Armv8.3-A pointer authentication enhancement
pan	Enable Armv8.1 Privileged Access-Never extension
pan-rwv	Enable Armv8.2 PAN s1e1R and s1e1W variants
perfmon	Enable Armv8 PMUv3 Performance Monitors extension
predres	Enable Armv8.5-A execution and data prediction invalidation instructions
rand	Enable random number generation instructions
ras	Enable Armv8 Reliability, Availability, and Serviceability extensions
rasv8_4	Enable Armv8.4-A Reliability, Availability, and Serviceability extension
rcpc	Enable support for RCPC extension
rcpc-immo	Enable Armv8.4-A RCPC instructions with immediate offsets
rdm	Enable Armv8.1 rounding double multiply add/subtract instructions
sb	Enable Armv8.5 speculation barrier
sel2	Enable Armv8.4-A Secure Exception Level 2 extension
sha2	Enable SHA1 and SHA256 support
sha3	Enable SHA512 and SHA3 support
sm4	Enable SM3 and SM4 support
spe	Enable Statistical Profiling extension
specrestrict	Enable Armv8.5 architectural speculation restriction
ssbs	Enable Armv8.5 speculative store bypass safe bit
sve	Enable Scalable Vector Extension (SVE) instructions
tlb-rmi	Enable Armv8.4-A TLB range and maintenance instructions
tme	Enable Transactional Memory extension
tracev8.4	Enable Armv8.4-A Trace extension
uaops	Enable Armv8.2 UAO PState
vh	Enable Armv8.1 Virtual Host extension

4.3.16.2 AArch32 state

Table 4-6 lists the features enabled by default with each architecture extension version. The features are cumulative, for example, `armv8.2-a` enables all `armv8.1-a` features plus some new features.

Table 4-6 AArch32 features enabled by default

Extension version	Enabled features
-march=armv8.1-a	db, fp-armv8, neon, dsp, trustzone, virtualization, crc, crypto
-march=armv8.2-a	All armv8.1-a features plus ras
-march=armv8.3-a	All armv8.2-a features
-march=armv8.4-a	All armv8.3-a features plus dotprod
-march=armv8.5-a	All armv8.4-a features plus sb
-march=armv8-r	Supports real-time profile (R series) db, fp-armv8, neon, dsp, crc, dfb, mp, virtualization
-march=armv8-m.base	Supports Armv8-M baseline instructions db, 8msecext, noarm, acquire-release, hwdiv, thumb-mode, mclass, strict-align, v7clrex
-march=armv8-m.main	Supports Armv8-M mainline instructions db, 8msecext, noarm, acquire-release, hwdiv, thumb-mode
-march=armv8.1-m.main	Supports Armv8.1-M mainline instructions All armv8-m.main features plus ras, lob

Table 4-7 lists all the AArch32 features per extension.

Table 4-7 AArch32 extension-enabled features

8msecext	Enable support for Armv8-M Security extensions
acquire-release	Enable support for Armv8 acquire and release instructions
aes	Enable AES support
crc	Enable support for CRC instructions
crypto	Enable Cryptography extensions
db	Enable data barrier (dmb/dsb) instructions
dfb	Enable full data barrier instruction
dotprod	Enable support for dot product instructions
dsp	Enable DSP instructions in Arm or Thumb-2
fp-armv8	Enable Armv8 FP
fp16fml	Enable full half-precision floating-point FML instructions
fpregs16	Enable 16-bit FP registers
fpregs64	Enable 64-bit FP registers
fullfp16	Enable full half-precision floating point
hwdiv	Enable Divide Instructions in Thumb

Table 4-7 AArch32 extension-enabled features (cont.)

lob	Enable Low Overhead Branch extensions
mp	Enable Multiprocessing extension
mve	Enable M-Class Vector extension with integer operations Implies support for Armv8.1-M mainline instructions plus dsp, fpregs16, fpregs64
mve.fp	Enable M-Class Vector extension with integer and floating operations Implies mve plus fp-armv8, fpregs16, fullfp16
neon	Enable NEON instructions
noarm	Does not support Arm mode execution
ras	Enable Reliability, Availability, and Serviceability extensions
sb	Enable Armv8.5-a speculation barrier
sha2	Enable SHA1 and SHA256 support
strict-align	Disallow all unaligned memory access
thumb-mode	Support Thumb mode execution
trustzone	Enable support for Arm TrustZone security extensions
v7clrex	Has Armv7-A clrex instruction
virtualization	Enable Virtualization extension

4.3.17 Armv8 security and vectorization features

The next generation of Armv8 extensions include new security features. This section provides an overview of these features, how they work, and how to enable them.

NOTE: The support for these features is currently under development in the toolchain. For the most up-to-date information on supported features and how to enable them contact sdllvm.support.

The security techniques focus on protecting code from modern Return Oriented Programming (ROP) and Jump Oriented Programming (JOP) code reuse forms of attacks. They can also protect from bugs and security vulnerabilities when dealing with memory accesses such as out-of-bound access, dangling pointers, use-after-expiration, and so on.

In ROP and JOP attacks, hackers reuse existing pieces of code, or *gadgets*, that collectively do what attackers want by hijacking control flow to execute gadgets in the right sequence. There are several ways to mitigate ROP and JOP attacks. The following sections discuss two of those techniques, Pointer Authentication (PAuth) and Branch Target Identification (BTI).

The security features are complex and require hardware support as well as support from the entire software stack, including the compiler, ABI, OS/kernel/bootloader, and libraries (such as libc). Currently, the community is actively working in these areas.

4.3.17.1 Pointer Authentication Extension (PAuth)

PAuth can protect the integrity of code and data by detecting malicious change of pointer values stored in memory. It can prevent use of contaminated addresses as return addresses, C function pointers, C++ virtual functions, C++ member function pointers, and so on. The protection mechanism adds a signature to every code pointer and certain data pointers, and then it authenticates the signatures before using the pointers. In summary:

- A pointer is signed before being stored in memory.
- A pointer is authenticated before use.
- A signature or Pointer Authentication Code (PAC) is computed at signing and compared at authenticating.
- When PAC authentication fails upon use of the pointer, the CPU traps, allowing the operating system to choose the course of action.

The hardware support for the PAuth extension is available in Armv8.3-A for AArch64 targets. It includes operations for signing and authenticating pointers, stripping signatures out of pointers, and PAC computation, which is performed by a hardware algorithm such as QARMA (https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf).

Following is an example of using PAuth for software stack protection (PACRET).

```
PACIASP       ; sign x30
SUB  sp, sp, #0x40
STP  x29, x30, [sp, #0x30]
ADD  x29, sp, #0x30
...
...



LDP  x29, x30, [sp, #0x30]
ADD  sp, sp, #0x40
AUTIASP       ; authenticate x30
RET
```

The protection code is automatically generated by the compiler in the function prologue and epilogue as shown in the assembly language, in which the return address is saved on the stack after being signed with PACIASP. Then it is authenticated with AUTIASP before use when returning from the function.

LLVM toolchain support for PAuth is currently under development. However, PACRET is currently supported. To enable it, set the following flag:

`-mbranch-protection=`*`<protection>`* **`-march=armv8.3-a`** (or a later version)

The *`<protection>`* field can be set to `pac-ret{+leaf+b-key}` where:

- `pac-ret` enables return address signing for non-leaf functions using the A-key.
- `+leaf` increases the scope of return address signing to include leaf functions.
- `+b-key` uses B-key instructions to sign addresses instead of A-key instructions.

For more details on PAuth, see the Arm documentation. For the most up-to-date information on supported features and how to enable them contact sdllvm.support.

4.3.17.2 Branch Target Identification (BTI)

BTI is another security feature used to mitigate the JOP code reuse form of attacks. These attacks take advantage of the fact that indirect branches in the code can legally land anywhere in the program.

BTI ensures that indirect branches must land on corresponding instructions; it pairs every indirect instruction with a corresponding legal instruction. Branching to an incompatible instruction raises a branch target exception.

The different BTI identifications for a target consist of the following:

- c – Target of indirect calls (BLR rn)
- j – Target of indirect jumps (BR rn)
- jc – Target of indirect jumps or calls

Following is an example of code protected by BIT, in which an attacker finds an exploitable gadget in the code. The attacker overwrites a stack location with a new function address to point to, and then it tries to execute an indirect branch to it. However, the call site expects that only an indirect call instruction can land on that location, and thus an exception will be raised.

```
_getBitsInByte:
    bti c           ;only an indirect call (blr) instruction can
                     land on this target
    mov x0, 0x8     ;return number of bits in a byte
    ldr x30, sp
    ret

_readPasswordHeader:
    mov x17, 0x1f0174ed0  ;put address of password file in scratch
                           register
    ldr  x0, x17          ;load from it (address left in register)
    ldr  x1, sp, #32      ;attacker overwrites [sp, #32] with start
                           address of _getBitsInByte
    br   x1               ;indirect jump using x1
    ...
```

This security extension is available in Armv8.5a for AArch64 targets.

All the BTI instructions are in the NOP space, which means binaries protected with BTI are backward-- compatible.

The LLVM toolchain supports the BTI feature. To enable, set the following flag:

`-mbranch-protection=bti -march=armv8.5-a` (or a later version)

For more details on BTI, see the Arm documentation.

For the most up-to-date information on supported features and how to enable them contact sdllvm.support.

4.3.17.3 Scalable Vector Extensions (SVE)

SVE is a SIMD vector extension introduced in Armv8.3 for Armv8-A AArch64 target.

The extension is vector-length agnostic. It allows hardware implementations to choose a vector register length between 128 to 2048 bits, in increments of 128 bits. With this extension, there is no need to define a new ISA for new vector lengths.

In addition, this extension allows for a new programming model in which software scales dynamically to the available vector length, without requiring recompilation or rewriting hand-coded assembly or compiler intrinsics for new vector lengths. This feature reduces software deployment and deployment effort and cost.

Besides the support for scalable and longer vector types, SVE supports predication, which is central to its design. A set of scalable predicate registers govern predicates for load, store and arithmetic operations, additional predicates for loop management, and a first fault register (FFR) for software speculation.

When combined, all the these features help tackle traditional barriers for compiler auto-vectorization. For example, the extensions support software-managed speculative vectorization, allowing uncounted loops to be vectorized. Moreover, in-vector serialized inner loop permits outer loop vectorization in spite of dependencies. These features help increase vector utilization, which in turn leads to better program performance. Following are more details on the new features:

- Gather-load and scatter-store

  These instructions provide enhanced addressing modes for non-linear data accesses. For example, instructions to load a single vector register from non-contiguous memory locations.

- Per-lane predication

  These instructions operate on individual lanes of a vector controlled by of a governing predicate register. This feature allows vectorization of loops containing complex control flow.

- Predicate-driven loop control and management

  These instructions eliminate loop heads and tails and other overhead by processing partial vectors, such as when handling leftover iterations.

- Vector partitioning and software-managed speculation

  Predicates help create sub-vectors (that is, partitions) in response to data and dynamic faults. The extension supports first-fault load instructions that allow vector access to safely cross a page boundary, which, in turn, enables vectorizing uncounted loops with break conditions or data dependent exits.

- Extended floating-point and bitwise horizontal reductions

  These instructions are applicable to more types of reducible loop-carried dependencies. For example, instructions that allow in-order or tree-based floating-point sum.

SVE is not an extension of NEON. It is a separate, optional extension with a new set of instruction encodings that target increased parallelism in High Performance Computing (HPC) scientific code. You can enable SVE in conjunction with the NEON extension.

The SDLLVM toolchain includes support to assemble source code containing SVE instructions, disassemble ELF object files containing SVE instructions, and use intrinsics to write SVE instructions directly from C code.

SDLLVM also includes the Polly auto-vectorizer that takes advantage of SVE scatter load and store instructions. Support for SVE predication instructions within Polly is currently under development. The following sections provide examples of these features.

Gather-load example

In this C code example and its corresponding assembly code, even and odd indexed values are accessed separately, gathering non-contiguous data from memory into vector registers with instruction `ld2w`. At the end, Z0 contains the values A[0], A[2], A[4], A[6]; and Z1 contains the values A[1], A[3], A[5], A[7]

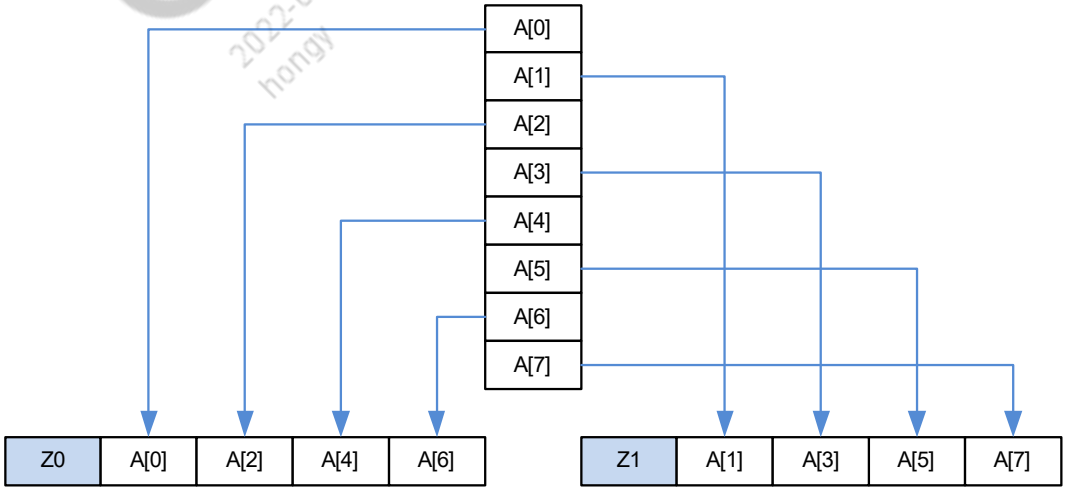
C code:

```
void gather_load(int *A, int *B, int n) {
  for (int I = 0; I < n; ++i) {
    B[i] = A[0] + A[1];
    A+=2;
  }
}
```

Assembly code:

```
.LBB1_9:                                // %polly.loop_header
                                        // =>This Inner Loop Header: Depth=1
ld2w    { z0.s, z1.s }, p0/z, [x15]
add     x10, x10, x11
cmp     x10, x9
add     x15, x15, x14
add     z0.s, z1.s, z0.s
st1w    { z0.s }, p0, [x16]
add     x16, x16, x13
b.ne    .LBB1_9
```

Memory and register layout:



Scatter-store example

In this C code example and its corresponding assembly code, even and odd indexed values are stored into non-contiguous memory. Values 1 and 2 are stored into even indexed and odd indexed locations of A, respectively, with instruction `st2w`. Z0 contains all 1s, and Z1 contains all 2s.

C code:

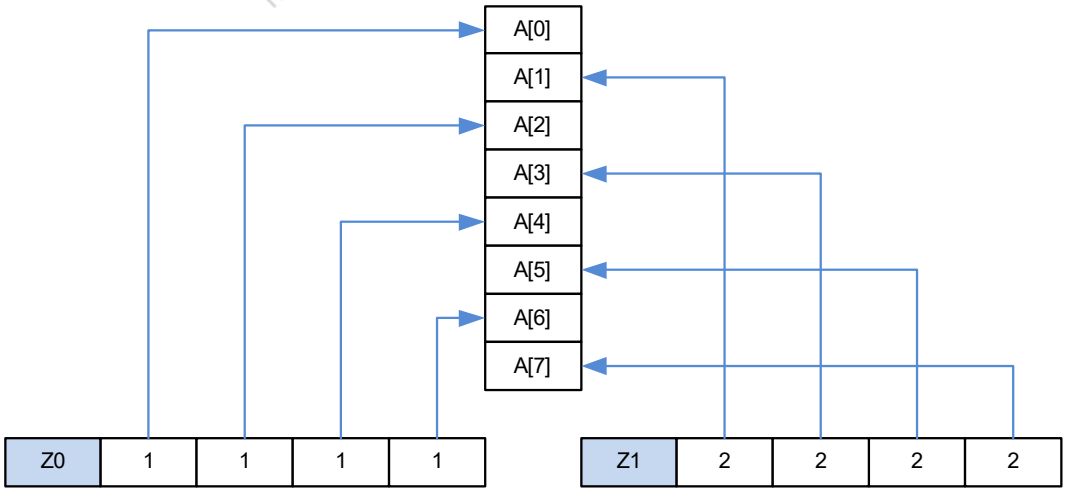
```
void scatter_store(int *A, int *B, int *C, int n) {
  for (int i = 0; i < n; ++i) {
    A[0] = 1;
    A[1] = 2;
    A+=2;
  }
}
```

Assembly code:

```
mov     z0.s, #1                        // =0x1
..
mov     z1.s, #2                        // =0x2
..
.LBB2_4:                                // %polly.loop_header
                                        // =>This Inner Loop Header: Depth=1

    add     x11, x11, x12
    st2w    { z0.s, z1.s }, p0, [x14]
    cmp     x11, x10
    add     x14, x14, x13
    b.ne    .LBB2_4
```

Memory and register layout:



Per-lane predication example

This C code example and its corresponding assembly code use per-lane predication through ACLE SVE intrinsics. SVE register Z0 contains argument vector A, and Z1 corresponds to vector B. Predicate register P0 corresponds to cc. Only the lanes mentioned as active by cc are used for the addition.

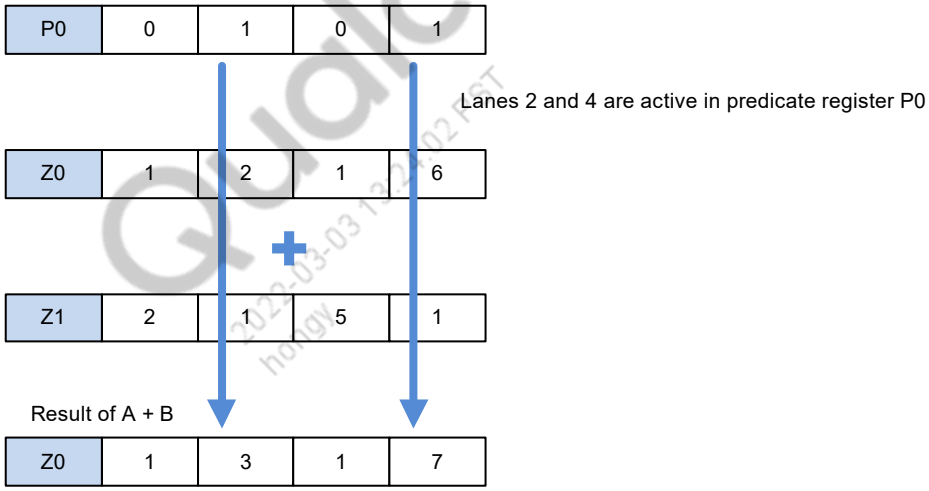
C code:

```
#include <arm_sve.h>
svint32_t per_lane_add(svbool_t cc, svint32_t A, svint32_t B) {
  return svadd_s32_m(cc, A, B);
}
```

Assembly code:

```
per_lane_add:
    add    z0.s, p0/m, z0.s, z1.s
    ret
```

Register layout:



For more details

See the ARM documentation or contact sdllvm.support.

4.3.18 Security threat mitigation

As security vulnerabilities are discovered in software or hardware, the LLVM community is committed to addressing them by fixing security holes in compiler-generated code and related tools, as well as providing hardware workarounds to mitigate threats. Following is an overview of the current mitigation features available in the compiler.

4.3.18.1 Spectre-Meltdown mitigation

Speculative Load Hardening is a best-effort mitigation against information leak attacks that use control flow miss-speculation, specifically miss-speculation of whether a branch is taken or not.

For a detailed discussion on Spectre-Meltdown mitigatio, go to the discussion at:

https://llvm.org/docs/SpeculativeLoadHardening.html

Enable the mitigation through a compiler flag:

```
-mspeculative-load-hardening  and -mno-speculative-load-hardening
```

Or by using a function attribute:

```
__attribute__((speculative_load_hardening))
__attribute__((no__speculative_load_hardening))
```

Check the Clang documentation for the precedence rules when combining this flag and function attribute, and for the implications of inlining optimization. Go to:

https://clang.llvm.org/docs/AttributeReference.html#no-speculative-load-hardening

Finally, be aware this feature impacts performance.

4.3.19 Code generation

-fasynchronous-unwind-tables

Generates an unwind table. The table is stored in DWARF2 format.

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

Prevents 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 setting is `disabled`.

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

In the following example, enabling `-fchar-array-precise-tbaa` results in the statement `d = *p` being hoisted out of loops, 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

Emits all data, even if unused.

-femit-all-decls

Emits 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 you are explicitly specifying the use of a fused operation, the substitution is considered IEEE compliant.

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

`fast`

Enables fused operations throughout the program.

`on`

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

`off`

Disables 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.

`-finstrument-functions`

Generates instrumentation calls in function entries and exits.

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

Attempts to merge functions that are equivalent, or differ by only a few instructions (Section 5.5). 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 might have a negative impact on program performance, it is disabled by default. It becomes enabled only when it is specified explicitly.

`-fno-exceptions`

Does not generate code for propagating exceptions.

`-fpic`

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

`-fPIC`

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

`-fpie`
`-fPIE`

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

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

Generates instrumentation for the address sanitizer (Section 8.2).

-fsanitize=memory
-fno-sanitize=memory

Generates instrumentation for the memory sanitizer (Section 8.5).

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

Generates instrumentation for the undefined behavior sanitizer. You can specify one or more events.

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 enumeration values that are out-of-range for an enumeration type.

float-cast-overflow

Floating-point conversion that 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

Divide an integer by zero.

nonnull-attribute

Return a NULL pointer from a function declared to never return NULL.

null

Use 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 might 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

Return a 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 8.7).

-fsanitize=integer

Generates instrumentation for the undefined behavior sanitizer for the following events:

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

-fsanitize=undefined

Generates instrumentation for the undefined behavior sanitizer for the following events:

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

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

Disables the generation of `-fsanitize` runtime checks in the specified functions or source code files (Section 8.1.1).

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

- Function names are prefixed with `fun:`
- Filenames 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.

You can specify file and function names using regular expressions, but `#` works as it does in shell wildcards.

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

Tracks the origin of uninitialized memory in the memory sanitizer (Section 8.5).

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`

Controls 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`

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

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

Controls 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`, when it is disabled by default.

-fsanitize-use-embedded-rt

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

-fshort-enums
-fno-short-enums

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

-fshort-wchar
-fno-short-wchar

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

-ftrap-function=*name*

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

-ftrapv

Traps on integer overflow.

-ftrapv-handler=*name*

Specifies 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

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

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

Sets the default symbol visibility for all global declarations.

-fwrapv

Treats signed integer overflow as two's complement.

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

Controls 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)

This option applies only to Armv7 processors that support hardware divide.

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

-mllvm -aarch64-disable-abs-reloc

Eliminates 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 setting is `disabled`.

-mllvm -arm-expand-memcpy-runtime

Sets 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

Controls 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 a data size above the specified threshold will not be compiled into vector store operations.

The default setting is 128.

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

Controls 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 a 0 value that can be implemented with NEON vector stores. A memset call that writes a 0 value with a data size above the specified threshold will not be compiled into vector store operations.

The default setting is 32.

-mllvm -arm-opt-memcpy

The optimized libc for Qualcomm® Krait™ CPU 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 with `-mllvm -arm-expand-memcpy-runtime`, the compiler transforms the LLVM IR by replacing memcpy calls with the runtime checks for a 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.

Also, set the option for the 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 setting is `disabled`.

-mllvm -disable-thumb-scale-addressing

Controls 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 setting 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

Controls the generation of hardware divide instructions (Section 4.4).

-mllvm -enable-arm-addressing-opt

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

The default setting is `enabled`.

-mllvm -enable-arm-peephole

Enables 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 setting 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 setting is `enabled`.

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

When this option is used 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 option ensures assembly code compatibility between LLVM and GNU tools while the tools are out of sync (for example, the 4.9 GNU assembler currently uses `#0` syntax).

-mllvm -enable-round-robin-RA

Enables a round-robin register allocation heuristic that 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 setting is `disabled`, unless `-mcpu=krait` is specified.

-mllvm -enable-select-to-intrinsics

Exposes more if statements to be converted to LLVM IR's SELECT instruction, which in turn can more easily be mapped to Arm hardware instructions.

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

-mllvm -favor-r0-7

Enables a heuristic in the Greedy Register Allocator that better guides the assignment of high-order registers (R8-R15) that are currently avoided aggressively in the allocator. The allocator exploits the fact that a Thumb-2 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 setting is `disabled`.

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

-mllvm -force-div-attr

Controls the generation of hardware divide instructions (Section 4.4).

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

Configures 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 setting is `L1`.

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

-mrestrict-it
-mno-restrict-it

Controls 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 the Armv8 architecture.

-mno-restrict-it

Allows generation of legacy IT blocks (that is, deprecated forms in the Armv7 architecture).

The default option setting is determined by the target Armv8 or Armv7 architecture. For Armv8 (AArch32) Thumb mode, `-mrestrict-it` is enabled by default, while for other targets it is disabled by default.

4.3.20 Vectorization

-ftree-vectorize

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

-fvectorize-loops

Performs automatic vectorization of loop code (Section 5.3).

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 CPU that has this extension with a `-mfpu` flag), or explicitly with `-mfpu=neon`.

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

-fvectorize-loops-debug

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

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

NOTE: LLVM does not support the GCC `-ftree-vectorizer-verbose` option.

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

Controls the automatic insertion of Arm `PLD` instructions into loops that are vectorized.

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

The default setting is `disabled`.

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

4.3.21 Parallelization

`-fparallel`

Performs automatic parallelization of loop code (Section 5.4).

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-num-workloads=`*n*

Specifies the maximum number of workloads that a given loop can be subdivided into for parallel execution on a pool of threads. The valid range is 1 to 64. The default value is 4.

The maximum number of workloads includes the base or main thread of the application, and is determined by the number of online cores available at program initialization.

When this option is used with `-fparallel-symphony`, the valid range changes to any nonzero positive value, and the default value changes to 128.

NOTE: This option was previously named **`-fparallel-num-threads`**.

`-fparallel-symphony`

Performs automatic parallelization of loop code at runtime using the SYMPHONY library (Section 5.4.1).

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.

4.3.22 Optimization

-O0

Does not optimize. This is the default optimization setting.

-O
-O1

Enables a small set of optimizations. We do not recommend this optimization level for performance or code size.

-O2

Enables optimizations for performance, including automatic loop vectorization (Section 4.3.20). Optimizations enabled at `-O2` improve performance, but they might cause a small-to-moderate increase in compiled code size.

-O3

Enables aggressive optimizations for performance. Optimizations enabled at `-O3` improve performance, but they might cause a large increase in compiled code size.

-O4

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

The Qualcomm LLVM compilers define `-O4` differently from the standard LLVM compiler. In particular, the Qualcomm compilers do not enable link-time optimization (Section 5.6) in `-O4`, while the standard compiler does enable it in `-O4`, additionally mapping `-O4` to `-O3`.

-Ofast

Enables all options at `-O3`, and increases the aggressiveness of optimizations such as function inlining and loop unrolling. Also, `-Ofast` enables fast math that allows floating-point computation optimizations. User applications that require IEEE floating-point conforming code must not use `-Ofast` because fast math does not preserve IEEE floating-point requirements.

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

-Os

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

-Osize

Enables `-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 is tuned for Armv7 targets only. It is not 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.

4.3.22.1 Recommended options for best performance

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

-Ofast -mcpu=cortex-a57

Thumb mode and `-fvectorize-loops` are enabled by default.

-Ofast -mcpu=cortex-a57 -marm

Arm mode and `-fvectorize-loops` are enabled by default.

4.3.22.2 Recommended options for best code size

Currently, LLVM generates the smallest code when compiled for Thumb-2 mode. Following are the recommended options for generating compact code:

`-Osize -mthumb`

For 32-bit mode.

`-Os`

For 64-bit mode (AArch64).

4.3.23 Specific optimizations

`-faggressive-unroll`

Use aggressive unroll heuristics to decide on unrolling loops. This option results in more loops that are unrolled.

`-fdata-sections`

Assigns each data item to its own section.

`-ffp-contract=[off|on|fast]`

Forms fused floating-point operations such as fused multiply-accumulates.

`off`

Do not fuse operations. (Default)

`fast`

Fuse operations whenever possible.

`on`

Enable the `FP_CONTRACT` default pragma.

`-ffunction-sections`

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

`-finline`

Specifies the `inline` keyword as active.

`-finline-functions`

Performs heuristically-selected inlining of functions.

`-floop-pragma`

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

`-fno-zero-initialized-in-bss`

Assigns all variables that are initialized to zero to the BSS section.

-fnomerge-all-constants

Does not merge constants.

-fomit-frame-pointer

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

-foptimize-sibling-calls

Optimizes function sibling calls and tail-recursive calls.

-fstack-protector

Generates code that checks selected functions for buffer overflows.

-fstack-protector-all

Generates code that checks all functions for buffer overflows.

-fstack-protector-strong

Generates code that 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

Enforces the strictest possible aliasing rules for the language being compiled.

-funit-at-a-time

Parses the entire compilation unit before beginning code generation.

-funroll-all-loops

Unrolls all loops.

-funroll-loops

Unrolls selected loops.

`-maggressive-size-opts`

Enables additional optimizations that might be useful for bare metal images.

`--param ssp-buffer-size=`*`size`*

Specifies the minimum size (in bytes) that a buffer must be if buffer-overflow checks are to be generated for it by the `-fstack-protector` options. The default value is 8.

4.3.24 Math optimizations

-fassociative-math

Allows the operands in a sequence of floating-point operations to be re-associated.

Because this option might 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.

The optimization flag, `-frounding-math`, is not supported: `[-Wignored-optimization-argument]`.

NOTE: This option enables additional features of parallelization (Section 5.4).

-ffast-math

Enables the Fast-math mode in the compiler front end.

This option has no effect on optimizations, but it defines the preprocessor macro `__FAST_MATH__` , which is the same as the GCC `-ffast-math` option.

-ffinite-math-only

Enables optimizations that assume that floating-point argument and result values are never NaNs nor +-Infs.

-fno-math-errno

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

-freciprocal-math

Enables optimizations that assume that the reciprocal of a value can be used instead of dividing by the value.

-fno-signed-zeros

Enables optimizations that ignore the sign of floating-point zero values.

-fno-trapping-math

Enables optimizations that assume that floating-point operations cannot generate user-visible traps.

-funsafe-math-optimizations

Enables code optimizations that assume the floating-point arguments and results are valid, and that they might violate IEEE or ANSI standards.

This option enables the following:

```
-fno-signed-zeros
-fno-trapping-math
-fassociative-math
-freciprocal-math
```

4.3.25 Link-time optimization

-flto

Performs link-time optimization (Section 5.6).

Use this option 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 that is used during link-time optimization (LTO).

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, we strongly recommend using `clang/clang++` to perform the linking.

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. You can specify the `gold` linker with the `-fuse` option (Section 4.3.10).

NOTE: 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.

4.3.26 Profile-guided optimizations

-fprofile-instr-append-file

Save into the same profile file the profiling data from different runs of the same executable or different executable runs.

-fprofile-instr-file-sync

We recommend using this flag for collecting profile data on Android.

Set this option to cause the compiler to instrument functions with a call to synchronize profile data to a file. The actual synchronization operation occurs periodically upon entering a function and the number of times the function is entered exceeds a threshold value, which is set via the `-fprofile-instr-sync-interval` option.

-fprofile-instr-file-sync-thread

We recommend using this flag for collecting profile data on Android.

Set this option to cause the compiler to create a service thread for syncing profiling data periodically with the time interval (in milliseconds) set via the `-fprofile-instr-sync-interval` option.

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

Specifies the name and location of the raw profile data file to be created. The default name is `default.profraw`.

The raw profile data file is used in instrumentation-based profile-guided optimization (Section 5.8).

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

Periodically synchronizes the collected profile data to the profile data file with the specified time interval (in milliseconds). We recommend using this flag for collecting profile data on Android.

If `-fprofile-instr-file-sync-thread` is enabled, it periodically synchronizes the collected profile data to the profile data file with the specified time interval.

If `-fprofile-instr-file-sync` is enabled, this flag also means a threshold value for the number of times a function is entered upon which the profile data synchronization is triggered.

-fprofile-instr-use=*filename*

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

-fprofile-sample-use=*filename*

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

4.3.27 Optimization reports

-fopt-reporter=(vectorizer|parallelizer|all)

Requests the specified type of optimization report data (Section 5.9).

-polly-max-pointer-aliasing-checks

Increases 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

Outputs the line numbers of the loops that were auto-parallelized and/or vectorized.

-Rpass-missed=loop-opt

Outputs the line number and reason why a loop was not optimized.

4.3.28 Adaptive execution

-mae
-mno-ae

Controls whether the compiler generates code checks to verify that the runtime environment is appropriate for the adaptive execution (AE) optimizations (Section 5.10).

Disabling this option causes the compiler to use the optimizations without performing any checks of the runtime environment.

Disabling AE might be required on programs that run on bare metal. Otherwise, it should not be done for programs that run on non-Qualcomm cores.

NOTE: If the AE plugin is removed, these options become unavailable and will generate the error message, `Unknown argument`, if they are specified.

4.3.29 Compiler security

--analyze

Invokes the static program analyzer (Section 8.10.1) on the specified input files.

-analyzer-checker=*checker*

Enables 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

Lists 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*

Disables 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

Generates the static analyzer output report in HTML format.

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

--analyzer-Werror

Converts all static analyzer warnings into errors.

--compile-and-analyze *dir*

Invokes the static program analyzer on an entire program.

The analysis report files are written to the specified directory.

To prevent the analyzer from generating HTML files and instead print a summarized report on the `stdout`, replace `dir` with the sub-flag, `--analyzer-perf`.

--compile-and-analyze-high

Invokes the static program analyzer on an entire program with only a small list of high priority checkers.

The analysis report files are written to the specified directory. If you use the sub-flag, `--analyzer-perf`, they are printed to the `stdout`.

--compile-and-analyze-medium

Invokes the static program analyzer on an entire program with a list of high+medium priority checkers.

The analysis report files are written to the specified directory. If you use the sub-flag, `--analyzer-perf`, they are printed to the `stdout`.

-ffcfi

Enables control-flow integrity checks (Section 8.9).

-fno-fcfi

Disables control-flow integrity checks.

4.3.30 LLVM 4.0-specific compiler flags

Table 4-8 LLVM 4.0 release: New compiler flags

Flag	Description
`-fcoroutines-ts`	Enables support for the C++ Coroutines TS
`-fno-coroutines-ts`	Disables support for the C++ Coroutines TS
`-fdebug-info-for-profiling`	Emits extra debug information to make sample profile more accurate
`-fno-debug-info-for-profiling`	Does not emit extra debug information for sample profiler
`-fbuiltin-module-map`	Loads the clang built-in intrinsics module map file
`-fdiagnostics-show-hotness`	Enables profile hotness information in a diagnostic line
`-fno-sanitize-address-use-after-scope`	Disables use-after-scope detection in AddressSanitizer
`-fsanitize-thread-memory-access`	Enables memory access instrumentation in ThreadSanitizer (default)
`-fno-sanitize-thread-memory-access`	Disables memory access instrumentation in ThreadSanitizer
`-fsanitize-thread-func-entry-exit`	Enables function entry/exit instrumentation in ThreadSanitizer (default)
`-fno-sanitize-thread-func-entry-exit`	Disables function entry/exit instrumentation in ThreadSanitizer
`-fsanitize-thread-atomics`	Enables atomic operations instrumentation in ThreadSanitizer (default)
`-fno-sanitize-thread-atomics`	Disables atomic operations instrumentation in ThreadSanitizer
`-fno-experimental-new-pass-manager`	Disables an experimental new pass manager in LLVM.
`-flto-jobs=`	Controls the back-end parallelism of `-flto=thin` (default of 0 means the number of threads will be derived from the number of CPUs detected)
`-fprebuilt-module-path=`	Specify the prebuilt module path
`-fmodules-disable-diagnostic-validation`	Disable validation of the diagnostic options when loading the module
`-fmodules-ts`	Enable support for the C++ modules TS
`-fno-diagnostics-show-hotness`	Disable profile hotness information in diagnostic line
`-fdiagnostics-absolute-paths`	Print absolute paths in diagnostics
`-frelaxed-template-template-args`	Enable C++17 relaxed-template template argument matching
`-fno-relaxed-template-template-args`	Disable C++17 relaxed-template template argument matching
`-faligned-allocation`	Enable C++17-aligned allocation functions
`-fno-aligned-allocation`	Disable C++17-aligned allocation functions
`-faligned-new`	Enable C++17-aligned allocation functions
`-fno-aligned-new`	Enable C++17-aligned allocation functions
`-fropi`	Enable Arm read-only position independence relocation model
`-fno-ropi`	Disable Arm read-only position independence relocation model

Table 4-8 LLVM 4.0 release: New compiler flags (cont.)

Flag	Description
`-frwpi`	Enable Arm read-write position independence relocation model
`-fno-rwpi`	Disable Arm read-write position independence relocation model
`-fpreserve-as-comments`	Preserve comments in inline assembly
`-fno-preserve-as-comments`	Do not preserve comments in inline assembly
`-fdebug-macro`	Emit macro debug information
`-fno-debug-macro`	Do not emit macro debug information
`-fsave-optimization-record`	Generate a YAML optimization record file
`-fno-save-optimization-record`	Do not generate a YAML optimization record file
`-foptimization-record-file=`	Specify the filename of any generated YAML optimization record
`-fstrict-return`	Always treat control flow paths that fall off the end of a non-void function as unreachable
`-fno-strict-return`	Do not treat control flow paths that fall off the end of a non-void function as unreachable
`-fsplit-dwarf-inlining`	Place debug types in their own section (ELF Only)
`-fno-split-dwarf-inlining`	Do not place debug types in their own sections (ELF only)
`-mlong-calls`	Generate branches with extended addressability, usually via indirect jumps.
`-mno-long-calls`	Restore the default behavior of not generating long calls
`-mexecute-only`	Disallow generation of data access to code sections (Arm only)
`-mno-execute-only`	Allow generation of data access to code sections (Arm only)
`-mpure-code`	Alias for `mexecute_only`
`-mno-pure-code`	Alias for `mno_execute_only`
`-mpie-copy-relocations`	Use copy relocations support for PIE builds
`-mno-pie-copy-relocations`	Do not use copy relocations support for PIE builds
`-mfentry`	Insert calls to `fentry` at function entry (x86 only)
`-save-stats`	Save LLVM statistics
`-precompile`	Precompile only the input

4.3.31 Control diagnostic 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 on the console.

4.3.31.1 Control how diagnostics are displayed

When LLVM emits a diagnostic message, it includes rich information in the output and provides 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 that shows exactly where the diagnostic occurs in the 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 4.3.7.

4.3.31.2 Diagnostic mappings

All diagnostics are mapped into one of the following classes:

- Ignored
- Note
- Warning
- Error
- Fatal

4.3.31.3 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 that generate a large number of errors or warnings in a grouped way.

Categories are not shown by default, but you can turn them on with the `-fdiagnostics-show-category` option (Section 4.3.7). 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: You can get the mapping of category names to category identifiers by invoking LLVM with the `-print-diagnostic-categories` option.

4.3.31.4 Control 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.

Disable warnings for a specific condition 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 4.3.5). If `-Werror` is specified without a condition, *all* the warnings are changed to errors. `-ferror-warn` changes only 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.

4.3.31.5 Control 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 can 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 useful when writing a header file that will be compiled by other people, because you do not know what warning flags they build with.

In the following 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. Thus, it is possible to use push and pop around GCC-compatible diagnostics—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.

4.3.31.6 Control 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 you can override this in several ways.

- Use the `system_header` pragma 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
  ```

- Use the `-isystem-prefix` and `-ino-system-prefix` options 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 that 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 that 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.

4.3.31.7 Enable all warnings

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

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

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

4.4 Use GCC cross-compile environments

The LLVM compilers are standalone compilers that rely on an existing system tools root environment—also known as *sysroot*—for accessing include files and libraries. 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 4.3.15). The target triple directory typically contains a `libc` directory that 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.`, which is specified with the compile option, `--sysroot`.

The LLVM compilers also require the location of the GNU linker (if they are not using `qcld`). This location is specified with the `-B` or `-gcc-toolchain` option, and it 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 -B GCC-top foo.c

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

With C++, it might be necessary to add a set of C++ include directories so the LLVM compilers can correctly search for the header files. 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, and so on).

This target triple might differ from the target triple used on the compiler command line.

NOTE: For help using sysroots, contact llvm-arm-support@qualcomm.com.

4.5 Use LLVM with GNU Assembler

Snapdragon LLVM includes support for using the GNU Assembler (GAS) as the system assembler. The `-mllvm -enable-android-compat` and `-mllvm -force-div-attr` options (Section 4.3.19) 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, it might 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, specify the `-mllvm -enable-android-compat` option.

- GCC 4.6 and 4.7 releases

  The DIV attribute is emitted whether or not the `-mhwdiv` option 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 `-mllvm -force-div-attr` option.

- Post GCC 4.7 releases

  The DIV attribute is emitted only when the `-mhwdiv` option is specified.

4.6 Built-in functions

The `arm_memcpy_bias.h` file contains the declaration of a specialized Arm 32-bit `memcpy` built-in function for a copy size of 1024.

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

Using this function in the source results in generated code with a runtime check for a copy size of 1024, and the inlining of `memcpy` for a copy size of 1024 using vector instructions.

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

4.7 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.

4.7.1 View phases

To view these phases during compilation, invoke the compiler using the `-ccc-print-phases` option (Section 4.3.2). 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}
```

4.7.2 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 4.3.2). 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" "-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"
```

4.7.3 Specify phase options

To pass a command option to the tool that performs a specific compilation phase, invoke the compiler using the `-X` option (Section 4.3.2). For example:

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

In this example, `-X` is used to pass the `--print-map` option 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, you must 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 `-analyzer-output html` option to the compiler.

5 Code optimization

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

NOTE: We strongly recommend that you try using the various code optimizations to improve the performance of your program. Using only the default optimization settings might result in suboptimal performance.

5.1 Optimize for performance

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

Table 5-1 Options to use for optimizing code performance

Core	Options
Armv7	`-Ofast -mcpu=krait`
Armv8 (AArch32)	`-Ofast -mcpu=cortex-a57`
Armv8 (AArch64)	

For more information on `-Ofast`, see Section 4.3.21.

5.2 Optimize for code size

LLVM currently generates the smallest code when compiling for Thumb-2 mode.

NOTE: Thumb-2 is available only on Armv6-T2 core, Armv7 core, and AArch32.

Table 5-2 Options to use for optimizing code size

Core	Options
Armv7	`-Osize -mcpu=krait`
Armv8 (AArch32)	`-Os -mcpu=cortex-a57`
Armv8 (AArch64)	

For the Armv7 core, the `-Osize` option is preferred over `-Os` because it enables additional optimizations for code size. For more information on `-Osize`, see Section 4.3.22.

5.3 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 4.3.20).

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 CPU that has this extension with the `-mcpu` flag) or explicitly with `-mfpu=neon`.

The following example is 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.

The vectorizer can also be enabled using the `-ftree-vectorize` option, which is an alias for `-fvectorize-loops`. The vectorizer currently operates only on the innermost loop of a nested loop.

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

5.4 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 4.3.21).

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 that is linked into the final executable image. The purpose of the component is to initialize a new thread at program initialization time, and subsequently to 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 are 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.

5.4.1 Auto-parallelization with SYMPHONY library

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

NOTE: SYMPHONY can be used only with Android applications.

5.4.1.1 SYMPHONY usage

To perform auto-parallelization with SYMPHONY, use the `-fparallel-symphony` option (Section 4.3.21).

`-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). We recommend using `-fparallel-symphony` only with the highest optimization level (`-Ofast`). However, it can be used with lower optimization levels.

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.

5.4.1.2 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 cannot 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 5-3 SYMPHONY library versions for supported development platforms

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`

Use the Android `adb` utility 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)
```

5.4.1.3 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.

5.5 Merge functions

LLVM includes support for function merging. By default, this optimization is disabled; to enable it, use the `-fmerge-functions` option (Section 4.3.19).

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 x;
   x = a + 4;
   return x * b;
}
```

```
int f2(int a, int b) {
   int x;
   x = a + 10;
   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 might have a negative impact on program performance, it is disabled by default, and becomes enabled only when it is specified explicitly.

5.6 Link-time optimization

Link-time optimization (LTO) comprises a set of powerful inter-modular optimizations that 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 that 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`) that 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.

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 might support LTO for Armv8 targets, depending on the GCC toolchain/sysroot version used. LTO is not supported on Windows using the Gold linker.

For more information on the Snapdragon LLVM Arm linker, see the *Qualcomm Snapdragon LLVM Arm Linker User Guide* (80-VB419-10).

For more information on the Gold linker, go to:

llvm.org/docs/GoldPlugin.html

5.6.1 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 can 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.

5.6.2 ThinLTO and incremental compilation

The ThinLTO compilation is a new type of LTO that supports incremental compilation. The regular monolithic LTO implementation merges all input bitcode files into a single module for whole-program analysis and optimization, which is time and memory consuming, and it prevents incremental compilations.

In ThinLTO mode, each bitcode file has additional summary information. During the link step, only the summaries are read, merged, and analyzed. With the use of a cache, incremental builds can be supported.

Usage:

- To use ThinLTO, simply add the `-flto=thin` option to compile and link commands:

  clang -flto=thin *input_file*

- To use incremental builds, add the cache option to link:

  clang -flto=thin -Wl,-flto-options=cache[=<*cache_dir*>**]** *input_files...* **-o** *output*

By default, the cache directory is `output.ltocache`.

5.6.3 LTO with fat objects

In addition to normal LTO, Snapdragon LLVM supports a mode that generates object files that can be used to build both non-LTO and LTO binaries. To enable this mode, pass `-fqc-embed-hbitcode` to the compiler. At link time, if you specify `-flto`, link-time optimization will be used; otherwise, link-time optimizations will be disabled.

5.7 Profile-guided optimization

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

1. A program is first executed to collect profile information on it.
2. 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 and 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 at:

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

5.7.1 Instrumentation-based PGO

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

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

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

1. Build the instrumented application.

   Use the compile option, `-fprofile-instr-generate`, to compile and link the application code. For example:

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

   `-fprofile-instr-generate` optionally accepts a filename argument that 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.

2. Generate profile information.

   Run the built application on your device to generate the profile information. For example, to run this application on Android, enter 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`.

3. Convert the profile information.

   Generate profile information in one of two ways:

   - Run the instrumented program once, which results in a single set of profile information
   - Run the instrumented 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
   ```

   This example inputs two raw profile files (`dataset-1.profraw` and `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`. The 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.

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

4. Rebuild the application using PGO.

   Enable PGO in your application builds by 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 5.7.5).

5.7.2 Instrumentation-based profile generation with Android apps

In instrumentation-based PGO, the collected profile data is typically written to a profile data file when the application exits. This default LLVM behavior registers the profile file synchronization (or write) operation with the C lib function, `atexit`, which is invoked when processes or shared libs are unloaded.

However, Android applications (APKs) typically do not have an exit mechanism. Therefore, to collect profile data while developing Android applications, alternative approaches are required.

Use the compile option, `-fprofile-instr-sync-interval`, (along with the other profile-generation options). This flag has an overloaded meaning. When combined with the `-fprofile-instr-file-sync-thread` flag, this option directs the compiler to create a service thread, which periodically syncs the collected profile data to the file at a user-specified interval (expressed in milliseconds).

However, some Android services might start early in the boot-up sequence, which might cause an issue when depending on non-light-weight runtime support for threads and timers at that time. In such a case, we recommend using the `-fprofile-instr-sync-interval` compiler option combined with the `-fprofile-instr-file-sync` option (the file synchronization thread service thread is disabled by default). In this setting, the option directs the compiler to instrument functions with a call to synchronize profile data to a file. The actual synchronization operation occurs periodically upon entering a function and when the number of times the function is entered exceeds the user-specified interval (expressed in the number of elapsed function calls).

The following example directs the compiler to synchronize collected profile data to the `/sdcard/default.profraw` file with a synchronization period of one second:

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

The following example directs the compiler to synchronize collected profile data to the `/sdcard/default.profraw` file at every1000000th execution of a function call:

```
clang++ -O2 -fprofile-instr-generate=/sdcard/default.profraw
-fprofile-instr-file-sync
-fprofile-instr-sync-interval=1000000 source.cc
```

In both examples, at every synchronization 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 postprocessing is performed with the llvm-profdata tool.

Control profile generation

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

The APIs must be added to the program source code. They explicitly control synchronizing of the collected profile data to the profile data file:

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

  Resets the profile data counters to zero, thus resetting the collected profile data.

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

  Synchronizes 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 require 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.

5.7.3 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 at:

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 these tools.

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 steps explain how to use Linux `perf` and `autofdo` to perform sampling-based PGO.

1. Build the application.

   Use the compile option, `-gline-tables-only`. For example:

   ```
   clang++ -gline-tables-only -O2 source.cc -o application
   ```

   The application must be compiled with `-gline-tables-only` (or `-g`) to ensure that the profile information maps accurately back to the source code.

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.

3. Convert the profile information.

   a. Install the `autofdo` tool.

   b. Convert the raw profiles into the required sample profile format. For example:

   ```
   create_llvm_prof --binary=./application --out=application.profile
   ```

4. Rebuild the application using PGO.

   Enable PGO in the application build by 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
   ```

   The application must be compiled with `-gline-tables-only` to ensure that the profile information maps accurately back to the source code.

   NOTE: Sample-based profile information can be used even as the user code changes over time (Section 5.7.5).

5.7.4 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. For detailed information on Snapdragon MDP, go to:

developer.qualcomm.com/mobile-development/development-devices/mobile-development-platform-mdp

One of the MDP developer features is the collection of sample-based profiles. Typically, a device must be rooted to collect sample data. However, MDP is preconfigured for this approach, 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 the PATH before using it.

The following procedure explains how to perform sampling-based PGO on a Snapdragon MDP.

1. Build the application.
   a. Use the compile option `-gline-tables-only`:

      ```
      clang++ -gline-tables-only -O2 source.cc -o application
      ```

   b. Move the resulting binary file to the MDP.
2. Generate profile information.
   a. Add `perf` to PATH (`perf` is pre-installed on a Snapdragon MDP):

      ```
      export PATH=/data/data/com.qualcomm.qview/:$PATH
      perf record -e cycles -c 10000 ./application
      ```

   b. Move the generated profile data files back to the host.
3. Convert the profile information.
   a. Install the `autofdo` tool on the host.
   b. Convert the raw profiles into the required sample profile format. For example:

      ```
      create_llvm_prof --binary=./application --out=application.profile
      ```

4. Rebuild the application using PGO.

   Enable PGO in the application build by using the profile data generated in step 3:

   ```
   clang++ -O3 -gline-tables-only -fprofile-sample-
   use=application.profile source.cc -o application
   ```

5.7.5 Profile resiliency

Profile information collected for PGO is associated back to your 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. You 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.

5.7.6 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 `-g` or `-gline-tables-only` options. These options help LLVM accurately associate the generated profiles to source code.

- PGO is resilient to changes in the 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 `-static` linker option, which is used to build static executables, is not supported.

- Profile data generated with `-fprofile-instr-sync-interval` might include a final profile counter section that is truncated. This can result in warnings or errors while postprocessing with the llvm-profdata tool.

  In this case, the messages can be ignored because all the preceding profile data sections were handled correctly by the llvm-profdata tool. The postprocessed output is thus valid and usable for PGO.

- When a program is compiled with `-fprofile-instr-generate`, `errno` might not be initially set to zero at the instrumented executable's startup.

- On Android targets, PGO dumps data to logcat and requires the log library to be either statically (`-llog` is passed to the linker flags) or dynamically linked.

5.8 Loop optimization pragmas

The compiler supports auto-vectorization pragmas that can be used to selectively enable and disable the following loop transformations:

NOTE: The compiler always verifies the correctness of any transformation, and it will not vectorize a loop unless it can prove it is safe to do so.

5.8.1 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]
```

For detailed descriptions of the supported loop pragmas listed in the following table, see Section 5.8.3 and Section 5.8.4.

Table 5-4 Supported vectorization loop pragmas

Name	Description
`vectorize(enable)`	Enable auto-vectorization for a loop.
`vectorize(disable)`	Disable auto-vectorization for a loop.
`vectorize(assume_safety)`	Enable auto-vectorization for a loop without verifying auto-vectorization correctness.
`vectorize_width(N)`	Enable auto-vectorization for a loop with the specified vector factor N. The vector factor is the number of iterations that are executed in parallel. NOTE: The value N must be a power of 2.

5.8.2 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 (such as `-fvectorize-loops`) or implicitly with an optimization level (for example, 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 `-floop-pragma` option. 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 5-5 Loop pragma options that enable auto-vectorization

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 requiring an additional compile option.

Following are the command option combinations that can enable auto-vectorization.

Table 5-6 Loop pragma option combinations

Combination	Description
`-fvectorize-loops -floop-pragma`	Enable auto-vectorization for all eligible loops.

5.8.3 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(assume_safety)
#pragma clang loop vectorize_width(N)
```

NOTE: These are the same vectorization pragmas that are supported by the LLVM community compiler.

5.8.3.1 #pragma clang loop vectorize(enable)

Enable vectorization for a loop.

This pragma is useful for vectorizing specific loops that would otherwise not be considered profitable by the compiler.

Example 1: Potential code bloat

The following loop is not profitable because of potential large code bloat. The pragma overrides the compiler profitability heuristic and enables auto-vectorization.

```
void foo(char *A, char *B, char *C, int n) {
  #pragma clang loop vectorize(enable)
  for (int i = 0; i < n; i++) {
    A[5*i] += B[i] * C[i];
    A[5*i+1] += B[i] + C[i];
    A[5*i+2] += B[i] - C[i];
    A[5*i+3] += B[i] * C[i];
  }
}
```

Example 2: Force auto-vectorization

This pragma can be useful for forcing auto-vectorization of a specific loop in the loop nest. If you know which loop level is hot, you can override the default compiler heuristic.

Specify an outer loop (where n is known to be very large):	Specify an inner loop (where `m` is known to be very large):

`int A[5000][5000];`
`int B[5000][5000];`
`int C[5000];`
`void foo (int n, int m) {`
`#pragma clang loop vectorize(enable)`
`for (int i = 0; i < n; i++)`
`for (int j = 0; j < m; j++)`
`A[i][j] = B[j][i];`
`}`
	`int A[5000][5000];`
	`int B[5000][5000];`
	`int C[5000];`
	`void foo (int n, int m) {`
	`for (int i = 0; i < n; i++)`
	`#pragma clang loop vectorize(enable)`
	`for (int j = 0; j < m; j++)`
	`A[i][j] = B[j][i];`
	`}`

Example 3: Ignored loop

If a loop is annotated with this pragma and the compiler proves that this annotation is illegal, the compiler ignores the loop and sends a warning.

```
int foo(char *A) {
  int e = A[0], s = 0, i = 0;
  #pragma clang loop vectorize(enable)
  while (i != e) {
    e = A[i];
    i++;
    s += e;
  }
  return s;
}
```

The following warning is issued for this example:

```
warning: loop not vectorized: failed explicitly specified loop
vectorization [-Wpass-failed=loop-vectorize]
```

5.8.3.2 #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.

5.8.3.3 #pragma clang loop vectorize(assume_safety)

Check that a loop is safe for vectorization.

The compiler typically generates runtime legality checks to ensure safety. The compiler does not generate checks for the loop specified by this pragma. It is your responsibility to ensure that the loop can be legally vectorized.

Example 1: Pointer aliasing

For vectorization to be done safely, the compiler generates aliasing checks to ensure that the pointers do not alias each other. If you know the pointers can never alias, you can specify a restrict keyword to the pointers or use this pragma. The checks can be expensive if there are many pointers in the loop.

```
void foo(char *A, char *B, char *C, int n) {
#pragma clang loop vectorize(assume_safety)
  for (int i = 0; i < n; i++) {
    A[i] += B[i] * C[i];
  }
}
```

In this code example, the compiler generates checks to make sure A does not alias with B or C for successful vectorization. With the pragma, no checks are generated.

Example 2: Data dependence checks

In the following code, the compiler generates checks to ensure that `m` is larger than the vector width. With the pragma, no checks are generated.

```
void foo1(char *A, int n, int m) {
#pragma clang loop vectorize(assume_safety)
  for (int i = 0; i < n; i++) {
    A[i+m] = A[i];
  }
}
```

Example 3: Ignored loop

If a loop is annotated with this pragma and the compiler proves that this annotation is illegal, the compiler ignores the loop and sends a warning.

```
int foo(char *A) {
  int e = A[0], s = 0, i = 0;
  #pragma clang loop vectorize(assume_safety)
  while (i != e) {
    e = A[i];
    i++;
    s += e;
  }
  return s;
}
```

The following warning issued for this example:

```
warning: loop not vectorized: failed explicitly specified loop
vectorization [-Wpass-failed=loop-vectorize]
```

5.8.3.4 #pragma clang loop vectorize_width(N)

Set the 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.

5.8.4 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 5-7 shows the interaction between reporting, options, and loop pragmas. A check mark 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 5-7 Loop optimization reporting

-fvectorize-loops	-floop-pragma	Report content
X	X	No reporting
X	4	Report on vectorization results only for loops with enable pragmas
4	X	Report vectorization results only
4	4	Report vectorization results for all loops
X	4	Report vectorization results only for loops with enable pragmas
4	X	Report vectorization results for all loops
4	4	Report vectorization results for all loops

This table 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.

5.8.5 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.

5.8.5.1 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

Typically, 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;
}
```

5.8.5.2 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;
}
```

5.8.5.3 Vectorize a non-profitable loop

The auto-vectorizer might 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;
  }
}
```

5.8.5.4 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;
}
```

5.9 Optimization reports

Optimization reports are a new compiler reporting mode that 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 you 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 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 specified optimization.

Using optimization reports to analyze a loop is an iterative process. There might 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 might 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 at:

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 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.

5.9.1 Example output

The following example of an optimization report shows the messages you 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=cortex-a57 -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)
  ^
```

5.9.2 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 you can take to avoid the problem and vectorize the loop.

5.9.2.1 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 you are 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 you 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) {
```

5.9.2.2 Non-affine loop bound

The loop optimizer requires all loop bounds to be affine (meaning that the number of iterations of the loop cannot be analyzed), which is a linear function of the loop induction variable. If the loop bound is not affine, 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 structures. Contrast this case with a standard `for` loop (such as `(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 might wrap, it is impossible for the compiler to compute how many iterations the loop might execute.

This problem can be fixed by using an int for the loop variable. Unlike unsigned integers, 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) {
  ^
```

5.9.2.3 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) {
    ^
```

5.9.2.4 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. You 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 the bar() function. 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)
  ^
```

5.9.2.5 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 that 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)
  ^
```

5.9.2.6 Base pointer not loop invariant

This message indicates that a pointer used to access memory might change during execution of the loop. To successfully vectorize a loop, the compiler depends on having base values that do not move during the loop. The problem might 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 structure 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)
  ^
```

5.9.2.7 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 (for example, `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, try to remove the double indirection in order to vectorize the loop.

5.9.2.8 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 might be more costly than the benefit gained from vectorizing the loop.

To fix this error, increase the number of allowed runtime checks by using the `-mllvm -polly-max-pointer-aliasing-checks` option, 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 by using the following option to vectorize the loop:

```
-mllvm -polly-max-pointer-aliasing-checks=5
```

Alternatively, `restrict` can 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]
```

5.10 Adaptive execution

Adaptive execution (AE) controls a set of optimizations that have been specially characterized and tuned for Qualcomm Snapdragon cores.

The performance of these optimizations has not been characterized on non-Qualcomm cores. As a result the optimizations are normally not used on such cores.

AE provides default settings for controlling the optimizations, and compiler options for overriding the defaults.

5.10.1 Default settings

AE is enabled by default for Android targets. It is disabled by default for all other targets.

NOTE: We do not recommend disabling AE for Android programs that can be run on non-Qualcomm cores, either intentionally or otherwise.

NOTE: Disabling AE might be required on programs that run on bare metal.

5.10.2 Options

The default AE settings can be changed with the compiler options `-mae` and `-mno-ae` (Section 4.3.28).

`-mae`

Directs the compiler to generate code checks that verify that the runtime environment is appropriate for executing the AE optimizations. This option is used when the generated code might execute on non-Qualcomm cores.

`-mno-ae`

Directs the compiler to generate the AE optimizations without any code checks to verify the runtime environment. In this case, executing the application on a non-Qualcomm processor will result in the use of optimizations that have not been characterized for the processor.

5.10.3 Plug-ins

AE is enabled by an LLVM compiler plug-in that is automatically installed as part of the Snapdragon LLVM installation.

The plugin is stored at the following location:

install_dir/lib/libLLVMAE.so (Linux)

install_dir/lib/libLLVMAE.dll (Windows)

NOTE: If the plug-in is removed, AE is automatically enabled for all targets, and it cannot be disabled.

6 Bare metal environment support

A bare metal environment is a software execution environment where the software image is executed directly on the hardware with no operating system support. All the required software mechanisms for performing operation system functionality such as system calls, image loading, and relocation are handled by code that is part of the software image rather than through an operation system.

Bare metal environments are typically used for boot images as well as firmware that must execute as soon as the system boots up, well before a full-fledged HLOS operating system (such as Linux, Android, or Windows) is brought up. Hence, bare metal images must include all the functionality such as startup code and memory management API as part of the image. In addition, the image layout is typically finalized at link time because runtime relocation is not common in bare metal environments.

This Snapdragon LLVM Arm Toolchain contains full support for the following popular Arm processors:

- Armv8 architecture processors with all extensions up to 8.7
- Cortex-M0 processor (Armv6-M architecture version)
- Cortex-M3 processor (Armv7-M architecture version)
- Cortex-A family of processors (Armv7 architecture version)
- 64-bit cores (AARCH64 architecture version)

For each target group, the Snapdragon LLVM toolchain includes a set of runtime C and C++ libraries specifically built for the target. In addition, there are several features (as explained in customization hooks, Section 6.4) available in the toolchain to facilitate bare metal image programmers to enable end-to-end development.

6.1 Compiler options specific to bare metal

The following options are introduced to support automatically adding the correct bare metal includes and libraries. You are not required to explicitly specify paths to bare metal includes or libraries if you use these options.

`-target {`*`arch`*`}-none-{`*`ABI`*`}`

Set the target architecture and ABI for code generation. We recommend the following values for the bare metal environment:

- *`arch`* can be `armv5`, `armv6m`, `armv7m`, `armv7`, or `aarch64`.
- For 32-bit targets, *`ABI`* can be `eabi` if libc is not needed, or `musleabi` if SDLLVM libc is needed.
- For 64-bit targets, *`ABI`* can be `elf` if libc is not needed, or `musleabi` if SDLLVM libc is needed.

`-fuse-baremetal-inc`

Adds the correct bare metal includes for the target during compilation.

NOTE: This option is OFF by default. It will not be enabled if `-nostdinc` or `-nobuiltininc` are specified on the command line.

Use the option as follows:

```
clang -target armv7m-none-eabi -fuse-baremetal-inc hello.c -c
```

This option adds the following on the compilation line:

```
-isystem <llvm_install_dir>/armv7m-none-eabi/libc/include
```

`-fuse-baremetal-libs`

Adds the correct bare metal libraries for the target during compilation.

NOTE: This option is OFF by default. It will not be enabled if `-nostdlib` or `-nodefaultlibs` are specified on the command line.

Use the option as follows:

```
clang -target armv7m-none-eabi -fuse-baremetal-libs hello.c
```

This option adds the following libraries:

```
-L<llvm_inEopstall_dir>/armv7m-none-eabi/libc/lib
-L<llvm_install_dir>/armv7m-none-eabi/lib
-lunwind
```

If the code being linked is C++, the clang++ driver is invoked. Thus, the following libraries will be added to the previous list of libraries:

```
-lc++
-lc++abi
```

-fuse-baremetal-rtlib

Add the correct built-ins (`compiler-rt`) library for the target during linking.

NOTE: This option is OFF by default. It will not be enabled if `-nostdlib` or `-nodefaultlibs` are specified on the command line.

Use the option as follows:

```
clang -target armv7m-none-eabi -fuse-baremetal-rtlib hello.c
```

The following libraries are added:

```
-L<llvm_install_dir>/lib/clang/<version>/lib/baremetal
--start-group -lc -lclang_rt.builtins-armv7m --end-group
-mfloat-abi=soft and -mfpu=none
```

Both `-mfloat-abi=soft` and `-mfpu=none` turn off floating point instruction generation. As a result, and only for the Armv7-M architecture, if either of these flags are present on the command line, link with the nofp (no floating point) version of the compiler-rt library.

Use the options as follows:

```
clang -target armv7m-none-eabi -fuse-baremetal-rtlib
-mfloat-abi=soft hello.c

clang -target armv7m-none-eabi -fuse-baremetal-rtlib -mfpu=none
hello.c

-L<llvm_install_dir>/lib/clang/<version>/lib/baremetal

--start-group -lc -lclang_rt.builtins-armv7m-nofp --end-group
```

-fuse-baremetal-crt

Add the correct entry point and initialization routines library (`crt1.o`) for the target during linking.

NOTE: This option is OFF by default. It is not enabled if `-nostdlib` or `-nodefaultlibs` is specified on the command line.

Use the option as follows:

```
clang -target armv7m-none-eabi -fuse-baremetal-crt hello.c
```

This adds the following object file on the command line:

```
<llvm_install_dir>/armv7m-none-eabi/libc/lib/crt1.o
```

-fuse-baremetal-sysroot

Instead of specifying the `-fuse-baremetal-inc`, `-fuse-baremetal-libs`, `-fuse-baremetal-rtlib`, or `-fuse-baremetal-crt` options separately, you can specify the `-fuse-baremetal-sysroot` option to get the correct includes and libraries for the target.

NOTE: This option is OFF by default. It is not enabled if `-nostdlib` or `-nodefaultlibs` are specified on the command line.

Use the option as follows:

```
clang++ -target armv7m-none-eabi -fuse-baremetal-sysroot
hello.cpp
```

This option adds the following includes and libraries:

```
-isystem <llvm_install_dir>/armv7m-none-eabi/libc/include
-L<llvm_install_dir>/armv7m-none-eabi/libc/lib
-L<llvm_install_dir>/armv7m-none-eabi/lib
-lunwind
-lc++
-lc++abi
-L<llvm_install_dir>/lib/clang/<version>/lib/baremetal
--start-group -lc -lclang_rt.builtins-armv7m --end-groupr>/
armv7m-none-eabi/libc/lib/crt1.o
```

-baremetal-lib-variant=*variant*

Specify the variant of libraries to be used.

For example, for the Armv6-M architecture, `-baremetal-lib-variant=nofltprint_nopthread_zeroinbss` allows the use of a special variant of libc that has no thread support and no floating number formatting, and zero-initialized data is placed in BSS sections.

NOTE: This option is OFF by default. It is only effective if `-fuse-baremetal-libs` or `fuse-baremetal-sysroot` is used, and the corresponding architecture has the specified variant available.

6.1.1 Notes

- Each option listed in Section 6.1 can be explicitly turned OFF using the `-fno-baremetal-<flagname>` option. For example:

  ```
  clang++ -target armv7m-none-eabi -fuse-baremetal-sysroot
  -fno-use-baremetal-inc
  ```

  This example adds all the required bare metal libraries except the bare metal includes.

  ```
  clang++ -target armv7m-none-eabi -fuse-baremetal-sysroot
  -fno-use-baremetal-libs
  ```

  This example adds all the required bare metal includes, rtlib, and crt. It will not include any of the bare metal C or C++ libraries.

  ```
  clang++ -target armv7m-none-eabi -fuse-baremetal-sysroot
  -fno-use-baremetal-rtlib
  ```

  This example adds all the required bare metal includes and libraries except the bare metal built-ins (`compiler-rt`) library.

- If multiple conflicting versions of an option are specified, the rightmost option prevails. For example:

  ```
  clang++ -target armv7m-none-eabi -fuse-baremetal-sysroot
  -fno-use-baremetal-sysroot
  ```

  In this example, no bare metal include or library is added because the right-most option (`-fno-baremetal-sysroot`) prevails.

6.1.2 Bare metal linker option

`-fuse-ld=qcld`

The linker used for bare metal is `ld.qcld`. Specify this `-fuse-ld=qcld` option to invoke this linker.

Use the option as follows:

```
clang -target armv7m-none-eabi -fuse-ld=qcld
-fuse-baremetal-sysroot hello.c
```

This option invokes *<llvm_install_dir>*`/bin/ld.qcld` for linking.

6.1.3 Commonly used options

We recommend the following options for reducing the code size of a bare metal application:

`-fomit-frame-pointers`
`-fno-exceptions` **(for c++)**
`-fshort-enums`
`-mno-interrupt-stack-align`

Disables the stack alignment for interrupts (IRQ, FIQ, and SWI) if the function does not have any other user-defined attribute. By default, stack alignment is enabled for all functions.

Other options:

`-m{no-}unaligned-access`

Enables/disables accessing 16-bit or larger data from a memory address that might be unaligned. By default, unaligned access is enabled except for all pre-Armv6 and Armv6-M architectures.

`-fno-zero-initialized-in-bss`

Normally, zero-initialized values are placed in BSS (ZI) sections. This option disables this behavior and that data will be placed in regular data sections. This is useful when the BSS sections have not been zero-initialized when the data is used.

`-mgeneral-regs-only`

Prevents the compiler from using floating-point and advanced SIMD registers for AArch64.

6.2 Toolchain components specific to bare metal

The toolchain comes with the following libraries built for the Armv5, Armv6-M, Armv7-M, Armv7, and AArch64 architectures:

- `libc` – An implementation of the ISO and POSIX C standard library based on the MUSL C library.
- `libc++` – An implementation of the C++ standard library that targets C++11. `libc++` is part of the LLVM family of projects.
- `libc++abi` – An implementation of low-level support for a standard C++ library. This project is part of the LLVM family of projects.
- `libunwind` – An implementation of stack unwinder used for C++ exception handling. This project is part of the LLVM family of projects.
- `libclang` – An implementation runtime library calls generated by the LLVM compiler. This project (`compiler-rt`) is part of the LLVM family of projects.

6.2.1 Location of bare metal libraries

Naming conventions:

- The `{arch}` part in the locations should be one of the following options: `armv5`, `armv6m`, `armv7m`, `armv7`, `aarch64`.
- The `{abi}` part in the locations should be `eabi` for 32-bit and `elf` for 64-bit (AArch64).

Locations:

- `libc`
  - Location is <*llvm_install_dir*>/{*arch*}-none-{*abi*}/lib/libc
  - Header files are in the `include` subdirectory
  - Libraries are in the `lib` subdirectory:
    - `libc.a` for the C library
    - `crt1.o` for the C runtime library
- `libc++`, `libc++abi`, `libunwind`
  - Location is <*llvm_install_dir*>/{*arch*}-none-{*abi*}
  - Header files are in the `include` subdirectory
  - Libraries are in the `lib` subdirectory:
    - `libc++.a` for libc++
    - `libc++abi.a` for libc++abi
    - `libunwind.a` for libunwind

- libclang_rt.builtins
  - Location is <*llvm_install_dir*>/lib/clang/<*version*>/lib/baremetal
    - For example, lib/clang/<*version*>/lib/baremetal
    - libclang_rt.builtins-{*arch*}.a
- libclang_rt.builtins-armv7-nofp.a
  - For Armv7 targets without hardware floating-point
- libclang_rt.builtins.Ospace-armv7m.a
  - For Armv7-M targets that prioritize for code size

6.2.2 C library for bare metal

6.2.2.1 Implemented functions and headers

The C library for bare metal implements functions specified by ISO/IEC 9899. Each library function is declared with a prototype in a header. The header declares a set of related functions, necessary types, and macros. Following are the headers provided by the C library:

```
<assert.h>
<complex.h>
<ctype.h>
<errorno.h>
<fenv.h>
<float.h>
<inttypes.h>
<iso646.h>
<limits.h>
<locale.h>
<math.h>
<signal.h>
<stdarg.h>
<stdbool.h>
<stddef.h>
<stdint.h>
<stdio.h>
<setjmp.h>
<stdlib.h>
<string.h>
<tgmath.h>
<time.h>
<wchar.h>
<wctype.h>
```

Some functions specified by the ISO/IEC 9899 standard require syscalls. These functions are not supported unless they are explicitly documented as supported (such as semihosting functions). In addition, some non-standard functions and headers are implemented for convenience.

Non-standard functions

`strlcpy(char *dst, const char *src, size_t size`

Size-bounded string copying.

`strlcat(char *dst, const char *src, size_t size)`

Size-bounded string concatenating.

`strnlen(const char *str, size_t size)`

Size-bounded string length calculation.

`strtok_r (char *str, const char *delim, char **saveptr)`

Parses a string into a sequence of tokens.

`memalign(size_t boundary, size_t size)`

Allocates a block of size bytes whose address is a multiple of boundary.

Non-standard headers

```
elf.h
pthread.h
```

6.2.2.2 Bare metal-specific features

The C library is optimized for code size by default. More specifically, the default C library is built as follows:

- Use smallest data types for enum types (`-fshort-enums`)
- Hidden symbol visibility by default (`-fvisibility=hidden`)
- Limited formatted printing (`printf`, `vfprint`) support:
  - No wide-character formatting; that is, `%S` and `%C` modifiers are not supported
  - Non-standard modifiers like `%m` are not supported
  - Some library variants have no floating-point number formatting; that is, `%f, %F, %e, %E, %g, %G, %a, %A` modifiers are not supported

6.3 Compiler built-ins for bare metal

Built-ins (also called *intrinsics*) are functions built into the compiler. They are syntactically similar to regular functions. The compiler lowers them into the appropriate instructions.

Syntax:

```
void __builtin_arm_breakpoint (innnnt n)
```

Example:

```
void foo() {
    __builtin_arm_breakpoint(0x2A);
}
```

This call to `__builtin_arm_breakpoint` generates the `BKPT` instruction with immediate operand `0x2A`.

Following are the built-ins supported by LLVM for bare metal. The built-ins are shown with their signature (return types and parameter types).

Table 6-1 Built-ins supported by LLVM for bare metal

Built-in	Description
`void __builtin_arm_breakpoint(unsigned int)`	Generates a `BKPT` instruction.
`unsigned int __builtin_arm_clz(unsigned int)`	(Count Leading Zeros) Returns the number of leading zero bits in a value.
`unsigned int __builtin_arm_current_pc()`	Returns the current value of the program counter.
`unsigned int __builtin_arm_current_sp()`	Returns the current value of the stack pointer.
`void __builtin_arm_dbg(unsigned int)`	Generates a `DBG` instruction.
`void __builtin_arm_dmb(unsigned int)`	Generates a `DMB` (data memory barrier) instruction.
`void __builtin_arm_dsb(unsigned int)`	Generates a `DSB` (data synchronization barrier) instruction.
`void __builtin_arm_isb(unsigned int)`	Generates an `ISB` (instruction synchronization barrier) instruction.
`void __builtin_arm_nop()`	Generates a `NOP` instruction.
`void __builtin_arm_prefetch(const int *, unsigned int, unsigned int)`	Generates data prefetch instructions to minimize cache-miss latency by moving data into the cache before it is accessed.
`int __builtin_arm_rsr(const char *)`	Reads a 32-bit system register (special register).
`unsigned long long int __builtin_arm_rsr64(const char *)`	Reads a 64-bit system register (special register).
`void * __builtin_arm_rsrp(const char *)`	Reads a system register containing an address.
`void __builtin_arm_wfe()`	Generates a `WFE` (wait for event) instruction.
`void __builtin_arm_wfi()`	Generates a `WFI` (wait for interrupt) instruction.
`void __builtin_arm_wsr(const char *, unsigned int)`	Writes a 32-bit system register (special register).
`void __builtin_arm_wsr64(const char *, unsigned long long int)`	Writes a 64-bit system register (special register).

Table 6-1 Built-ins supported by LLVM for bare metal (cont.)

Built-in	Description
`void __builtin_arm_wsrp(const char *, void *)`	Writes a system register containing an address.
`void * __builtin_return_address(unsigned int)`	Returns the return address of a function on the call stack. The input parameter is the call depth; 0 means the current function.
`void __disable_irq()`	Disables IRQ interrupts.
`void __enable_irq()`	Enables IRQ interrupts and clears the I-bit in the current program status register (CPSR). For Cortex-M profile processors, clears the exception mask register (PRIMASK).
`unsigned int __ror(unsigned int, unsigned int)`	Generates an `ROR` (rotate operand right) instruction. It right rotates a value by the specified number of places.

6.4 Customization hooks for bare metal images

6.4.1 __attribute__((at(address, [prefix])))

This attribute specifies the absolute address of the variable, where `address` is the expected address, and the optional `prefix` can be used to customize the section name.

The compiler places the variable in its own section named `prefix._AT_@address`. The linker automatically places the section in the specified address.

For example:

```
#define BASE 0x4000
#define OFFSET 0x100
#define ADDR (BASE + OFFSET)
int v __attribute__((at(ADDR))) = 4;
```

The compiler will place variable `v` in section `._AT_@0x4100`, and the linker will then place the section at address 0x4100.

For an uninitialized variable, using the `at` attribute will make it a regular data typed section rather than a BSS typed section. To generate a BSS section, the `.bss` prefix is required. For example:

```
int v0 __attribute__((at(0xc0ffee00, ".bss"))) = 0;
```

The compiler will place variable `v0` in section `.bss._AT_@0xC0FFEE00`.

6.4.2 Entry point and initialization

The entry point and initialization routines are defined in `crt1.o` of the C library. Passing `-fuse-baremetal-crt` to Clang will link the object into image; see -fuse-baremetal-crt.

6.4.2.1 Entry point

The entry point is `__main(`. It first performs initialization and eventually calls user-defined `main()`.

6.4.2.2 Initialize heap and stack

If dynamic memory allocation routines such as `malloc()`/`calloc()` are used, the initial heap starting and ending address must be set up. The C library uses two variables, `__heap_base` and `__heap_limit`, for heap base and heap end, respectively. To set the initial SP to some specific value, either assign the initial stack address to `__initial_sp` or change the SP register directly.

Heap space can also be defined in the linker script by directly assigning addresses to `__libc_heap_start` and `__libc_heap_end`. The values are used only if both `__heap_base` and `__heap_limit` are not defined.

There are two ways to set the variables:

- Assign the address to the variables directly before calling `__main`. For example:

```
extern uintptr_t __heap_base, __heap_limit, __initial_sp;
void boot() {
  __heap_base = 0x1000;
  __heap_limit = __heap_base + heap_size;
  __initial_sp = 0x8000;
  __main();
}
```

  The default `__user_setup_stackheap()` defined in `crt1.o` will initialize the SP register to the value defined by `__initial_sp`.

- Re-implement `__user_setup_stackheap` and set the variable there. For example:

```
.global __user_setup_stackheap
.type __user_setup_stackheap, %function
__user_setup_stackheap:
ldr r0,  =__heap_base
ldr r1, #0x1000
str r1, [r0]
ldr r0, =__heap_limit
ldr r1, #0x2000
str r1, [r0]
mov sp, #4000
bx lr
```

6.4.3 Override library functions

It is possible to redefine functions of the C library. For example, you can provide your own memory allocation routines to replace the standard `malloc()`. However, other functions of the C library such as `snprintf()` might call `malloc()` as well. To re-route all calls to a user-defined version, two methods are available:

- Implement the functionality with the same function name. For example:

```
mymalloc.c:
void * malloc() { /* user-defined implementation */ }
```

  NOTE: During linking, the object file that defines the function (such as `mymalloc.o`) must appear after all other user object files that use the function, but before `-lc`.

- Implement the functionality with the `__wrap_` prefix. For example:

```
void * __wrap_malloc() { /* user-defined implementation*/
```

Then during link step, pass the linker flag `--wrap <function>` (such as `--wrap malloc`). Object files can be listed in any order on the command line.

6.4.4 I/O functions in C library

High-level APIs (such as `printf`, `putc`) will eventually call low-level input/output functions. The default implementation is for semi-hosting. You can re-implement those low-level functions to re-target the input/output.

- Read from a file or STDIO:

```
size_t __stdio_read(FILE *f, unsigned char *buf, size_t len)
```

- Write to a file or STDOUT/STDERR:

```
size_t __stdio_write(FILE *f, const unsigned char *buf, size_t len)
```

- File seek:

```
off_t __stdio_seek(FILE *f, off_t off, int whence)
```

- File open:

```
FILE *fopen(const char *restrict filename, const char *restrict
mode)
```

- File close:

```
int __fclose_ca(FILE *f)
```

6.4.5 Semihosting support

The C library implements the following semihosting sys calls that enable code running on an Arm target to use the I/O facilities on a host computer that is running a compliant debugger:

```
SYS_OPEN (0x01)
SYS_CLOSE (0x02)
SYS_WRITE (0x05)
SYS_READ (0x06)
SYS_SEEK (0x0A)
SYS_FLEN (0x0C)
```

The following C functions are supported for semihosting:

```
fopen
fclose
fseek
fread
fwrite
printf/fprintf
putc/fputc
```

6.5 Examples

This section provides several examples. The first example shows how to set up an interrupt vector table. An interrupt vector table (also called exception vectors) contains the addresses of all exception handlers. The `Reset` handler is usually the entry point of the bare metal image. For Cortex-M3, the first entry of the interrupt vector table is the initial stack pointer.

The example that shows how to set up stack space has a corresponding linker script example.

6.5.1 Set up interrupt vector table

In `vector_table.c`:

```
typedef void (*FPtr)(void) __attribute__((interrupt("IRQ")));
#define STACK_START_ADDR 0x0E00
#define STACK_SIZE 0x400
static const FPtr vector_table[] __attribute__((at(0x0), used)) =
  { (FPtr)Stack_Start_Addr /* the initial SP value */,
     __main, /* handler for RESET*/
     abort_isr,    HardFault_Handler,
    ...
  }
```

- `at(0x0)` instructs the tool chain to place `vector_table` at address 0
- `used` attribute informs the compiler to keep the static variable even if it is unreferenced in the source code

6.5.2 Set up stack space

In `main.c`:

```
int main() {
   /* The loader or the image itself should zero-initialize BSS sections
   */
   extern char DRAM_BSS_START, DRAM_BSS_SIZE;
   memset((void*)&DRAM_BSS_START, 0, (int)(&DRAM_BSS_SIZE));

   start_my_image();
   }

   int stack_base_address = STACK_START_ADDR;
   // Re-implement __user_setup_stackheap to set the initial value of SP.
   // The "naked" attribute prevents the compiler to generate code for
   prologue/epilogue that could
   // clobber the value of SP.
   __attribute__((naked)) void __user_setup_stackheap() {
     asm("ldr r0, =stack_base_address");
     asm("ldr sp, [r0]");
     asm("bx lr");
   }
```

Following is the corresponding linker script:

```
ENTRY(__main)  /* Specify entry point as __main */
PHDRS {
 /* Here we create two load regions / segments */
  CODE_LOAD PT_LOAD;
  DATA_LOAD PT_LOAD;
}
SECTIONS {
  /* Here we place all code and RO data into section "EXEC" and assign
it to CODE_LOAD segment */
  EXEC 0x0 : {
    *.o (.text*)
    foo\bar\init.o (MyCode) /* Pleasese back-slash as path separator
for both Windows and Linux build environment */
    *.o(.rodata*)
  } : CODE_LOAD

/* Here we place all data into to DRAM_DATA section and assign it to
DATA_LOAD segment */
  DRAM_DATA : {
    *(.data*)
  } : DATA_LOAD

/* Here we place all BSS data into DRAM_BSS section and it will go to
the same segment as previous section. Usually, BSS sections should be
placed at end of a segment */
  DRAM_BSS : {
    DRAM_BSS_START = .;
    *.o (.bss*)
    }
    DRAM_BSS_SIZE = SIZEOF(DRAM_BSS);

/* Here we create the space to be used for stack. The section is
defined from a lower address */
  STACK (STACK_START_ADDR - STACK_SIZE) : {    . +=  STACK_SIZE;   }
  /DISCARD/ : {     *(.ARM.exidx*)  }  /* Unneeded sections go here */
```

6.6 Port from Arm Compiler toolchain

The Snapdragon LLVM toolchain provides features and capabilities comparable to the Arm Compiler toolchain. Following are the corresponding components of both toolchains.

Table 6-2 Toolchain components mapping

Components	Arm Compiler toolchain	Snapdragon LLVM toolchain	Notes
C and C++ compiler	armcc (version 4 and 5) armclang (version 6)	clang	
Arm and Thumb assembler	armasm	clang	Clang only supports GNU assembly language
Arm librarian	armar	arm-ar	
Arm linker	armlink	clang or ld.qcld	Clang can be used as driver for linking
Arm image conversion utility	fromelf	llvm-elf-to-hex.py	
Supporting libraries	Built into toolchain	libc, libc++, libc++abi, libunwid, libclang_rt.builtins	

There are some differences in various language syntax between Arm Compiler toolchains and Snapdragon LLVM toolchain. Following are some of the differences.

6.6.1 Assembly files

The syntax of most instructions is the same for armasm and clang, but directives are different. Clang supports GNU assembly language syntax and requires UAL (Unified Assembly Language) syntax as dictated by the Arm architecture specification. The following table lists the mapping of some of the directives.

Table 6-3 Mapping directives

armasm supported syntax	clang supported syntax
`@ (comment)`	`;`
`IMPORT`	`.extern`
`EXPORT`	`.global`
`PRESERVE8`	`.eabi_attribute Tag_ABI_align8_preserved, 1`
`AREA <name>, <attribute>`	`section <name>, <flags>`
`MACRO`	`.macro`
`MEND`	`.endm`
`xxx`	`.Lxxx`
`<name> FUNCTION`	`.type <name>, %function`

Table 6-3 Mapping directives (cont.)

armasm supported syntax	clang supported syntax
<name>:	
ENDFUNCTION	(not needed)
SPACE	.space
DCD	.word
<sym> EQU *<val>*	.set *<sym>*, *<val>*
CODE32	.code 32
LTORG	.ltorg
ALIGN	.balign
INCLUDE	.include
:OR:	|
:AND:	&

6.6.2 C/C++ source code

No change is required for C/C++ code that conforms to the C/C++ standards. Compiler-specific syntax like inline assembly, attributes, intrinsics, and built-in functions must be ported.

Table 6-4 Commonly used attributes, intrinsics, and built-in functions

armcc	clang
__weak	__attribute__((weak))
__irq	__attribute__((interrupt("IRQ")))
__inline__	inline
__CLZ	__builtin_clz
__dsb	__builtin_arm_dsb
__dmb	__builtin_arm_dmb
__return_address	__builtin_return_address(0)

6.6.3 Mapping of commonly used compiler flags

Table 6-5 Commonly used compiler flags

armcc	clang
--protect_stack	-fstack-protector
--loose_implicit_case	-Wno-int-conversion
--cpu *<CPU>*	-mcpu=*<CPU>*
--C99	-std=c99
--diag_error=warning	-Werror

Table 6-5 Commonly used compiler flags (cont.)

armcc	clang
`--debug`	`-g`
`--dwarf3`	`-gdwarf-3`

6.6.4 Linker script

The syntax of the linker script (scatter loading files) is very different. This is no simple mapping between the two. For details, see the *Qualcomm Snapdragon LLVM ARM Linker User Guide* (80-VB419-102).

6.7 Code coverage for bare metal environments

The source-based code coverage helps you to understand which part of the code is used during execution. This can help to identify if a test case effectively covers the code, discover potential dead code, and so on

6.7.1 Requirements for bare metal images

To retrieve the coverage data, semihosting is required. The resulting raw data is stored on host via semihosting.

Extra image size and run-time memory is expected to accommodate the instrumentation code and data.

6.7.2 Code coverage workflow

1. Build the source code with instrumentation.
2. Link the objects against the run-time library.
3. Run the code for one or more times. For each run, export the coverage data.
4. Combine the coverage data from multiple runs, and generate the report.

6.7.3 Build a program with code coverage support

6.7.3.1 Compile

Compile each object with the following flags:

```
-fcoverage-mapping
```

For bare metal images, you must explicitly dump the result data at the intended stop point. This is because bare metal images do not call the exit() function, so the run-time library will not capture the exiting point.

```
#include "qc_coverage.h"

void my_job_done() {
   __qc_llvm_profile_write_file("my_code_coverage.dat");
}
```

6.7.3.2 Link

If clang is invoked to drive the linking, and assuming `-fuse-baremetal-sysroot` is passed and the `-target` flag is set properly, add `-fcoverage-mapping` in the linking command. The linker automatically searches for the required library.

If linking is done by invoking the linker directly, you must explicitly specify the run-time library for profiling, such as `-lclang_rt.profile-armv5`.

For bare metal images, the linker script must be updated to place some specific sections:

1. Place the following read only sections somewhere in memory:

   ```
   __llvm_prf_names : { KEEP(*(__llvm_prf_names)) }
   ```

2. Place the following data sections somewhere in memory:

   ```
   __llvm_prf_data : { KEEP(*(__llvm_prf_data)) }
   __llvm_prf_cnts : { KEEP(*(__llvm_prf_cnts)) }
   ```

3. The following section is not required to be in memory (like debug info):

   ```
   __llvm_covmap : { KEEP(*(__llvm_covmap)) }
   ```

6.7.3.3 Example

```
PHDRS {
  CORE_REGION PT_LOAD;
}

SECTIONS {
  CORE_RO : {
    *\init*.o (entry)
    * (.text*)
    * (.rodata*)
  } : CORE_REGION

  __llvm_prf_names : { KEEP(*(__llvm_prf_names)) }
```

```
  CORE_RW : ALIGN(0x100000) {
    * (.data*)
    * (stacks)
    }
  __llvm_prf_data : { KEEP(*(__llvm_prf_data)) }
  __llvm_prf_cnts : { KEEP(*(__llvm_prf_cnts)) }

  CORE_ZI: {
    * (.bss*)
  }

  __llvm_covmap : { KEEP(*(__llvm_covmap)) }
   /DISCARD/ : {
     *(.ARM.exidx*)
   }
}
```

6.7.4 Export code coverage data

The code can be run multiple times. For bare metal images, the data is stored on the host machine via semi-hosting, and the filename is specified in the source code in `__qc_llvm_profile_write_file`. The data file might need to be renamed before the next run.

6.7.5 Generate code coverage reports

You can consolidate the code coverage data from multiple runs into a single report. The report is in HTML format and is stored in a specified directory.

```
mkdir code_coverage_report
<llvm_root>/bin/llvm-profdata merge data_from_run1 data_from_run2 ...
-output=merged.data
<llvm_root>/bin/llvm-cov show -format=html -instr-profile=merged.data
-show-region-summary=false image.elf -o code_coverage_report
```

Using a browser, open the `index.html` under `code_coverage_report`.

`llvm-cov export` converts the coverage data to JSON format, which can be used for writing tools on top of the LLVM coverage infrastructure.

6.7.6 Interpret code coverage reports

The index page of the coverage report shows coverage by file.

- **Function Coverage** is the percentage of functions that were executed during the coverage runs.
- **Line coverage** is the percentage of lines.

These percentages only count code compiled into the executable, so they ignore code that is disabled using an `ifdef`. Cells that show coverage less than 80% are highlighted in red.

If you click on the link to one of the individual files, you can see on a line-by-line basis which lines are covered.

- Code that is not executed is highlighted in red.
- The number in blue on the left (after the line number) is the number of times a line was executed.
- Source code lines that are not code, like global variables or code disabled using an `ifdef`, do not show any execution count.

7 Resource analyzer

The resource analyzer is a tool that determines the stack usage (in bytes) for each function in an executable ELF file. The stack usage information is output as an SVG file that contains an annotated call graph of the executable.

NOTE: Python version 2.7 or higher is required to run the resource analyzer, which is a standalone tool and is not part of the compiler.

7.1 Usage

The following procedure explains how to use the resource analyzer.

1. Run the resource analyzer on an executable ELF file. For example:

   `llvm-arm-ra.py` *`application`*

   This command creates a DOT file named `cfg.dot`. To create the output file with a different name, use the `-c` *`filename`* option.

   The resource analyzer is assumed to be stored in *`install_dir`*`/tools/bin/`.

2. Create an SVG file:

   Use the Graphviz `dot` utility to convert the DOT file into an SVG file. For example:

   ```
   dot -Tsvg cfg.dot -o call_graph.svg
   ```

   Where:

   - `-Tsvg` specifies the input DOT file
   - `-o` specifies the name of the output SVG file

3. View the SVG file in your browser by specifying the file pathname in the browser address bar. For example:

   ```
   file://foo/bar/call_graph.svg
   ```

The SVG file displays the call graph for the executable file, annotated with the stack usage (in bytes) for each function.

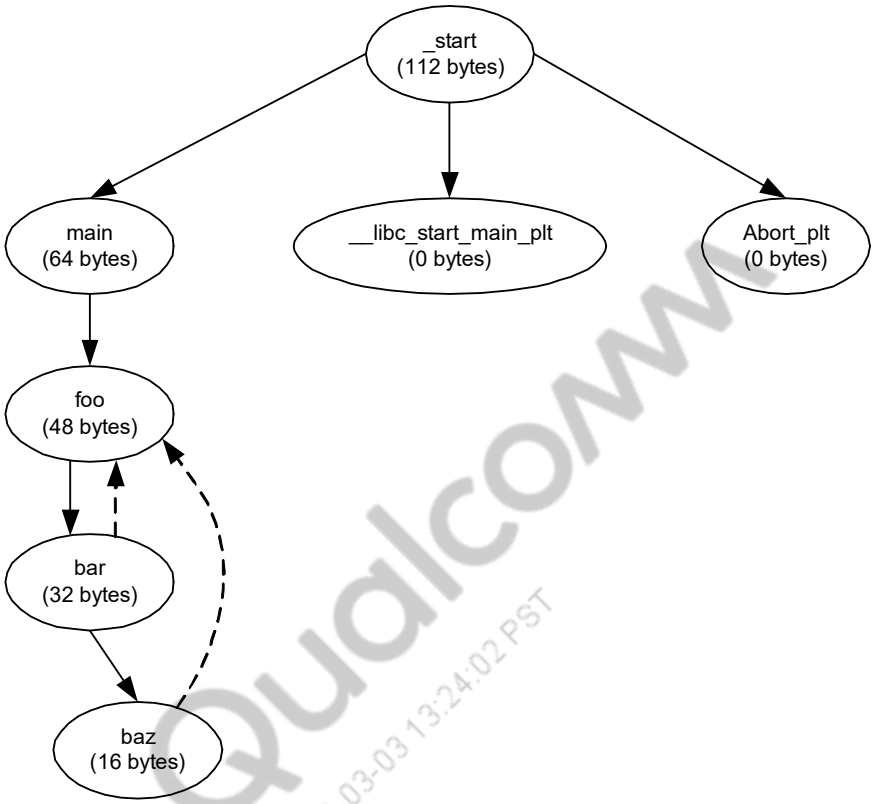


Figure 7-1 Call graph example

7.2 Options

The resource analyzer can be controlled by command line options. For example:

```
llvm-arm-ra.py application -o filename
```

Descriptions

-c *filename*
--cfg *filename*

Specify the name of the DOT file to be created. The default setting is `cfg.dot`.

-d *filename*
--objdump_tool *filename*

Specify the pathname of the `objdump` utility to be used to generate objdump data.

- For 32-bit ELF files, the default executable is `arm-linux-gnueabi-objdump`.
- For 64-bit ELF files, it is `aarch64-linux-gnu-objdump`.

The resource analyzer generates an error message if it cannot find these executables on the system PATH.

-e *filename*
--edge_file *filename*

Use an edge file to create custom edges in the generated call graph. The specified edge file is a text file that contains pairs of function names. Each line in the file has the following format:

```
f1:f2
```

The result is the addition of a custom edge (from function `f1` to function `f2`) in the call graph. However, it might add to the stack usage of `f1`, if and when applicable.

-f *filename*
--function_file *filename*

Use a function file to limit the generated call graph to the specified functions. The specified function file is a text file that contains a list of function names. Only the functions specified in this list will be included in the call graph.

-h
--help

Display resource analyzer command and option summary.

-o *filename*
--objdump_file *filename*

Specify the filename of the generated `objdump_file`.

-v
--verbose

Display detailed debug output for the resource analyzer.

7.3 Notes

Recursive functions

The resource analyzer does not report stack usage information for recursive functions.

Call graph cycles

If two functions `f1` and `f2` call each other (thus creating a cycle in the call graph), the resource analyzer removes edges from the call graph until the cycle is broken.

In the example call graph in Section 7.1, the functions `foo` and `bar` call each other. In this case, the resource analyzer keeps the edge `foo -> bar` but removes the edge `bar -> foo`, thus breaking the cycle.

NOTE: Edges removed by the resource analyzer are displayed in the call graph with dotted lines.

Missing main functions

For input ELF files (such as * . `so`) that do not have a main function, the resource analyzer creates a dummy start root node and connects this node to each function in the file that is not called by another function.

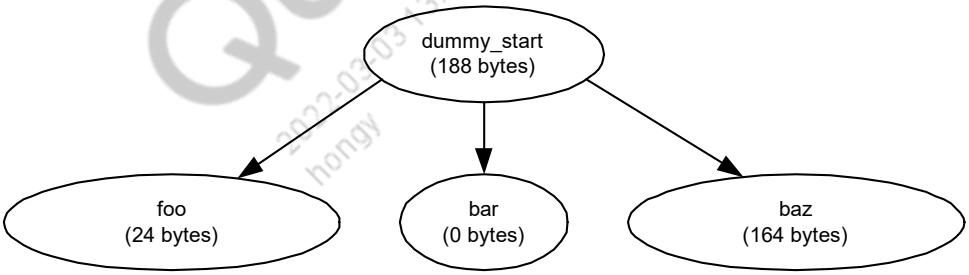


Figure 7-2 Example of a dummy root node connected to functions

dot utility

The dot utility (which is part of the Graphviz package) is required to convert the CFG file to SVG format.

On Linux, install Graphviz with the following command:

```
apt-get install gv
```

On Windows, download Graphviz from the following link:

http://www.graphviz.org/Download.php

7.4 Algorithm

The resource analyzer determines the stack usage as follows:

```
Stack size of function F = Sum (stack size of {push, sub} instructions
of F + max{stack sizes of all functions called by F})
```

push

- Stack size of push instruction = register size for architecture * number of registers pushed
- Register size = 4 bytes for 32-bit architectures or 8 bytes for 64-bit architectures
- Example:

  ```
  push {r3, lr}
  ```

- Stack size of `push` instruction = 4 bytes * 2 registers = 8 bytes for 32-bit architectures, or 8 bytes * 2 registers = 16 bytes for 64-bit architectures

sub

- Stack size of sub instruction = value of the non-negative immediate that modifies the stack pointer (`sp`)
- Example:

  ```
  sub sp, sp, #1280
  ```

- Stack size of `sub` instruction = 1280 bytes

8 Compiler security tools

The LLVM compilers support several tools and features for improving the security and reliability of program code.

8.1 Sanitizer support

Not all sanitizers are supported on all targets. Table 8-1 shows which sanitizers are supported on which targets.

Table 8-1 Sanitizer support

Sanitizer	AARCH64-Linux	AARCH64-Android	Arm-Linux	Arm-Android
Address	4	4	4	4
Data flow	4	X	X	X
Leak	4	X	X	X
Memory	4	X	X	X
Thread	4	X	X	X
Undefined behavior	4	X	4	X

8.1.1 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 that lists the source-level entities to be ignored. Then pass this file to the compiler with the `-fsanitize-blacklist` option (Section 4.3.19).

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*]

Where:

- *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 structure 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 structure type.
- *regexp* specifies a regular expression that 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 wildcards. 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 function (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
```

8.1.2 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 the executable, the sanitizer runtime library, and the LLVM Symbolizer (Section 8.8) to an Android device. The sanitizer runtime library is a shared object that 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, and so on).

Example

Choose one of the examples that are provided in the individual sanitizer sections (see Section 8.2.1).

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, and so on).

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, and so on).

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, and so on) and library (asan, msan, and so on).

Include the symbolizer in the argument string (as shown in these steps) 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"
```

8.1.3 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`, and so on) and path variable (`ASAN`, `MSAN`, and so on).

Include the symbolizer (as shown in these steps) only if the sanitizer you are using requires a symbolizer to resolve the symbol names.

8.2 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 that 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.

8.2.1 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` might 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 might be necessary to disable inlining (only 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.

8.2.2 Symbolize 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 the `$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
```

8.2.3 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, set the environment variable:

```
ASAN_OPTIONS=check_initialization_order=1.
```

Memory leak detection

For more information on memory leak detection in ASan, see Section 8.4.

8.2.4 Issue suppression

ASan generally does not produce false positives, so if you see one, look again. Most likely it is a true positive.

8.2.4.1 Suppress 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 is resolved. However, you can use the following suppression mechanism to unblock yourself and continue with testing. Use this suppression mechanism only 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
```

8.2.4.2 __has_feature(address_sanitizer)

In some cases, you might 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 that 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).

8.2.4.3 __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`.

8.2.4.4 Blacklist—Runtime suppression

ASan supports the use of sanitizer special case lists to suppress error reports in the specified source files or functions (Section 8.1.1). It also 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 structure types).

ASan defines a 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
```

8.2.5 Suppress memory leaks

If the Leak Sanitizer (Section 8.4) 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 filename, or library name in the symbolized stack trace of the leak report. For details, see Section 8.4.

8.2.6 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. Up to a 3x increase can occur.
- On 64-bit platforms, ASan maps (but does not reserve) 16+ terabytes of virtual address space. Thus, tools like `ulimit` might not work as usually expected.
- Static linking is not supported.

8.2.7 Options

When running the instrumented executable and ASan does not detect any errors, there is no output. Conversely, when you there is no output, it can mean either that no errors occurred or that the executable was not instrumented with the ASan runtime.

To verify that the executable is instrumented with ASan, use the `ASAN_OPTIONS` environment variable and `verbosity=1` flag. Doing this directs the ASan runtime to output a startup message when the executable is launched. For example (in Bash):

```
$ ASAN_OPTIONS=verbosity=1 ./myExe
```

If no output is generated while verbose mode is enabled, this implies the executable was not instrumented with ASan. Check that the `-fsanitize=address` option 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 or 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
```

Descriptions

verbosity

Be more verbose (mostly for testing the tool itself). (Default = 0)

malloc_context_size

Number of frames in malloc/free stack traces (0-256). (Default = 30)

redzone

Size of minimal redzone. (Default = 16)

log_path

Path to log files. If specified as `log_path=PATH`, every process will write error reports to `PATH.PID`. (Default = stderr)

sleep_before_dying

Sleep for the specified number of seconds before exiting the process on failure. (Default = 0)

quarantine_size

Size of quarantine (in bytes) for finding use-after-free errors. Lower values save memory but increase false negatives rate. (Default = 256 MB)

exitcode

Call `_exit(exitcode)` on error. (Default = 1)

abort_on_error

If set to 1, on error call `abort()` instead of `_exit(exitcode)`. (Default = 0)

strict_memcmp

If set to 1, treat `memcmp("foo", "bar", 100)` as a bug. (Default = 1)

alloc_dealloc_mismatch

If set to 1, check for mismatches between `malloc()/new/new` and `free()/delete/delete`. (Default = 1)

handle_segv

If set to 1, ASan installs its own handler for SIGSEGV. (Default = 1)

allow_user_segv_handler

If set to 1, allows you can override the SIGSEGV handler installed by ASan. (Default = 0)

check_initialization_order

If set to 1, detect existing initialization order problems. (Default = 0)

strip_path_prefix

If `strip_path_prefix=PREFIX`, remove the substring `.*PREFIX` from the reported filenames. (Default =" ")

8.2.8 Notes

The ASan runtime library does not yet demangle symbols, but the LLVM symbolizer can be used to demangle symbols (Section 8.8).

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, go to:

clang.llvm.org/docs/AddressSanitizer.html

8.3 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.

8.3.1 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 functions are defined in the `sanitizer/dfsan_interface.h` header file. For more information about each function, see the header file.

8.3.2 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 that is intended to eventually cover the `glibc` library on Linux. But it might become necessary to extend the ABI list when 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 when function is called from a library or function that cannot be instrumented.

DFSan's ABI list file uses the same format as a sanitizer special case list (Section 8.1.1). 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 because 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 that do not write to user-accessible memory). Its return value is unlabeled.
- `functional` – Like `discard`, except 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 might wrap the original function or provide its own implementation.

This category is generally used for functions that cannot be uninstrumented to write to user-accessible memory, or for functions that 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 a 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
```

8.3.3 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;
}
```

For more information on DFSan, go to:

clang.llvm.org/docs/DataFlowSanitizer.html

8.4 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 8.2) to enable both memory error and leak detection, or it can be used as a standalone tool.

NOTE: LSan adds almost no performance overhead until the very end of the process, when an extra leak detection phase is performed.

8.4.1 Usage

To use LSan, simply build the program with the Address Sanitizer (Section 8.2).

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 standalone mode, link the program with the `-fsanitize=leak` option. 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.

For more information on LSan, go to:

clang.llvm.org/docs/LeakSanitizer.html

8.5 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 that 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 can optionally display information on where the uninitialized memory was originally allocated.

8.5.1 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` might 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 might need to disable inlining (only 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
```

By default, MSan exits on the first detected error. If you find the error report hard to understand, try enabling origin tracking (Section 8.5.3).

8.5.1.1 __has_feature(memory_sanitizer)

In some cases, you might 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 that 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).

8.5.1.2 __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 might still instrument such functions.

8.5.1.3 Blacklist

MSan supports the use of sanitizer special case lists to suppress error reports in the specified source files or functions (Section 8.1.1). All *Use of uninitialized value* warnings are suppressed, and all values loaded from memory are considered fully initialized.

8.5.2 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.

8.5.3 Origin tracking

MSan can track origins of uninitialized values, similar to Valgrind's `-track-origins` option. This feature is enabled with the `-fsanitize-memory-track-origins=2` option (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 `-fsanitize-memory-track-origins=1` option enables a slightly faster mode when MSan collects only allocation points and not intermediate stores.

8.5.4 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.

NOTE: This feature is experimental.

To enable this feature at runtime, perform the following steps:

1. During compilation, specify the `-fsanitize-memory-use-after-dtor` option.
2. Before running the program, set the `MSAN_OPTIONS=poison_in_dtor=1` environment variable.

8.5.5 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 can 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`.

8.5.6 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. Thus, tools like `ulimit` might 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.

For more information on MSan, go to:

clang.llvm.org/docs/MemorySanitizer.html

8.6 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 that 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).

8.6.1 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 filenames 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)
```

8.6.1.1 __has_feature(thread_sanitizer)

In some cases, you might 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 that 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).

8.6.1.2 __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 might still instrument such functions.

8.6.1.3 Blacklist

TSan supports the use of sanitizer special case lists to suppress data race reports in the specified source files or functions (Section 8.1.1).

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.

8.6.2 Limitations

- TSan uses more real memory than a native run. At the default settings, the memory overhead is 5x plus 1 MB 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. Thus, tools like `ulimit` might 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.

For more information on TSan, go to:

clang.llvm.org/docs/ThreadSanitizer.html

8.7 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 that result in overflow

UBSan is a runtime tool that requires compile-time instrumentation of the code. It includes an optional runtime library that 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.

8.7.1 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.

8.7.2 Checks and option values

UBSan performs checks that are individually controlled by option values passed to the `-fsanitize=`*`event`* option (Section 4.3.19).

`alignment`

Use of a misaligned pointer or creation of a misaligned reference.

`bool`

Load of a boolean value that 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 that is not in the range of representable values for that enumerated type.

`float-cast-overflow`

Conversion to, from, or between floating-point types that 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`

Pass a NULL pointer as a function parameter that is declared to never be NULL.

null

Use a NULL pointer or creation of a NULL reference.

object-size

Attempt to use bytes that 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

Return a NULL pointer from a function that is declared to never return NULL.

shift

Shift operators where the amount shifted is greater or equal to the promoted bitwidth 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++.

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 in this section, other than `unsigned-integer-overflow`.

integer

Checks for undefined or suspicious integer behavior (such as an unsigned integer overflow).

8.7.3 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 the program with the environment variable `UBSAN_OPTIONS=print_stacktrace=1`.
3. Ensure that the `llvm-symbolizer` binary is in `PATH`.

8.7.4 Issue suppression

UBSan generally does not produce false positives, so if you see one, look again. Most likely it is a true positive.

8.7.4.1 __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 the function deliberately contains possible signed integer overflow, you can use the following:

```
__attribute__((no_sanitize("signed-integer-overflow"))).
```

8.7.4.2 Blacklist

UBSan supports the use of sanitizer special case lists to suppress error reports in the specified source files or functions (Section 8.1.1).

8.7.4.3 Runtime suppression

Sometimes you can suppress UBSan error reports for specific files, functions, or libraries without recompiling the code. You must pass a path to suppression file in a `UBSAN_OPTIONS` environment variable.

```
UBSAN_OPTIONS=suppressions=MyUBSan.supp
```

You must specify a check (Section 8.2.3) 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 the binary must have enough debug information or a symbol table so the runtime code can figure out the source file or function name to match against the suppression.
- Only recoverable checks can be suppressed.
- For the previous example, you can also pass `-fsanitize-recover=signed-integer-overflow,alignment,vptr`, although most of the UBSan checks are recoverable by default.
- Check groups (such as `undefined`) cannot be used in suppressions files. Only fine-grained checks are supported.

8.7.5 Notes

In the C language specification, *undefined behavior* is the result of performing certain erroneous operations that are not flagged with an error. 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, go to:

blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

For more information on UBSan, go to:

clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

8.8 LLVM Symbolizer

The LLVM compiler release includes a tool named *LLVM Symbolizer*, which can be used to convert program addresses into source code locations.

The Symbolizer is a command-line tool—it reads object filenames and addresses from the standard input, and writes the corresponding source code locations to the standard output.

If an object filename is directly specified as a command-line argument, the Symbolizer treats it as the name of the input object file, and reads only addresses from the standard input.

NOTE: To perform its conversion, the Symbolizer uses the symbol tables and debug information sections that are stored in the object files.

To start the Symbolizer from a command line, enter the following:

`llvm-symbolizer` *`options...`*

Command options are used to control the symbolizer (Section 8.8.2).

The Symbolizer normally returns 0 as a program return code. Any other code values indicate an internal program error.

The Symbolizer is used with ASan (Section 8.2), MSan (Section 8.5), and UBSan (Section 8.7).

8.8.1 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
```

8.8.2 Options

The Symbolizer is controlled by command-line options.

-default-arch *arch_name*

If a binary contains object files for multiple architectures (for example, 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 setting is an empty string.

The architecture can alternatively be specified by passing the string *binary_name:arch_name* as part of the input (Section 8.8.1).

NOTE: If an architecture is not specified in either way, addresses will not be symbolized.

-demangle

Print demangled function names. The default setting is `enabled`.

-functions=(none|short|linkage)

Specify how function names are printed.

`none`

Omit the function name.

`short`

Print the short function name.

`linkage`

Print the full linkage name (Default).

The default setting is `enabled`.

-inlining

If a source code location is in an inlined function, prints all the inlined frames. The default setting is `enabled`.

-obj

Path to the object file to be symbolized.

-use-symbol-table

Favor function names stored in the symbol table over function names in debug information sections.

8.9 Control flow integrity

Control flow integrity (CFI) is a compiler feature that aborts the program upon detecting certain forms of undefined behavior that might 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
```

8.9.1 Configuration

CFI must be configured to handle control-flow violations; otherwise, the violations are ignored by default. It can also be configured to generate different types of code instrumentation. Different types of programs can 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, enter:

llc *options...*

Command options are used to control the static compiler (Section 8.9.3).

8.9.2 Usage

1. Compile the program files, enabling CFI and generating the 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 the bitcode files:

```
llvm-link -o prog.ll foo.ll bar.ll
```

3. Static-compile the bitcode files, configuring CFI and generating the relocatable object file:

```
llc -cfi-enforcing -cfi-type=sub -jump-table-type=simplified
-filetype=obj -o prog.o prog.ll
```

4. Link the relocatable object file into the executable binary:

   ```
   clang -o prog prog.o
   ```

5. Run the CFI-instrumented executable binary:

   ```
   ./prog
   ```

8.9.3 Options

The LLVM static compiler is controlled by command-line options.

`-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, and 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: We recommend using `simplified` because it offers the best balance between security and robustness.

8.9.4 Handler functions

If a user-defined handler function is specified (with the `-cfi-func-name` option), the function must accept two `char*` parameters:

- The first parameter is a C string that 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.

To be specified with `-cfi-func-name`, the handler function must be defined as a linker symbol.

8.9.5 Notes

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, go to:

www.pcc.me.uk/~peter/acad/usenix14.pdf

For more information on CFI, go to:

clang.llvm.org/docs/ControlFlowIntegrity.html

For more information on the LLVM static compiler, go to:

llvm.org/docs/CommandGuide/llc.html

8.10 Static program analysis

The LLVM compiler release includes the following tools for performing static analysis on a program:

- Static analyzer

  A tool that analyzes source code to find potential bugs in C and C++ programs. It can be used to analyze individual files or entire programs.

- Postprocessor

  A tool that creates a summary of the reports that are generated by performing static analysis while compiling a program.

- Scan-build

  An additional tool for compiling and statically analyzing a program. It can be used with a make-based build system.

8.10.1 Static analyzer

The static analyzer is a source code analysis tool that is integrated into the LLVM compiler. It analyzes a program for 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* that are organized into the `alpha`, `core`, `cplusplus`, `debug`, and `security` categories
- 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

8.10.1.1 Analyze programs

To use the static analyzer on an entire program, invoke the LLVM compiler on the program using the `--compile-and-analyze` option (Section 4.3.29). For example:

```
clang --compile-and-analyze dir input_files...
```

- `--compile-and-analyze <dir>` 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.

We recommend statically analyzing an entire program at once (as opposed to selected source files) 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 generates all the HTML report files in the specified directory. The filenames generated for a report are based on hashing functions, so the report files are not overwritten.

8.10.1.2 Analyze programs using default flags

To use the static analyzer on specific program source files, invoke the LLVM compiler on the files using the static analyzer options (Section 4.3.29). For example:

```
clang --analyze -Xclang --analyzer-output -Xclang html
-o dir files
```

Where:

- `--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 `--analyzer-output html` option to the compiler. For details, see Section 4.3.2.

- `-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'
```

Each potential bug flagged in a report includes the path (control and data) required for locating the bug in the program.

NOTE: To convert static analyzer warnings to errors, use the `--analyzer-Werror` option.

8.10.2 Analyze programs with priority modes

In addition to the default `--compile-and-analyze` flag, the static analyzer also offers a list of specific directives for more targeted analysis.

- High Priority mode

  This mode turns on the highest priority checkers that catch critical issues and have low false positive rates.

  This mode can be enabled by the `--compile-and-analyze-high` flag.

- Medium Priority mode

  This mode is slightly more inclusive compared to the high priority mode, but it still avoids some noisy checkers that are enabled by default.

  This mode can be enabled by the `--compile-and-analyze-medium` flag.

- KW mode

  This mode reduces issues that are typically caught by commercial static analysis tools. It is especially helpful if you want to catch those issues early in the development process.

  Enable this mode with the `--compile-and-analyze-kw` flag.

8.10.2.1 Manage checkers

The static analyzer supports more than 100 individual checkers that 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 `-analyzer-checker` option (Section 4.3.29). This option specifies the checker to be enabled (in the following example, `NewDelete`).

```
clang++ --compile-and-analyze <dir> -Xclang -analyzer-
checker=NewDelete test.cpp
```

To disable individual checkers, invoke the static analyzer using the `-analyzer-disable-checker` option (Section 4.3.29). For example:

```
clang++ --compile-and-analyze <dir> -Xclang -analyzer-disable-
checker= deadcode.deadstore test.cpp
```

NOTE: `-Xclang` must be used to pass the `-analyzer-output`, `-analyzer-checker`, and `-analyzer-disable-checker` options to the compiler. For details, see Section 4.3.29.

To list all the supported checkers, use the following command:

```
clang -cc1 -analyzer-checker-help
```

To list only the default checkers, use the `-v` option while running the analyzer on a test file (Section 4.3.1). For example:

```
clang++ -v --compile-and-analyze --analyzer-perf test.cpp
```

8.10.2.1.1 Packages

The individual checkers are organized into the following categories:

- alpha
- core
- cplusplus (only for analyzing C++ programs)
- debug
- security
- unix
- optin

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 --compile-and-analyze <dir> -Xclang -analyzer-checker=alpha
test.cpp
```

8.10.2.1.2 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 --compile-and-analyze <dir> -Xclang -analyzer-
checker=alpha,core test.cpp
```

8.10.3 Cross-file analysis

A limitation of the clang static analyzer is its inability to analyze code that is called across file boundaries. This is due to the nature of the compiler, which compiles files individually. This feature enables the clang static analyzer to perform its analysis on code that is called across file boundaries.

The cross-translation unit (CTU) feature allows the analysis of called functions even if the definition of the function is external to the currently analyzed file. This feature allows detection of bugs in library functions stemming from incorrect usage. If an external function is invoked, this feature also allows for more precise analysis of the caller in general.

Enabling CTU analysis is a three-step process:

1. Generate the `compile_commands.json` file.

   The makefile is read by the intercept-build script. This script is responsible for creating a `compile_commands.json` file that contains a list of every build command that is run from this build system.

   ```
   $share/scan-build-py/bin/intercept-build make
   ```

2. Generate the AST database.

   Generate the ASTs of every function in the build tree by issuing the following command:

   ```
   $share/scan-build-py/bin/analyze-build --cdb <location of the
   compile_commands json file> -o
   <location of the intended report directory> --ctu-collect-only
   --ctudir
   <location where all the AST's are to be stored>
   ```

3. Run the cross-file analysis.

   Kick off the static analyzer in the usual way (with the `--compile-and-analyze` flag) but with following added flags:

   ```
   -Xclang -analyzer-checker=debug.ExprInspection -Xclang -analyzer-
   config -Xclang -ctu-dir=<location of the intended report directory>
   ```

8.10.3.1 Handle false positives

While checking a program for potential bugs, the static analyzer might 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 8.10.2.1). 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:

- Compile-time blacklist file
- Special comment
- Preprocessor symbol
- Function attribute
- Postprocess blacklist file

We do not recommend using comments or symbols to handle false positives because they make the code inaccessible to the analyzer. Instead, report any false positives so the existing checkers can be improved to eliminate them. For more information on false positives, go to:

clang-analyzer.llvm.org/faq.html

8.10.3.1.1 Compile-time blacklist file

To silence warnings before they are reported by the analyzer, use the external file-based suppression mechanism. Enable this mechanism by adding the compiler flag, `-analyzer-suppression-file<file location>`.

Suppression data in this text file must be entered line-by-line with the following format:

```
<filename.extension<function identifier> <checker name>
```

The analyzer reads the contents of this file during the analysis and does not report any warnings that are mentioned in it.

8.10.3.1.2 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 8.10.2.1) 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]
```

8.10.3.1.3 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 is not required 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__
```

8.10.3.1.4 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);
}
```

8.10.3.1.5 Postprocess blacklist file

The blacklist file consists of row-wise indications of warnings that must be silenced. It uniquely identifies each warning by filename, function name, and bug description. Place this file in a network location that is accessible to all developer machines using the analyzer.

During postprocessing, identify the blacklist file through the `--blacklist-file <file>` flag. The `post-process` script does not report the bugs marked in this file. Besides silencing warnings across developer machines, this mechanism also allows you to silence warnings across build variants.

The blacklist file contains row-by-row warnings in the following format:

```
<filename.cpp> <function identifier/Global> <full warning description>
```

For example:

```
test.cpp foo Address of stack memory associated with local variable
buff returned to caller [core.StackAddressEscape]
```

8.10.3.2 Create whitelist directories

Often, unwieldy build systems make it impossible to turn on the static analyzer for only some subdirectories. For example, given the following hierarchy, assume you want to see results for only the `Qcomm_code` directories and not the `Open_source` directory:

ANDROID

Open_source Qcomm_hardware_code Qcomm_software_code

The build system employed here allows only `cflags` to be set at a global level. In this case, create a whitelist file containing row-wise entries of the folders to be scanned. For example:

```
ANDROID/Qcomm_hardware_code
ANDROID/Qcomm_software_code
```

The `post-process` script must be notified of the whitelist file through the flag, `--whitelist-file <whitelist text file>`. The script then identifies the row-wise entries and checks them against any static analysis warnings found. If none of the entries are a substring of the file location of a specific bug, that bug is silenced. Thus, having only

`Qcomm_hardware_code` and `Qcomm_software_code` in the whitelist file will suffice because they will both be part of the location of the warnings you are interested in.

8.10.3.3 Treat warnings as errors

Static analysis warnings can be treated as errors by appending the `--analyzer-Werror` flag to the build flags. For example:

```
clang --compile-and-analyze <dir> -Xclang --analyzer-Werror test.c
```

8.10.3.4 Checker categorization by priority

Because checkers fall under two major priority categories (high and medium), the following flags are available in addition to the regular `--compile-and-analyze` flag:

`--compile-and-analyze-high`

The high flag allows only a small subset of checkers to be turned on. These checkers are security critical and have an extremely low false positive rate. Teams adopting this tool should start with the high flag.

`--compile-and-analyze-medium`

The medium flag is slightly more permissive than the high flag. The medium flag contains a few more checkers that are deemed security critical but have a slightly higher false positive rate.

8.10.3.5 Performance mode

The static analyzer can be run in Performance mode. This mode prevents the creation of all the raw HTML files, and instead it displays a summary of the warnings in the console output.

To enable this mode, append the `--analyzer-perf` flag to the `--compile-and-analyze` flag in place of the `<dir>` option. For example:

```
clang --compile-and-analyze --analyzer-perf test.c
clang --compile-and-analyze-high --analyzer-perf test.c
```

8.10.3.6 YAML configuration file

The static analyzer can also be configured using a single configuration file (`--qc-config-file`) that combines all previously mentioned options in a single location. For example:

```
clang --qc-config-file qc_sa_config.txt test.c
```

The configuration file uses YAML format and is subject to all syntax restrictions. Following is a sample configuration file with embedded comments.

```
##########################################
# Static Analyzer configuration file     #
##########################################

##########################################
# Global SA options                      #
##########################################
---
entry-type:      global_settings
# Verbosity suppression level. Most useful is "high" i.e. "high suppression" == fewer false
positives. The "high" setting here is equivalent to the command line compile-and-analyze-
high flag.
# Available options: zero|default|high|medium|low
verbosity:       high
# Location for individual reports. Absolute or relative path.
analyzer-output-dir: sa_report_dir
# Format for output reports. Most useful html.
analyzer-output-format: html
# Same as -analyzer-config stable-report-filename=true reduces the number of duplicate
reports
stable-report-filename: true
# Do not output duplicate report for different path to the same location.
# The same header file can be included from multiple .c/cpp locations - report issues only
once.
no-duplicate-reports: true

##########################################
# Per-checker individual options         #
# It is applied _after_ the GLOBAL settings #
##########################################
---
entry-type:      checker_config
config:
  - package:         core
    # "default" - use verbosity setting from global_settings
    # "on" - enable checker
    # "off" - disable checker
    checkers:
      # Check for dereferences of null pointers
      - checker-name:    NullDereference
        state:           off
      # Check for logical errors for function calls and Objective-C message expressions
(e.g., uninitialized arguments, null function pointers)
      - checker-name:    CallAndMessage
        state:           off
  - package:         alpha.core
    checkers:
     # Check when casting a malloc'ed type T, whether the size is a multiple of the size of T
      - checker-name:    CastSize
        state:           default
```

```
# Here follows the full list of all available packages and checkers with individual settings
for each.

##########################################
# Individual files to skip during SA     #
##########################################
---
entry-type:      suppression_list
# Blacklist file. See -analyzer-suppression-file option for details
analyzer-suppression-file: suppression_file
# Per path report suppression. If any part of the file path matches this list - no report is
generated. Currently there is no wildcard support.
suppress-path:
  - modem_proc/audio_avs/main/voice/algos/mmecns/mmecns_lib/src
  - modem_proc/audio_avs
...
```

8.10.4 Postprocessor

The postprocessor is a report generator that is implemented as a standalone script. This `post-process` script creates a summary of the report that is generated by using the `-compile-and-analyze` option (Section 8.10.1). To invoke the postprocessor, enter the following command:

`post-process --report-dir` *dir*

The postprocessor reads all the files from the directory specified by the `--report-dir` option and writes in the same directory a summary report file named `index.html`. The report title can be specified with the `--html-title` option. For more information on the postprocessor, enter the **`post-process --help`** command.

The `post-process` script is stored in the `$INSTALL_PREFIX/bin` directory.

NOTE: In some cases the static analyzer might generate multiple report files for the same bug. The postprocessor cleans up after multiple report files. For this reason, run it regularly to keep the report directory clean.

8.10.5 Scan-build

`Scan-build` is a standalone tool for compiling and statically analyzing a program. You can use it with a make-based build system (instead, however, we recommend using the LLVM static analyzer whenever possible; see Section 8.10.1).

The `scan-build` tool enables you 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
```

The `scan-build` script is stored in the directory, `$INSTALL_PREFIX/bin`.

For more information, enter the command, **`scan-build --help`**.

9 Port code from GCC

This chapter describes issues commonly encountered while porting to LLVM an application that was previously built only with GCC.

NOTE: For more information on GCC compatibility, see Chapter 11.

9.1 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 4.3.5.

9.2 Errors and warnings

LLVM enforces strict conformance to the C11 language standard. As a result, you might encounter new errors and warnings when compiling GCC code.

To handle these messages when porting to LLVM, consider the following steps:

1. Remove the `-Werror` command option if it is being used (because it converts all warnings into errors).
2. Update the code to eliminate the remaining errors and warnings.

9.3 Function declarations

LLVM enforces the C11 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 is flagged as 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 `-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, go to:

clang.llvm.org/compatibility.html#inline

For a discussion of different inlining approaches, go to:

http://www.greenend.org.uk/rjk/tech/inline.html

9.4 Casting to incompatible types

LLVM enforces the C11 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 might 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.

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 whether they are signed or unsigned (for example, `int` versus `unsigned int`) can be aliased.

To satisfy strict aliasing when porting to LLVM, consider the following steps:

1. Use `-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.

9.5 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
}
```

9.6 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
```

9.7 Inline versus extern inline

LLVM conforms to the C11 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 C11 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 this example is incomplete, because if `add()` is not inlined (for example, when compiling without optimization), `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. The two definitions *must* be equivalent.
- Compile with the GNU89 dialect by adding `-std=gnu89` to the set of LLVM options. We do not recommend this approach.

10 Coding practices

This chapter describes recommended coding practices when using the LLVM compilers. These practices typically result in the compiler generating more optimized code.

10.1 Use int types for loop counters

We strongly recommend using an `int` type for loop counters, which results in the compiler generating more efficient code. If the code uses a non-`int` type, the compiler must insert zero and sign-extensions to abide by C rules. For example, we do not recommend the following code:

```
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;
}
```

10.2 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 might generate incorrect code.

10.3 Do not pass or return structures by value

We strongly recommend that structures are passed to (and returned from) functions by reference and not by value.

If a structure is passed to a function by value, the compiler must generate code that makes a copy of the structure during application runtime. This can be extremely inefficient, and will reduce the performance of the compiled code. For this reason, we recommend that structures be passed by pointer.

For example, 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);
}
```

10.4 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, we recommend using a compiler intrinsic instead of inline assembly.

Intrinsics are easy to insert in a C file, and they are portable across processor versions. If intrinsics are insufficient, add a new function written in assembly. The assembly function should be called from C code.

11 Language compatibility

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 it might 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.

11.1 C compatibility

This section describes common compatibility and portability issues with LLVM C.

11.1.1 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, gnu11, 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:

- The c* modes define `__STRICT_ANSI__`.
- Target-specific defines not prefixed by underscores (such as `linux`) are defined in gnu* modes.
- The default setting for trigraphs in gnu* modes is OFF. Trigraphs can be enabled by the `-trigraphs` option.
- The parser recognizes `asm` and `typeof` as keywords in the gnu* modes. The `__asm__` and `__typeof__` variants are recognized in all modes.
- Arrays that are VLAs according to the standard, but which can be constant-folded by the compiler front end, are treated as fixed size arrays.

  For example, `int X[(1, 2)];` is technically a VLA. The 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 default setting for *99 modes is to implement `inline` as specified in C11, while the *89 modes implement the GNU version.

  The `__gnu_inline__` attribute can be used to override the default for individual functions.
- Digraphs are not recognized in c89 mode.
- The scope of names defined in a `for`, `if`, `switch`, `while`, or `do` statement is different. For 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 that 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.

11.1.2 GCC extensions not yet implemented

LLVM tries to be compatible with GCC as much as possible, but the following GCC extensions are not yet implemented in LLVM:

- `#pragma weak`

  Will probably be implemented at some point in the future, at least partially.
- Decimal floating (`_Decimal32`, and so on) and fixed-point types (`_Fract`, and so on)

  No one has expressed interest in this extension yet; currently, it is not clear when they will be implemented.
- Nested functions

  A complex feature that is infrequently used, so it is unlikely to be implemented soon.
- Global register variables

  Requires additional LLVM back-end support, so this extension is unlikely to be implemented soon.
- Static initialization of flexible array members

  A rarely used extension, but it could be implemented pending user demand.

- `__builtin_va_arg_pack` and `__builtin_va_arg_pack_len`

  Rarely used, but they might be implemented in potentially interesting places such as the `glibc` headers, pending user demand.

  Because LLVM pretends to be like GCC 4.2 and this extension was introduced in GCC 4.3, the `glibc` headers currently do not try to use this extension with LLVM.

- Forward-declaring function parameters

  This extension has not shown up in any real-world code yet, though, so it might never be implemented.

11.1.3 Intentionally unsupported GCC extensions

LLVM intentionally does not implement the following GCC extensions:

- Variable-length arrays in structures

  This extension is not implemented for several reasons:

  - It is tricky to implement
  - The extension is completely undocumented
  - The extension appears to be rarely used

  LLVM supports flexible array members (arrays with a zero or unspecified size at the end of a structure).

- An equivalent to GCC's `fold`

  This extension 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 reliably implement.

11.1.4 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. For example:

```
addr = (float *)val;
```

11.1.5 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 it produces code that typically crashes when the result goes out of scope if the jump is taken. (It is possible for this bug to go undetected, because it often does not crash if the stack is fresh; that is, it 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 so it does not require jumping into the scope of a `__block` variable; for example, by limiting that scope:

```
{
  __block int result;
  run_specially_somehow(^{ result = c->state; });
  return result;
}
```

11.1.6 Non-initialization of __block variables

In the following example, 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 programs that depend on this behavior relies 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.

11.1.7 Inline assembly

Typically, LLVM is highly compatible with the GCC inline assembly extensions, allowing the same set of constraints, modifiers, and operands as GCC inline assembly.

11.2 C++ compatibility

This section describes common compatibility and portability issues with LLVM C++.

The types in C++ code that are compiled with `-std=c++11` are not inter-operable with the following:

- Types in C++ code that are compiled with `-std=c++03`
- Code compiled with no `-std=` specification

For example, the `std::string` from a C++03 compilation is unrelated to a `std::string` from a C++11 compilation, and the two `std::string` types cannot be easily converted to each other.

11.2.1 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:

https://svn.boost.org/trac/boost/changeset/73202

11.2.2 Variable-length arrays

GCC and C11 allow the size of an array to be determined at runtime. 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 C11 programs.

- The element type of a variable length array must be a *plain old data* (POD) type. 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 the code uses variable length arrays in a manner that LLVM does not support, several methods are available to fix the code:

- 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).
- Use `std::vector` or some other suitable container type.
- Allocate the array on the heap instead using `new Type[]`; remember to delete it using `delete[]`.

11.2.3 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 is 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, the compiler also does an argument-dependent lookup (ADL).

  In an 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 an ADL until it knows all the argument types.

In the example code, `Multiply` is called with dependent arguments, so an ADL is not done until the template is instantiated. At that point, the arguments both have type `int`, which does not contain any class types, and so the ADL does not look in any namespaces. Because neither form of lookup found the declaration of `Multiply`, the code does not compile.

Following is another example that uses 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) {
                ^
```

As before, unqualified lookup did not find any declarations with the name, `operator<<`. Unlike before, the argument types both contain class types:

- One type is an instance of the class template type, `std::basic_ostream`
- The other is the type `ns::Data` that is declared in the example

Therefore, the ADL looks in the `std` and `ns` namespaces for an `operator<<`. Because one of the argument types was still dependent during the template definition, the ADL is not done until the template is instantiated during `Use`, which means the `operator<<` it finds has already been declared. Unfortunately, it was declared in the global namespace, not in either of the namespaces that the ADL look in.

Two methods exist to fix this problem:

- 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. Either move the template definition or function definition, or add a forward declaration of the function before the template.

- Move the function into the same namespace as one of its arguments so that the ADL applies.

11.2.4 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 11.2.3, unqualified names such as `DoThis` and `DoThat` are looked up when the `Derived` template is defined, not when it is instantiated. When looking up a name used in a class, you typically look into the base classes. However, you cannot look into the `Base<T>` base class because its type depends on the template argument `T`, so the standard says to ignore it.

As LLVM indicates, the fix is to tell the compiler that you 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 approach works whether the methods are static or not, but be careful: if `DoThis` is virtual, calling it this way will bypass virtual dispatch.

11.2.5 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 &);
```

According to the standard, types that do not 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 example.

In LLVM, the decision to enforce the rule consistently provides a better experience, but unfortunately, it also results in some code that is rejected when other compilers will accept it.

11.2.6 Templates with no valid instantiations

The following code contains a typo: `innit()` was written instead of `init()`.

```
template <class T> class Processor {
  ...
  void init();
  ...
};

...

template <class T> void process() {
  Processor<T> processor;
  processor.innit();      // <-- should be 'init()'
  ...
}
```

Unfortunately, the compiler cannot flag this mistake as soon as it detects it. Inside a template, assumptions cannot be made about dependent types such as `Processor<T>`. Suppose that later in this file you add an explicit specialization of `Processor`, like the following example:

```
template <> class Processor<char*> {
  void innit();
};
```

The program will work, but only if you instantiate `process()` with `T = char*`. Therefore, it is hard, and sometimes impossible, to diagnose mistakes in a template definition before it is 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 precompiled 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: because the code is unused, remove it.

11.2.7 Default initialization of const variable of class type

The default initialization of a `const` variable of a class type requires a user-defined default constructor.

If a class or structure 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 issue, define a default constructor for the class:

```
class Foo {
public:
  Foo() {}
  ...
};

void Bar() {
  const Foo foo; // Now the compiler is happy.
  ...
}
```


11.2.8 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. For example:

```
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 Acknowledgments

QTI would like to thank the LLVM community for their many contributions to the LLVM Project.

B References

Title	Number
Qualcomm Technologies, Inc.	
Qualcomm Snapdragon LLVM ARM Linker User Guide	80-VB419-102
Version-specific Hexagon Programmer's Reference Manuals:	
■ *Hexagon V4 Programmer's Reference Manual*	80-N2040-9
■ *Hexagon V5x Programmer's Reference Manual*	80-N2040-8
■ *Hexagon V60/V61 Programmer's Reference Manual*	80-N2040-33
■ *Hexagon V62 Programmer's Reference Manual*	80-N2040-36
■ *Hexagon V62 Reference Manual For Programmers (Internal Use Only)*	80-N2040-80
■ *Qualcomm Hexagon V65 Programmer's Reference Manual*	80-N2040-39
■ *Qualcomm Hexagon V66 Programmer's Reference Manual*	80-N2040-42
■ *Qualcomm Hexagon V67 Programmer's Reference Manual*	80-N2040-45
■ *Qualcomm Hexagon V68 Programmer's Reference Manual*	80-N2040-46
Version-specific Hexagon HVX Programmer's Reference Manuals:	
■ *Hexagon V60 HVX Programmers Reference Manual*	80-N2040-30
■ *Hexagon V62 HVX Programmer's Reference Manual*	80-N2040-37
■ *Qualcomm Hexagon V65 HVX Programmer's Reference Manual*	80-N2040-41
■ *Qualcomm Hexagon V66 HVX Programmer's Reference Manual*	80-N2040-44
■ *Qualcomm Hexagon V68 HVX Programmer's Reference Manual*	80-N2040-47
C language references	
The C Programming Language (2nd Edition), Brian Kernighan and Dennis Ritchie, Prentice Hall, 1988	ISBN 0-13-110370-9
The C++ Programming Language (3rd Edition), Bjarne Stroustrup, Addison-Wesley, 1997	ISBN 0201889544
Compiler references	
Compilers: Principles, Techniques, and Tools (2nd Edition), Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman, Prentice Hall, 2006	ISBN 0-321-48681-1
Engineering a Compiler (2nd Edition), Keith Cooper and Linda Torczon, Morgan Kaufmann, 2011	ISBN-13: 978-0120884780