Clang - the C, C++ Compiler

Publication #	**57224**	Revision:	**3.1**
Issue Date:	**July 2021**		

Contents

List of Tables

Revision History

Date	Revision	Description
July 2021	3.1	Updated section 1.2.2.
March 2021	3.0	Initial version.

Chapter 1 Synopsis

Clang/Clang++ [*options*] *filename* ...

1.1 Description

Clang is a C, C++, and Objective-C compiler, which encompasses preprocessing, parsing, optimization, code generation, assembly, and linking. Depending on which high-level mode setting is passed, Clang will stop before doing a full link. While Clang is highly integrated, it is important to understand the stages of compilation and how to invoke Clang. These stages are as follows:

- **Driver**

  The Clang executable is a small driver, which controls the overall execution of other tools, such as the compiler, assembler, and linker. Typically, you need not interact with the driver, but you transparently use it to run the other tools.

- **Preprocessing**

  This stage handles tokenization of the input source file, macro expansion, #include expansion, and handling of the other preprocessor directives. The output of this stage is typically called a ".i" (for C), ".ii" (for C++), ".mi" (for Objective-C), or ".mii" (for Objective-C++) file.

- **Parsing and Semantic Analysis**

  This stage parses the input file, translating the preprocessor tokens into a parse tree. When in the form of a parse tree, it applies semantic analysis to compute types for expressions and to determine whether the code is well formed. This stage is responsible for generating most of the compiler warnings and parse errors. The output of this stage is an Abstract Syntax Tree (AST).

- **Code Generation and Optimization**

  This stage translates an AST into a Low-level Intermediate Code (LLVM IR) and ultimately, to a machine code. This phase is responsible for optimizing the generated code and handling the target-specific code generation. The output of this stage is typically called a *.s* or assembly file.

  Clang also supports the use of an integrated assembler, in which the code generator produces object files directly. This avoids the overhead of generating the *.s* file and of calling the target assembler.

- **Assembler**

  This stage runs the target assembler to translate the output of the compiler into a target object file. The output of this stage is typically called a *.o* or object file.

- **Linker**

  This stage runs the target linker to merge the multiple object files into an executable or dynamic library. The output of this stage is typically called an *a.out*, *.dylib* or *.so* file.

1.1.1 Support for Annex F (IEEE-754/IEC 559) of C99/C11

The Clang compiler does not support IEC 559 math functionality. Clang also does not control and honor the definition of __STDC_IEC_559__ macro. Under specific options such as -Ofast and -ffast-math, the compiler will enable a range of optimizations that provide faster mathematical operations that may not conform to the IEEE-754 specifications. The macro __STDC_IEC_559__ value may be defined but ignored when these faster optimizations are enabled.

1.2 Options

1.2.1 Target Selection

The following table lists all the target selection options:

Table 1. Target Selection Options

Option	Description
`-march=<cpu>`	Use it to specify if Clang must generate code for a specific processor family member and later. For example, if you specify -march=i486, the compiler can generate instructions that are valid on i486 and later processors, but which may not exist on the earlier ones.
`-march=znver1`	Use this architecture option for enabling the best code generation and tuning for AMD Zen based x86 architecture. All the x86 Zen ISA and associated intrinsic are supported.
`-march=znver2`	Use this architecture option for enabling the best code generation and tuning for AMD Zen2 based x86 architecture. All x86 Zen2 ISA and associated intrinsic are supported.
`-march=znver3`	Use this architecture option for enabling best code generation and tuning for AMD Zen3 based x86 architecture. All x86 Zen3 ISA and associated intrinsic are supported.

1.2.2 Code Generation

The following table lists all the code generation options:

Table 2. Code Generation Options

Level	Description
-O0	Means no optimization: this level compiles the fastest and generates the most debuggable code.
-O1	Somewhere between the levels -O0 and -O2.
-O2	A moderate level of optimization, which enables most optimizations.
-O3	Similar to the level -O2, except that it enables the optimizations, which take longer to perform or may generate larger code (in an attempt to make the program run faster). The -O3 level in AOCC has more optimizations when compared to the base LLVM version on which it is based. These optimizations include improved handling of indirect calls, advanced vectorization, and so on.
-Ofast	Enables all the optimizations from -O3 along with other aggressive optimizations that may violate strict compliance with language standards. The -Ofast level in AOCC has more optimizations when compared to the base LLVM version on which it is based. These optimizations include partial unswitching, improvements to inlining, unrolling, and so on.
-Os	Similar to the level -O2, but with extra optimizations to reduce the code size.
-Oz	Similar to the level -Os (and thus, -O2), but reduces the code size further.
-O	Equivalent to the level O2.
-O4 and higher	Equivalent to the level O3.

For more information on these options, refer *LLVM documentation*.

The following optimizations are not present in LLVM and are specific to AOCC:

- **`-fstruct-layout`**`=[1,2,3,4,5,6,7]`

  Analyzes the whole program to determine if the structures in the code can be peeled and if the pointer or integer fields in the structure can be compressed. If feasible, this optimization transforms the code to enable these improvements. This transformation is likely to improve cache utilization and memory bandwidth. It is expected to improve the scalability of programs executed on multiple cores.
  This is effective only under *flto* as the whole program analysis is required to perform this optimization. You can choose different levels of aggressiveness with which this optimization can be applied to your application; with 1 being the least aggressive and 7 being the most aggressive level.

  – **`fstruct-layout`**`=1` enables structure peeling.

- `fstruct-layout=2` enables structure peeling and selectively compresses self-referential pointers in these structures to 32-bit pointers, wherever safe.
- `fstruct-layout=3` enables structure peeling and selectively compresses self-referential pointers in these structures to 16-bit pointers, wherever safe.
- `fstruct-layout=4` enables structure peeling, pointer compression as in level 2 and further enables compression of structure fields, which are of integer type. This is performed under a strict safety check.
- `fstruct-layout=5` enables structure peeling, pointer compression as in level 3 and further enables compression of structure fields which are of integer type. This is performed under a strict safety check.
- `fstruct-layout=6` enables structure peeling, pointer compression as in level 2 and further enables compression of structure fields, which are of type 64-bit **signed int** or **unsigned int**. You must ensure that the values assigned to 64-bit **signed int** fields are in range -(2^31 - 1) to +(2^31 - 1) and 64-bit **unsigned int** fields are in the range 0 to +(2^31 - 1). Else, incorrect results may be obtained. This compression is performed without considering any safety analysis. So, you must ensure the safety based on the program compiled.
- `fstruct-layout=7` enables structure peeling, pointer compression as in level 3 and further enables compression of structure fields, which are of type 64-bit **signed int** or **unsigned int**. You must ensure that the values assigned to 64-bit **signed int** fields are in range -(2^31 - 1) to +(2^31 - 1) and 64-bit **unsigned int** fields are in the range 0 to +(2^31 - 1). Else, incorrect results may be obtained. This compression is performed without considering any safety analysis. So, must ensure the safety based on the program compiled.

Notes:

1. `fstruct-layout=4` *and* `fstruct-layout=5` *are derived from* `fstruct-layout=2` *and* `fstruct-layout=3` *respectively, with the added feature of safe compression of integer fields in structures. Going from* `fstruct-layout=4` *to* `fstruct-layout=5` *may result in higher performance if the pointer values are such that the pointers can be compressed to 16-bits.*
2. `fstruct-layout=6` *and* `fstruct-layout=7` *are derived from* `fstruct-layout=2` *and* `fstruct-layout=3` *respectively, with the added feature of compression of the integer fields in structures. These are similar to* `fstruct-layout=4` *and* `fstruct-layout=5`*, but here, the integer fields of the structures are always compressed from 64-bits to 32-bits, without any safety guarantee.*

- **`-fitodcalls`**

  It promotes indirect to direct calls by placing conditional calls. Application or benchmarks that have small and deterministic set of target functions for function pointers that are passed as call parameters benefit from this optimization. Indirect-to-direct call promotion transforms the code to use all possible determined targets under runtime checks and falls back to the original code for all the other cases. Runtime checks are introduced by the compiler for each of these possible function pointer targets followed by direct calls to the targets.

  This is a link time optimization, which is invoked as `-flto -fitodcalls`.

- **`-fitodcallsbyclone`**

  Performs value specialization for functions with function pointers passed as an argument. It does this specialization by generating a clone of the function. The cloning of the function happens in the call chain as needed to allow conversion of indirect function call to direct call. This complements `-fitodcalls` optimization and is also a link time optimization, which is invoked as `-flto -fitodcallsbyclone`.

- **`-fremap-arrays`**

  Transforms the data layout of a single dimensional array to provide better cache locality. This optimization is effective only under *flto* as the whole program analysis is required to perform this optimization, which can be invoked as `-flto -fremap-arrays`.

- **`-finline-aggressive`**

  Enables improved inlining capability through better heuristics. This optimization is more effective when using with *flto* as the whole program analysis is required to perform this optimization, which can be invoked as `-flto -finline-aggressive`.

- **`-fnt-store`**

  Generates a non-temporal store instruction for array accesses in a loop with a large trip count.

- **`-fnt-store=aggressive`**

  This is an experimental option to generate non-temporal store instruction for array accesses in a loop, whose iteration count cannot be determined at compile time. In this case, compiler assumes the iteration count is huge.

The following optimization options must be invoked through driver `-mllvm <options>` as follows:

- **`-enable-partial-unswitch`**

  Enables partial loop un-switching, which is an enhancement to the existing loop unswitching optimization in LLVM. Partial loop un-switching hoists a condition inside a loop from a path for which the execution condition remains invariant, whereas the original loop un-switching works for a condition that is completely loop invariant. The condition inside the loop gets hoisted out from the invariant path and original loop is retained for the path where condition is variant.

- **`-aggressive-loop-unswitch`**

  Experimental option which enables aggressive loop unswitching heuristic (including `-enable-partial-unswitch`) based on the usage of the branch conditional values. Loop unswitching leads to code-bloat. Code-bloat can be minimized if the hoisted condition is executed more often. This heuristic prioritizes the conditions based on the number of times they are used within the loop. The heuristic can be controlled with the following options:

  – `-unswitch-identical-branches-min-count=<n>`

    Enables unswitching of a loop with respect to a branch conditional value (B), where B appears in at least <**n**> compares in the loop. This option is enabled with -aggressive-loop-unswitch. The default value is 3.

    Usage: -mllvm -aggressive-loop-unswitch -mllvm -unswitch-identical-branches-min-count=<n>

    Where, **n** is a positive integer and lower value of <**n**> facilitates more unswitching.

– `-unswitch-identical-branches-max-count=<n>`

  Enables unswitching of a loop with respect to a branch conditional value (B), where B appears in at most **<n>** compares in the loop. This option is enabled with -aggressive-loop-unswitch. The default value is 6.

  Usage: -mllvm -aggressive-loop-unswitch -mllvm -unswitch-identical-branches-max-count=<n>

  Where, **n** is a positive integer and higher value of **<n>** facilitates more unswitching.

Note: *These options may facilitate more unswitching in some of the workloads. Since, loop-unswitching inherently leads to code-bloat, facilitating more unswitching may significantly increase the code size. Hence, it may also lead to longer compilation times.*

- **`-enable-strided-vectorization`**

  Enables strided memory vectorization as an enhancement to the interleaved vectorization framework present in LLVM. It enables the effective use of gather and scatter kind of instruction patterns. This option must be used along with the interleave vectorization option.

- **`-enable-epilog-vectorization`**

  Enables vectorization of epilog-iterations as an enhancement to existing vectorization framework. This enables generation of an additional epilog vector loop version for the remainder iterations of the original vector loop. The vector size or factor of the original loop should be large enough to allow an effective epilog vectorization of the remaining iterations. This optimization takes place only when the original vector loop is vectorized with a vector width or factor of sixteen. This vectorization width of sixteen may be overwritten by -min-width-epilog-vectorization command line option.

- **`-enable-redundant-movs`**

  Removes any redundant mov operations including redundant loads from memory and stores to memory. This can be invoked using `-Wl,-plugin-opt=-enable-redundant-movs`.

- **`-merge-constant`**

  Attempts to promote frequently occurring constants to registers. The aim is to reduce the size of the instruction encoding for instructions using constants and obtain a performance improvement.

- **`-function-specialize`**

  Optimizes the functions with compile time constant formal arguments.

- **`-lv-function-specialization`**

  Generates specialized function versions when the loops inside function are vectorizable and the arguments are not aliased with each other.

- **`-enable-vectorize-compares`**

  Enables vectorization on certain loops with conditional breaks assuming the memory access are safely bound within the page boundary.

- **`-inline-recursion=[1,2,3,4]`**

  Enables inlining for recursive functions based on heuristics with level 4 being most aggressive. The default level will be 2. Higher levels may lead to code-bloat due to expansion of recursive functions at call sites.

  – For level 1-2: Enables inlining for recursive functions using heuristics with inline depth 1. Level 2 uses more aggressive heuristics.

  – For level 3: Enables inlining for all recursive functions with inline depth 1.

  – For level 4: Enables inlining for all recursive function with inline depth 10.

  This is more effective with flto as the whole program analysis is required to perform this optimization, which can be invoked as `-flto -inline-recursion=[1,2,3,4]`.

- **`-reduce-array-computations=[1,2,3]`**

  Performs array dataflow analysis and optimizes the unused array computations.

  – `reduce-array-computations=1`: Eliminates the computations on unused array elements.

  – `reduce-array-computations=2`: Eliminates the computations on zero valued array elements.

  – `reduce-array-computations=3`: Eliminates the computations on unused and zero valued array elements (combination of 1 and 2).

    This optimization is effective with flto as the whole program analysis is required to perform this optimization, which can be invoked as `-flto -reduce-array-computations=[1,2,3]`.

- **`-global-vectorize-slp={true,false}`**

  Vectorizes the straight-line code inside a basic block with data reordering vector operations. This option is set to **true** by default.

- **`-region-vectorize`**

  Experimental option for enabling vectorization on certain loops with complex control flow which the normal vectorizer cannot handle.
  This optimization is effective with *flto* as the whole program analysis is required to perform this optimization, which can be invoked as `-flto -region-vectorize`.

- **`-enable-X86-prefetching`**
  Enables the generation of x86 prefetch instruction for the memory references inside a loop/ inside an inner most loop of a loop nest to prefetch the second dimension of multidimensional array/memory references in the inner most of a loop nest. This is an experimental pass; its profitability is being improved.

- **`-suppress-fmas`**
  Identifies the reduction patterns on FMA and suppresses the FMA generation as it is not profitable on the reduction patterns.

- **`-enable-licm-vrp`**
  Enables estimation of the virtual register pressure before performing loop invariant code motion. This estimation is used to control the number of loop invariants that will be hoisted during the loop invariant code motion.

- **`-loop-splitting`**
  Enables splitting of loops into multiple loops to eliminate the branches, which compare the loop induction with an invariant or constant expression. This option is enabled under -O3 by default. To disable this optimization, use `-loop-splitting=false`.
- **`-enable-ipo-loop-split`**
  Enables splitting of loops into multiple loops to eliminate the branches, which compares the loop induction with a constant expression. This constant expression can be derived through inter-procedural analysis. This option is enabled under -O3 by default. To disable this optimization, use `-enable-ipo-loop-split=false`.
- **`-compute-interchange-order`**
  Enables heuristic for finding the best possible interchange order for a loop nest. To enable this option, use `-enable-loopinterchange`. This option is set to **false** by default.
  Usage: `-mllvm -enable-loopinterchange -mllvm -compute-interchange-order`
- **`-convert-pow-exp-to-int={true,false}`**
  Converts the call to floating point exponent version of pow to its integer exponent version if the floating-point exponent can be converted to integer. This option is set to **true** by default.
- `-`**`do-block-reordering={none,normal,aggressive}`**
  Reorders the control predicates in increasing order of complexity from outer predicate to inner when it is safe. The **normal** mode reorders simple expressions while the **aggressive** mode will reorder predicates involving function calls if it can determine that they have no side-effects. This option is set to **normal** by default.
- **`-fuse-tile-inner-loop`**
  Enables fusion of adjacent tiled loops as a part of loop tiling transformation. This option is set to **false** by default.
- **`-enable-loop-vectorization-with-conditions`**

  Enables efficient vectorization of loops with conditions by conditionally executing the vector instructions as opposed to flattening the loop body and vectorizing. The vectorized code uses vector versions of compare instructions to guard the instructions in the loop body and uses masked instructions to guard against unsafe memory operations.
- **`-favoid-fpe-causing-opt`**

  Restricts a few optimizations that leads to floating point exceptions.

1.2.3 Deprecated Options

- **`-vectorize-memory-aggressively`** (from AOCC 2.2.0)

1.2.4 Driver

- **`-mllvm <options>`**

  Need to provide `-mllvm`, so that, the option can pass through the compiler front end and is applied on the optimizer where this optimization is implemented.

  For example, `-mllvm -enable-strided-vectorization`
- **`-fuse-ld=lld`**

  To invoke *lld* linker from compiler driver as it is the preferred linker.

Note: *For more information on the Clang options, refer Clang Documentation.*