3.11 Options That Control Optimization

These options control various sorts of optimizations.

Without any optimization option, the compiler’s goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you expect from the source code.

Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

The compiler performs optimization based on the knowledge it has of the program. Compiling multiple files at once to a single output file mode allows the compiler to use information gained from all of the files when compiling each of them.

Not all optimizations are controlled directly by a flag. Only optimizations that have a flag are listed in this section.

Most optimizations are completely disabled at `-O0` or if an `-O` level is not set on the command line, even if individual optimization flags are specified. Similarly, `-Og` suppresses many optimization passes.

Depending on the target and how GCC was configured, a slightly different set of optimizations may be enabled at each `-O` level than those listed here. You can invoke GCC with `-Q --help=optimizers` to find out the exact set of optimizations that are enabled at each level. See Overall Options, for examples.

`-O`
`-O1`

Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

With `-O`, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

`-O` turns on the following optimization flags:

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions-called-once
-fipa-modref
-fipa-profile
-fipa-pure-const
-fipa-reference
-fipa-reference-addressable
-fmerge-constants
-fmove-loop-invariants
-fmove-loop-stores
-fomit-frame-pointer
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-pta
-ftree-scev-cprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-ter
-funit-at-a-time
```

`-O2`

Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to `-O`, this option increases both compilation time and the performance of the generated code.

`-O2` turns on all optimization flags specified by `-O1`. It also turns on the following optimization flags:

```
-falign-functions  -falign-jumps
-falign-labels  -falign-loops
-fcaller-saves
-fcode-hoisting
-fcrossjumping
-fcse-follow-jumps  -fcse-skip-blocks
-fdelete-null-pointer-checks
-fdevirtualize  -fdevirtualize-speculatively
-fexpensive-optimizations
-ffinite-loops
-fgcse  -fgcse-lm
-fhoist-adjacent-loads
-finline-functions
-finline-small-functions
-findirect-inlining
-fipa-bit-cp  -fipa-cp  -fipa-icf
-fipa-ra  -fipa-sra  -fipa-vrp
-fisolate-erroneous-paths-dereference
-flra-remat
-foptimize-sibling-calls
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-freorder-blocks-algorithm=stc
-freorder-blocks-and-partition  -freorder-functions
-frerun-cse-after-loop
-fschedule-insns  -fschedule-insns2
-fsched-interblock  -fsched-spec
-fstore-merging
-fstrict-aliasing
-fthread-jumps
-ftree-builtin-call-dce
-ftree-loop-vectorize
-ftree-pre
-ftree-slp-vectorize
-ftree-switch-conversion  -ftree-tail-merge
-ftree-vrp
-fvect-cost-model=very-cheap
```

Please note the warning under `-fgcse` about invoking `-O2` on programs that use computed gotos.

`-O3`

Optimize yet more. `-O3` turns on all optimizations specified by `-O2` and also turns on the following optimization flags:

```
-fgcse-after-reload
-fipa-cp-clone
-floop-interchange
-floop-unroll-and-jam
-fpeel-loops
-fpredictive-commoning
-fsplit-loops
-fsplit-paths
-ftree-loop-distribution
-ftree-partial-pre
-funswitch-loops
-fvect-cost-model=dynamic
-fversion-loops-for-strides
```

`-O0`

Reduce compilation time and make debugging produce the expected results. This is the default.

`-Os`

Optimize for size. `-Os` enables all `-O2` optimizations except those that often increase code size:

```
-falign-functions  -falign-jumps
-falign-labels  -falign-loops
-fprefetch-loop-arrays  -freorder-blocks-algorithm=stc
```

It also enables `-finline-functions`, causes the compiler to tune for code size rather than execution speed, and performs further optimizations designed to reduce code size.

`-Ofast`

Disregard strict standards compliance. `-Ofast` enables all `-O3` optimizations. It also enables optimizations that are not valid for all standard-compliant programs. It turns on `-ffast-math`, `-fallow-store-data-races` and the Fortran-specific `-fstack-arrays`, unless `-fmax-stack-var-size` is specified, and `-fno-protect-parens`. It turns off `-fsemantic-interposition`.

`-Og`

Optimize debugging experience. `-Og` should be the optimization level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience. It is a better choice than `-O0` for producing debuggable code because some compiler passes that collect debug information are disabled at `-O0`.

Like `-O0`, `-Og` completely disables a number of optimization passes so that individual options controlling them have no effect. Otherwise `-Og` enables all `-O1` optimization flags except for those that may interfere with debugging:

```
-fbranch-count-reg  -fdelayed-branch
-fdse  -fif-conversion  -fif-conversion2
-finline-functions-called-once
-fmove-loop-invariants  -fmove-loop-stores  -fssa-phiopt
-ftree-bit-ccp  -ftree-dse  -ftree-pta  -ftree-sra
```

`-Oz`

Optimize aggressively for size rather than speed. This may increase the number of instructions executed if those instructions require fewer bytes to encode. `-Oz` behaves similarly to `-Os` including enabling most `-O2` optimizations.

If you use multiple `-O` options, with or without level numbers, the last such option is the one that is effective.

Options of the form `-f`*`flag`* specify machine-independent flags. Most flags have both positive and negative forms; the negative form of `-ffoo` is `-fno-foo`. In the table below, only one of the forms is listed—the one you typically use. You can figure out the other form by either removing ‘`no-`’ or adding it.

The following options control specific optimizations. They are either activated by `-O` options or are related to ones that are. You can use the following flags in the rare cases when “fine-tuning” of optimizations to be performed is desired.

`-fno-defer-pop`

For machines that must pop arguments after a function call, always pop the arguments as soon as each function returns. At levels `-O1` and higher, `-fdefer-pop` is the default; this allows the compiler to let arguments accumulate on the stack for several function calls and pop them all at once.

`-fforward-propagate`

Perform a forward propagation pass on RTL. The pass tries to combine two instructions and checks if the result can be simplified. If loop unrolling is active, two passes are performed and the second is scheduled after loop unrolling.

This option is enabled by default at optimization levels `-O1`, `-O2`, `-O3`, `-Os`.

`-ffp-contract=`*`style`*

`-ffp-contract=off` disables floating-point expression contraction. `-ffp-contract=fast` enables floating-point expression contraction such as forming of fused multiply-add operations if the target has native support for them. `-ffp-contract=on` enables floating-point expression contraction if allowed by the language standard. This is currently not implemented and treated equal to `-ffp-contract=off`.

The default is `-ffp-contract=fast`.

`-fomit-frame-pointer`

Omit the frame pointer in functions that don't need one. This avoids the instructions to save, set up and restore the frame pointer; on many targets it also makes an extra register available.

On some targets this flag has no effect because the standard calling sequence always uses a frame pointer, so it cannot be omitted.

Note that `-fno-omit-frame-pointer` doesn't guarantee the frame pointer is used in all functions. Several targets always omit the frame pointer in leaf functions.

Enabled by default at `-O1` and higher.

`-foptimize-sibling-calls`

Optimize sibling and tail recursive calls.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-foptimize-strlen`

Optimize various standard C string functions (e.g. `strlen`, `strchr` or `strcpy`) and their `_FORTIFY_SOURCE` counterparts into faster alternatives.

Enabled at levels `-O2`, `-O3`.

`-fno-inline`

Do not expand any functions inline apart from those marked with the `always_inline` attribute. This is the default when not optimizing.

Single functions can be exempted from inlining by marking them with the `noinline` attribute.

`-finline-small-functions`

Integrate functions into their callers when their body is smaller than expected function call code (so overall size of program gets smaller). The compiler heuristically decides which functions are simple enough to be worth integrating in this way. This inlining applies to all functions, even those not declared inline.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-findirect-inlining`

Inline also indirect calls that are discovered to be known at compile time thanks to previous inlining. This option has any effect only when inlining itself is turned on by the `-finline-functions` or `-finline-small-functions` options.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-finline-functions`

Consider all functions for inlining, even if they are not declared inline. The compiler heuristically decides which functions are worth integrating in this way.

If all calls to a given function are integrated, and the function is declared `static`, then the function is normally not output as assembler code in its own right.

Enabled at levels `-O2`, `-O3`, `-Os`. Also enabled by `-fprofile-use` and `-fauto-profile`.

`-finline-functions-called-once`

Consider all `static` functions called once for inlining into their caller even if they are not marked `inline`. If a call to a given function is integrated, then the function is not output as assembler code in its own right.

Enabled at levels `-O1`, `-O2`, `-O3` and `-Os`, but not `-Og`.

`-fearly-inlining`

Inline functions marked by `always_inline` and functions whose body seems smaller than the function call overhead early before doing `-fprofile-generate` instrumentation and real inlining pass. Doing so makes profiling significantly cheaper and usually inlining faster on programs having large chains of nested wrapper functions.

Enabled by default.

`-fipa-sra`

Perform interprocedural scalar replacement of aggregates, removal of unused parameters and replacement of parameters passed by reference by parameters passed by value.

Enabled at levels `-O2`, `-O3` and `-Os`.

`-finline-limit=`*n*

By default, GCC limits the size of functions that can be inlined. This flag allows coarse control of this limit. *n* is the size of functions that can be inlined in number of pseudo instructions.

Inlining is actually controlled by a number of parameters, which may be specified individually by using `--param` *`name`*`=`*`value`*. The `-finline-limit=`*n* option sets some of these parameters as follows:

`max-inline-insns-single`

is set to *n*/2.

`max-inline-insns-auto`

is set to *n*/2.

See below for a documentation of the individual parameters controlling inlining and for the defaults of these parameters.

Note: there may be no value to `-finline-limit` that results in default behavior.

Note: pseudo instruction represents, in this particular context, an abstract measurement of function’s size. In no way does it represent a count of assembly instructions and as such its exact meaning might change from one release to an another.

`-fno-keep-inline-dllexport`

This is a more fine-grained version of `-fkeep-inline-functions`, which applies only to functions that are declared using the `dllexport` attribute or declspec. See Declaring Attributes of Functions.

`-fkeep-inline-functions`

In C, emit `static` functions that are declared `inline` into the object file, even if the function has been inlined into all of its callers. This switch does not affect functions using the `extern inline` extension in GNU C90. In C++, emit any and all inline functions into the object file.

`-fkeep-static-functions`

Emit `static` functions into the object file, even if the function is never used.

`-fkeep-static-consts`

Emit variables declared `static const` when optimization isn't turned on, even if the variables aren't referenced.

GCC enables this option by default. If you want to force the compiler to check if a variable is referenced, regardless of whether or not optimization is turned on, use the `-fno-keep-static-consts` option.

`-fmerge-constants`

Attempt to merge identical constants (string constants and floating-point constants) across compilation units.

This option is the default for optimized compilation if the assembler and linker support it. Use `-fno-merge-constants` to inhibit this behavior.

Enabled at levels `-O1`, `-O2`, `-O3`, `-Os`.

`-fmerge-all-constants`

Attempt to merge identical constants and identical variables.

This option implies `-fmerge-constants`. In addition to `-fmerge-constants` this considers e.g. even constant initialized arrays or initialized constant variables with integral or floating-point types. Languages like C or C++ require each variable, including multiple instances of the same variable in recursive calls, to have distinct locations, so using this option results in non-conforming behavior.

`-fmodulo-sched`

Perform swing modulo scheduling immediately before the first scheduling pass. This pass looks at innermost loops and reorders their instructions by overlapping different iterations.

`-fmodulo-sched-allow-regmoves`

Perform more aggressive SMS-based modulo scheduling with register moves allowed. By setting this flag certain anti-dependences edges are deleted, which triggers the generation of reg-moves based on the life-range analysis. This option is effective only with `-fmodulo-sched` enabled.

`-fno-branch-count-reg`

Disable the optimization pass that scans for opportunities to use "decrement and branch" instructions on a count register instead of instruction sequences that decrement a register, compare it against zero, and then branch based upon the result. This option is only meaningful on architectures that support such instructions, which include x86, PowerPC, IA-64 and S/390. Note that the `-fno-branch-count-reg` option doesn't remove the decrement and branch instructions from the generated instruction stream introduced by other optimization passes.

The default is `-fbranch-count-reg` at `-O1` and higher, except for `-Og`.

`-fno-function-cse`

Do not put function addresses in registers; make each instruction that calls a constant function contain the function’s address explicitly.

This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

The default is `-ffunction-cse`

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

If the target supports a BSS section, GCC by default puts variables that are initialized to zero into BSS. This can save space in the resulting code.

This option turns off this behavior because some programs explicitly rely on variables going to the data section—e.g., so that the resulting executable can find the beginning of that section and/or make assumptions based on that.

The default is `-fzero-initialized-in-bss`.

`-fthread-jumps`

Perform optimizations that check to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.

Enabled at levels `-O1`, `-O2`, `-O3`, `-Os`.

`-fsplit-wide-types`

When using a type that occupies multiple registers, such as `long long` on a 32-bit system, split the registers apart and allocate them independently. This normally generates better code for those types, but may make debugging more difficult.

Enabled at levels `-O1`, `-O2`, `-O3`, `-Os`.

`-fsplit-wide-types-early`

Fully split wide types early, instead of very late. This option has no effect unless `-fsplit-wide-types` is turned on.

This is the default on some targets.

`-fcse-follow-jumps`

In common subexpression elimination (CSE), scan through jump instructions when the target of the jump is not reached by any other path. For example, when CSE encounters an `if` statement with an `else` clause, CSE follows the jump when the condition tested is false.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-fcse-skip-blocks`

This is similar to `-fcse-follow-jumps`, but causes CSE to follow jumps that conditionally skip over blocks. When CSE encounters a simple `if` statement with no else clause, `-fcse-skip-blocks` causes CSE to follow the jump around the body of the `if`.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-frerun-cse-after-loop`

Re-run common subexpression elimination after loop optimizations are performed.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-fgcse`

Perform a global common subexpression elimination pass. This pass also performs global constant and copy propagation.

Note: When compiling a program using computed gotos, a GCC extension, you may get better run-time performance if you disable the global common subexpression elimination pass by adding `-fno-gcse` to the command line.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-fgcse-lm`

When `-fgcse-lm` is enabled, global common subexpression elimination attempts to move loads that are only killed by stores into themselves. This allows a loop containing a load/store sequence to be changed to a load outside the loop, and a copy/store within the loop.

Enabled by default when `-fgcse` is enabled.

`-fgcse-sm`

When `-fgcse-sm` is enabled, a store motion pass is run after global common subexpression elimination. This pass attempts to move stores out of loops. When used in conjunction with `-fgcse-lm`, loops containing a load/store sequence can be changed to a load before the loop and a store after the loop.

Not enabled at any optimization level.

`-fgcse-las`

When `-fgcse-las` is enabled, the global common subexpression elimination pass eliminates redundant loads that come after stores to the same memory location (both partial and full redundancies).

Not enabled at any optimization level.

`-fgcse-after-reload`

When `-fgcse-after-reload` is enabled, a redundant load elimination pass is performed after reload. The purpose of this pass is to clean up redundant spilling.

Enabled by `-O3`, `-fprofile-use` and `-fauto-profile`.

`-faggressive-loop-optimizations`

This option tells the loop optimizer to use language constraints to derive bounds for the number of iterations of a loop. This assumes that loop code does not invoke undefined behavior by for example causing signed integer overflows or out-of-bound array accesses. The bounds for the number of iterations of a loop are used to guide loop unrolling and peeling and loop exit test optimizations. This option is enabled by default.

`-funconstrained-commons`

This option tells the compiler that variables declared in common blocks (e.g. Fortran) may later be overridden with longer trailing arrays. This prevents certain optimizations that depend on knowing the array bounds.

`-fcrossjumping`

Perform cross-jumping transformation. This transformation unifies equivalent code and saves code size. The resulting code may or may not perform better than without cross-jumping.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-fauto-inc-dec`

Combine increments or decrements of addresses with memory accesses. This pass is always skipped on architectures that do not have instructions to support this. Enabled by default at `-O1` and higher on architectures that support this.

`-fdce`

Perform dead code elimination (DCE) on RTL. Enabled by default at `-O1` and higher.

`-fdse`

Perform dead store elimination (DSE) on RTL. Enabled by default at `-O1` and higher.

`-fif-conversion`

Attempt to transform conditional jumps into branch-less equivalents. This includes use of conditional moves, min, max, set flags and abs instructions, and some tricks doable by standard arithmetics. The use of conditional execution on chips where it is available is controlled by `-fif-conversion2`.

Enabled at levels `-O1`, `-O2`, `-O3`, `-Os`, but not with `-Og`.

`-fif-conversion2`

Use conditional execution (where available) to transform conditional jumps into branch-less equivalents.

Enabled at levels `-O1`, `-O2`, `-O3`, `-Os`, but not with `-Og`.

`-fdeclone-ctor-dtor`

The C++ ABI requires multiple entry points for constructors and destructors: one for a base subobject, one for a complete object, and one for a virtual destructor that calls operator delete afterwards. For a hierarchy with virtual bases, the base and complete variants are clones, which means two copies of the function. With this option, the base and complete variants are changed to be thunks that call a common implementation.

Enabled by `-Os`.

`-fdelete-null-pointer-checks`

Assume that programs cannot safely dereference null pointers, and that no code or data element resides at address zero. This option enables simple constant folding optimizations at all optimization levels. In addition, other optimization passes in GCC use this flag to control global dataflow analyses that eliminate useless checks for null pointers; these assume that a memory access to address zero always results in a trap, so that if a pointer is checked after it has already been dereferenced, it cannot be null.

Note however that in some environments this assumption is not true. Use `-fno-delete-null-pointer-checks` to disable this optimization for programs that depend on that behavior.

This option is enabled by default on most targets. On Nios II ELF, it defaults to off. On AVR and MSP430, this option is completely disabled.

Passes that use the dataflow information are enabled independently at different optimization levels.

`-fdevirtualize`

Attempt to convert calls to virtual functions to direct calls. This is done both within a procedure and interprocedurally as part of indirect inlining (`-findirect-inlining`) and interprocedural constant propagation (`-fipa-cp`). Enabled at levels `-O2`, `-O3`, `-Os`.

`-fdevirtualize-speculatively`

Attempt to convert calls to virtual functions to speculative direct calls. Based on the analysis of the type inheritance graph, determine for a given call the set of likely targets. If the set is small, preferably of size 1, change the call into a conditional deciding between direct and indirect calls. The speculative calls enable more optimizations, such as inlining. When they seem useless after further optimization, they are converted back into original form.

`-fdevirtualize-at-ltrans`

Stream extra information needed for aggressive devirtualization when running the link-time optimizer in local transformation mode. This option enables more devirtualization but significantly increases the size of streamed data. For this reason it is disabled by default.

`-fexpensive-optimizations`

Perform a number of minor optimizations that are relatively expensive.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-free`

Attempt to remove redundant extension instructions. This is especially helpful for the x86-64 architecture, which implicitly zero-extends in 64-bit registers after writing to their lower 32-bit half.

Enabled for Alpha, AArch64 and x86 at levels `-O2`, `-O3`, `-Os`.

`-fno-lifetime-dse`

In C++ the value of an object is only affected by changes within its lifetime: when the constructor begins, the object has an indeterminate value, and any changes during the lifetime of the object are dead when the object is destroyed. Normally dead store elimination will take advantage of this; if your code relies on the value of the object storage persisting beyond the lifetime of the object, you can use this flag to disable this optimization. To preserve stores before the constructor starts (e.g. because your operator new clears the object storage) but still treat the object as dead after the destructor, you can use `-flifetime-dse=1`. The default behavior can be explicitly selected with `-flifetime-dse=2`. `-flifetime-dse=0` is equivalent to `-fno-lifetime-dse`.

`-flive-range-shrinkage`

Attempt to decrease register pressure through register live range shrinkage. This is helpful for fast processors with small or moderate size register sets.

`-fira-algorithm=`*`algorithm`*

Use the specified coloring algorithm for the integrated register allocator. The *algorithm* argument can be '`priority`', which specifies Chow's priority coloring, or '`CB`', which specifies Chaitin-Briggs coloring. Chaitin-Briggs coloring is not implemented for all architectures, but for those targets that do support it, it is the default because it generates better code.

`-fira-region=`*`region`*

Use specified regions for the integrated register allocator. The *region* argument should be one of the following:

`‘all’`

Use all loops as register allocation regions. This can give the best results for machines with a small and/or irregular register set.

`‘mixed’`

Use all loops except for loops with small register pressure as the regions. This value usually gives the best results in most cases and for most architectures, and is enabled by default when compiling with optimization for speed (`-O`, `-O2`, …).

`‘one’`

Use all functions as a single region. This typically results in the smallest code size, and is enabled by default for `-Os` or `-O0`.

`-fira-hoist-pressure`

Use IRA to evaluate register pressure in the code hoisting pass for decisions to hoist expressions. This option usually results in smaller code, but it can slow the compiler down.

This option is enabled at level `-Os` for all targets.

`-fira-loop-pressure`

Use IRA to evaluate register pressure in loops for decisions to move loop invariants. This option usually results in generation of faster and smaller code on machines with large register files (>= 32 registers), but it can slow the compiler down.

This option is enabled at level `-O3` for some targets.

`-fno-ira-share-save-slots`

Disable sharing of stack slots used for saving call-used hard registers living through a call. Each hard register gets a separate stack slot, and as a result function stack frames are larger.

`-fno-ira-share-spill-slots`

Disable sharing of stack slots allocated for pseudo-registers. Each pseudo-register that does not get a hard register gets a separate stack slot, and as a result function stack frames are larger.

`-flra-remat`

Enable CFG-sensitive rematerialization in LRA. Instead of loading values of spilled pseudos, LRA tries to rematerialize (recalculate) values if it is profitable.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-fdelayed-branch`

If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.

Enabled at levels `-O1`, `-O2`, `-O3`, `-Os`, but not at `-Og`.

`-fschedule-insns`

If supported for the target machine, attempt to reorder instructions to eliminate execution stalls due to required data being unavailable. This helps machines that have slow floating point or memory load instructions by allowing other instructions to be issued until the result of the load or floating-point instruction is required.

Enabled at levels `-O2`, `-O3`.

`-fschedule-insns2`

Similar to `-fschedule-insns`, but requests an additional pass of instruction scheduling after register allocation has been done. This is especially useful on machines with a relatively small number of registers and where memory load instructions take more than one cycle.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-fno-sched-interblock`

Disable instruction scheduling across basic blocks, which is normally enabled when scheduling before register allocation, i.e. with `-fschedule-insns` or at `-O2` or higher.

`-fno-sched-spec`

Disable speculative motion of non-load instructions, which is normally enabled when scheduling before register allocation, i.e. with `-fschedule-insns` or at `-O2` or higher.

`-fsched-pressure`

Enable register pressure sensitive insn scheduling before register allocation. This only makes sense when scheduling before register allocation is enabled, i.e. with `-fschedule-insns` or at `-O2` or higher. Usage of this option can improve the generated code and decrease its size by preventing register pressure increase above the number of available hard registers and subsequent spills in register allocation.

`-fsched-spec-load`

Allow speculative motion of some load instructions. This only makes sense when scheduling before register allocation, i.e. with `-fschedule-insns` or at `-O2` or higher.

`-fsched-spec-load-dangerous`

Allow speculative motion of more load instructions. This only makes sense when scheduling before register allocation, i.e. with `-fschedule-insns` or at `-O2` or higher.

`-fsched-stalled-insns`
`-fsched-stalled-insns=`*n*

Define how many insns (if any) can be moved prematurely from the queue of stalled insns into the ready list during the second scheduling pass. `-fno-sched-stalled-insns` means that no insns are moved prematurely, `-fsched-stalled-insns=0` means there is no limit on how many queued insns can be moved prematurely. `-fsched-stalled-insns` without a value is equivalent to `-fsched-stalled-insns=1`.

`-fsched-stalled-insns-dep`
`-fsched-stalled-insns-dep=`*n*

Define how many insn groups (cycles) are examined for a dependency on a stalled insn that is a candidate for premature removal from the queue of stalled insns. This has an effect only during the second scheduling pass, and only if `-fsched-stalled-insns` is used. `-fno-sched-stalled-insns-dep` is equivalent to `-fsched-stalled-insns-dep=0`. `-fsched-stalled-insns-dep` without a value is equivalent to `-fsched-stalled-insns-dep=1`.

`-fsched2-use-superblocks`

When scheduling after register allocation, use superblock scheduling. This allows motion across basic block boundaries, resulting in faster schedules. This option is experimental, as not all machine descriptions used by GCC model the CPU closely enough to avoid unreliable results from the algorithm.

This only makes sense when scheduling after register allocation, i.e. with `-fschedule-insns2` or at `-O2` or higher.

`-fsched-group-heuristic`

Enable the group heuristic in the scheduler. This heuristic favors the instruction that belongs to a schedule group. This is enabled by default when scheduling is enabled, i.e. with `-fschedule-insns` or `-fschedule-insns2` or at `-O2` or higher.

`-fsched-critical-path-heuristic`

Enable the critical-path heuristic in the scheduler. This heuristic favors instructions on the critical path. This is enabled by default when scheduling is enabled, i.e. with `-fschedule-insns` or `-fschedule-insns2` or at `-O2` or higher.

`-fsched-spec-insn-heuristic`

Enable the speculative instruction heuristic in the scheduler. This heuristic favors speculative instructions with greater dependency weakness. This is enabled by default when scheduling is enabled, i.e. with `-fschedule-insns` or `-fschedule-insns2` or at `-O2` or higher.

`-fsched-rank-heuristic`

Enable the rank heuristic in the scheduler. This heuristic favors the instruction belonging to a basic block with greater size or frequency. This is enabled by default when scheduling is enabled, i.e. with `-fschedule-insns` or `-fschedule-insns2` or at `-O2` or higher.

`-fsched-last-insn-heuristic`

Enable the last-instruction heuristic in the scheduler. This heuristic favors the instruction that is less dependent on the last instruction scheduled. This is enabled by default when scheduling is enabled, i.e. with `-fschedule-insns` or `-fschedule-insns2` or at `-O2` or higher.

`-fsched-dep-count-heuristic`

Enable the dependent-count heuristic in the scheduler. This heuristic favors the instruction that has more instructions depending on it. This is enabled by default when scheduling is enabled, i.e. with `-fschedule-insns` or `-fschedule-insns2` or at `-O2` or higher.

`-freschedule-modulo-scheduled-loops`

Modulo scheduling is performed before traditional scheduling. If a loop is modulo scheduled, later scheduling passes may change its schedule. Use this option to control that behavior.

`-fselective-scheduling`

Schedule instructions using selective scheduling algorithm. Selective scheduling runs instead of the first scheduler pass.

`-fselective-scheduling2`

Schedule instructions using selective scheduling algorithm. Selective scheduling runs instead of the second scheduler pass.

`-fsel-sched-pipelining`

Enable software pipelining of innermost loops during selective scheduling. This option has no effect unless one of `-fselective-scheduling` or `-fselective-scheduling2` is turned on.

`-fsel-sched-pipelining-outer-loops`

When pipelining loops during selective scheduling, also pipeline outer loops. This option has no effect unless `-fsel-sched-pipelining` is turned on.

`-fsemantic-interposition`

Some object formats, like ELF, allow interposing of symbols by the dynamic linker. This means that for symbols exported from the DSO, the compiler cannot perform interprocedural propagation, inlining and other optimizations in anticipation that the function or variable in question may change. While this feature is useful, for example, to rewrite memory allocation functions by a debugging implementation, it is expensive in the terms of code quality. With `-fno-semantic-interposition` the compiler assumes that if interposition happens for functions the overwriting function will have precisely the same semantics (and side effects). Similarly if interposition happens for variables, the constructor of the variable will be the same. The flag has no effect for functions explicitly declared inline (where it is never allowed for interposition to change semantics) and for symbols explicitly declared weak.

`-fshrink-wrap`

Emit function prologues only before parts of the function that need it, rather than at the top of the function. This flag is enabled by default at `-O` and higher.

`-fshrink-wrap-separate`

Shrink-wrap separate parts of the prologue and epilogue separately, so that those parts are only executed when needed. This option is on by default, but has no effect unless `-fshrink-wrap` is also turned on and the target supports this.

`-fcaller-saves`

Enable allocation of values to registers that are clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code.

This option is always enabled by default on certain machines, usually those which have no call-preserved registers to use instead.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-fcombine-stack-adjustments`

Tracks stack adjustments (pushes and pops) and stack memory references and then tries to find ways to combine them.

Enabled by default at `-O1` and higher.

`-fipa-ra`

Use caller save registers for allocation if those registers are not used by any called function. In that case it is not necessary to save and restore them around calls. This is only possible if called functions are part of same compilation unit as current function and they are compiled before it.

Enabled at levels `-O2`, `-O3`, `-Os`, however the option is disabled if generated code will be instrumented for profiling (`-p`, or `-pg`) or if callee's register usage cannot be known exactly (this happens on targets that do not expose prologues and epilogues in RTL).

`-fconserve-stack`

Attempt to minimize stack usage. The compiler attempts to use less stack space, even if that makes the program slower. This option implies setting the `large-stack-frame` parameter to 100 and the `large-stack-frame-growth` parameter to 400.

`-ftree-reassoc`

Perform reassociation on trees. This flag is enabled by default at `-O1` and higher.

`-fcode-hoisting`

Perform code hoisting. Code hoisting tries to move the evaluation of expressions executed on all paths to the function exit as early as possible. This is especially useful as a code size optimization, but it often helps for code speed as well. This flag is enabled by default at `-O2` and higher.

`-ftree-pre`

Perform partial redundancy elimination (PRE) on trees. This flag is enabled by default at `-O2` and `-O3`.

`-ftree-partial-pre`

Make partial redundancy elimination (PRE) more aggressive. This flag is enabled by default at `-O3`.

`-ftree-forwprop`

Perform forward propagation on trees. This flag is enabled by default at `-O1` and higher.

`-ftree-fre`

Perform full redundancy elimination (FRE) on trees. The difference between FRE and PRE is that FRE only considers expressions that are computed on all paths leading to the redundant computation. This analysis is faster than PRE, though it exposes fewer redundancies. This flag is enabled by default at `-O1` and higher.

`-ftree-phiprop`

Perform hoisting of loads from conditional pointers on trees. This pass is enabled by default at `-O1` and higher.

`-fhoist-adjacent-loads`

Speculatively hoist loads from both branches of an if-then-else if the loads are from adjacent locations in the same structure and the target architecture has a conditional move instruction. This flag is enabled by default at `-O2` and higher.

`-ftree-copy-prop`

Perform copy propagation on trees. This pass eliminates unnecessary copy operations. This flag is enabled by default at `-O1` and higher.

`-fipa-pure-const`

Discover which functions are pure or constant. Enabled by default at `-O1` and higher.

`-fipa-reference`

Discover which static variables do not escape the compilation unit. Enabled by default at `-O1` and higher.

`-fipa-reference-addressable`

Discover read-only, write-only and non-addressable static variables. Enabled by default at `-O1` and higher.

`-fipa-stack-alignment`

Reduce stack alignment on call sites if possible. Enabled by default.

`-fipa-pta`

Perform interprocedural pointer analysis and interprocedural modification and reference analysis. This option can cause excessive memory and compile-time usage on large compilation units. It is not enabled by default at any optimization level.

`-fipa-profile`

Perform interprocedural profile propagation. The functions called only from cold functions are marked as cold. Also functions executed once (such as `cold`, `noreturn`, static constructors or destructors) are identified. Cold functions and loop less parts of functions executed once are then optimized for size. Enabled by default at `-O1` and higher.

`-fipa-modref`

Perform interprocedural mod/ref analysis. This optimization analyzes the side effects of functions (memory locations that are modified or referenced) and enables better optimization across the function call boundary. This flag is enabled by default at `-O1` and higher.

`-fipa-cp`

Perform interprocedural constant propagation. This optimization analyzes the program to determine when values passed to functions are constants and then optimizes accordingly. This optimization can substantially increase performance if the application has constants passed to functions. This flag is enabled by default at `-O2`, `-Os` and `-O3`. It is also enabled by `-fprofile-use` and `-fauto-profile`.

`-fipa-cp-clone`

Perform function cloning to make interprocedural constant propagation stronger. When enabled, interprocedural constant propagation performs function cloning when externally visible function can be called with constant arguments. Because this optimization can create multiple copies of functions, it may significantly increase code size (see `--param ipa-cp-unit-growth=`*`value`*). This flag is enabled by default at `-O3`. It is also enabled by `-fprofile-use` and `-fauto-profile`.

`-fipa-bit-cp`

When enabled, perform interprocedural bitwise constant propagation. This flag is enabled by default at `-O2` and by `-fprofile-use` and `-fauto-profile`. It requires that `-fipa-cp` is enabled.

`-fipa-vrp`

When enabled, perform interprocedural propagation of value ranges. This flag is enabled by default at `-O2`. It requires that `-fipa-cp` is enabled.

`-fipa-icf`

Perform Identical Code Folding for functions and read-only variables. The optimization reduces code size and may disturb unwind stacks by replacing a function by equivalent one with a different name. The optimization works more effectively with link-time optimization enabled.

Although the behavior is similar to the Gold Linker's ICF optimization, GCC ICF works on different levels and thus the optimizations are not same - there are equivalences that are found only by GCC and equivalences found only by Gold.

This flag is enabled by default at `-O2` and `-Os`.

`-flive-patching=`*`level`*

Control GCC's optimizations to produce output suitable for live-patching.

If the compiler's optimization uses a function's body or information extracted from its body to optimize/change another function, the latter is called an impacted function of the former. If a function is patched, its impacted functions should be patched too.

The impacted functions are determined by the compiler's interprocedural optimizations. For example, a caller is impacted when inlining a function into its caller, cloning a function and changing its caller to call this new clone, or extracting a function’s pureness/constness information to optimize its direct or indirect callers, etc.

Usually, the more IPA optimizations enabled, the larger the number of impacted functions for each function. In order to control the number of impacted functions and more easily compute the list of impacted function, IPA optimizations can be partially enabled at two different levels.

The *level* argument should be one of the following:

‘`inline-clone`’

Only enable inlining and cloning optimizations, which includes inlining, cloning, interprocedural scalar replacement of aggregates and partial inlining. As a result, when patching a function, all its callers and its clones’ callers are impacted, therefore need to be patched as well.

`-flive-patching=inline-clone` disables the following optimization flags:

```
-fwhole-program  -fipa-pta  -fipa-reference  -fipa-ra
-fipa-icf  -fipa-icf-functions  -fipa-icf-variables
-fipa-bit-cp  -fipa-vrp  -fipa-pure-const  -fipa-reference-addressable
-fipa-stack-alignment -fipa-modref
```

‘`inline-only-static`’

Only enable inlining of static functions. As a result, when patching a static function, all its callers are impacted and so need to be patched as well.

In addition to all the flags that `-flive-patching=inline-clone` disables, `-flive-patching=inline-only-static` disables the following additional optimization flags:

```
-fipa-cp-clone  -fipa-sra  -fpartial-inlining  -fipa-cp
```

When `-flive-patching` is specified without any value, the default value is *inline-clone*.

This flag is disabled by default.

Note that `-flive-patching` is not supported with link-time optimization (`-flto`).

`-fisolate-erroneous-paths-dereference`

Detect paths that trigger erroneous or undefined behavior due to dereferencing a null pointer. Isolate those paths from the main control flow and turn the statement with erroneous or undefined behavior into a trap. This flag is enabled by default at `-O2` and higher and depends on `-fdelete-null-pointer-checks` also being enabled.

`-fisolate-erroneous-paths-attribute`

Detect paths that trigger erroneous or undefined behavior due to a null value being used in a way forbidden by a `returns_nonnull` or `nonnull` attribute. Isolate those paths from the main control flow and turn the statement with erroneous or undefined behavior into a trap. This is not currently enabled, but may be enabled by `-O2` in the future.

`-ftree-sink`

Perform forward store motion on trees. This flag is enabled by default at `-O1` and higher.

`-ftree-bit-ccp`

Perform sparse conditional bit constant propagation on trees and propagate pointer alignment information. This pass only operates on local scalar variables and is enabled by default at `-O1` and higher, except for `-Og`. It requires that `-ftree-ccp` is enabled.

`-ftree-ccp`

Perform sparse conditional constant propagation (CCP) on trees. This pass only operates on local scalar variables and is enabled by default at `-O1` and higher.

`-fssa-backprop`

Propagate information about uses of a value up the definition chain in order to simplify the definitions. For example, this pass strips sign operations if the sign of a value never matters. The flag is enabled by default at `-O1` and higher.

`-fssa-phiopt`

Perform pattern matching on SSA PHI nodes to optimize conditional code. This pass is enabled by default at `-O1` and higher, except for `-Og`.

`-ftree-switch-conversion`

Perform conversion of simple initializations in a switch to initializations from a scalar array. This flag is enabled by default at `-O2` and higher.

`-ftree-tail-merge`

Look for identical code sequences. When found, replace one with a jump to the other. This optimization is known as tail merging or cross jumping. This flag is enabled by default at `-O2` and higher. The compilation time in this pass can be limited using `max-tail-merge-comparisons` parameter and `max-tail-merge-iterations` parameter.

`-ftree-dce`

Perform dead code elimination (DCE) on trees. This flag is enabled by default at `-O1` and higher.

`-ftree-builtin-call-dce`

Perform conditional dead code elimination (DCE) for calls to built-in functions that may set `errno` but are otherwise free of side effects. This flag is enabled by default at `-O2` and higher if `-Os` is not also specified.

`-ffinite-loops`

Assume that a loop with an exit will eventually take the exit and not loop indefinitely. This allows the compiler to remove loops that otherwise have no side-effects, not considering eventual endless looping as such.

This option is enabled by default at `-O2` for C++ with -std=c++11 or higher.

`-ftree-dominator-opts`

Perform a variety of simple scalar cleanups (constant/copy propagation, redundancy elimination, range propagation and expression simplification) based on a dominator tree traversal. This also performs jump threading (to reduce jumps to jumps). This flag is enabled by default at `-O1` and higher.

`-ftree-dse`

Perform dead store elimination (DSE) on trees. A dead store is a store into a memory location that is later overwritten by another store without any intervening loads. In this case the earlier store can be deleted. This flag is enabled by default at `-O1` and higher.

`-ftree-ch`

Perform loop header copying on trees. This is beneficial since it increases effectiveness of code motion optimizations. It also saves one jump. This flag is enabled by default at `-O1` and higher. It is not enabled for `-Os`, since it usually increases code size.

`-ftree-loop-optimize`

Perform loop optimizations on trees. This flag is enabled by default at `-O1` and higher.

`-ftree-loop-linear`
`-floop-strip-mine`
`-floop-block`

Perform loop nest optimizations. Same as `-floop-nest-optimize`. To use this code transformation, GCC has to be configured with `--with-isl` to enable the Graphite loop transformation infrastructure.

`-fgraphite-identity`

Enable the identity transformation for graphite. For every SCoP we generate the polyhedral representation and transform it back to gimple. Using `-fgraphite-identity` we can check the costs or benefits of the GIMPLE -> GRAPHITE -> GIMPLE transformation. Some minimal optimizations are also performed by the code generator isl, like index splitting and dead code elimination in loops.

`-floop-nest-optimize`

Enable the isl based loop nest optimizer. This is a generic loop nest optimizer based on the Pluto optimization algorithms. It calculates a loop structure optimized for data-locality and parallelism. This option is experimental.

`-floop-parallelize-all`

Use the Graphite data dependence analysis to identify loops that can be parallelized. Parallelize all the loops that can be analyzed to not contain loop carried dependences without checking that it is profitable to parallelize the loops.

`-ftree-coalesce-vars`

While transforming the program out of the SSA representation, attempt to reduce copying by coalescing versions of different user-defined variables, instead of just compiler temporaries. This may severely limit the ability to debug an optimized program compiled with `-fno-var-tracking-assignments`. In the negated form, this flag prevents SSA coalescing of user variables. This option is enabled by default if optimization is enabled, and it does very little otherwise.

`-ftree-loop-if-convert`

Attempt to transform conditional jumps in the innermost loops to branch-less equivalents. The intent is to remove control-flow from the innermost loops in order to improve the ability of the vectorization pass to handle these loops. This is enabled by default if vectorization is enabled.

`-ftree-loop-distribution`

Perform loop distribution. This flag can improve cache performance on big loop bodies and allow further loop optimizations, like parallelization or vectorization, to take place. For example, the loop

```
DO I = 1, N
  A(I) = B(I) + C
  D(I) = E(I) * F
ENDDO
```

is transformed to

```
DO I = 1, N
   A(I) = B(I) + C
ENDDO
DO I = 1, N
   D(I) = E(I) * F
ENDDO
```

This flag is enabled by default at `-O3`. It is also enabled by `-fprofile-use` and `-fauto-profile`.

`-ftree-loop-distribute-patterns`

Perform loop distribution of patterns that can be code generated with calls to a library. This flag is enabled by default at `-O2` and higher, and by `-fprofile-use` and `-fauto-profile`.

This pass distributes the initialization loops and generates a call to memset zero. For example, the loop

```
DO I = 1, N
  A(I) = 0
  B(I) = A(I) + I
ENDDO
```

is transformed to

```
DO I = 1, N
   A(I) = 0
ENDDO
DO I = 1, N
   B(I) = A(I) + I
ENDDO
```

and the initialization loop is transformed into a call to memset zero. This flag is enabled by default at `-O3`. It is also enabled by `-fprofile-use` and `-fauto-profile`.

`-floop-interchange`

Perform loop interchange outside of graphite. This flag can improve cache performance on loop nest and allow further loop optimizations, like vectorization, to take place. For example, the loop

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

is transformed to

```
for (int i = 0; i < N; i++)
  for (int k = 0; k < N; k++)
    for (int j = 0; j < N; j++)
      c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

This flag is enabled by default at `-O3`. It is also enabled by `-fprofile-use` and `-fauto-profile`.

`-floop-unroll-and-jam`

Apply unroll and jam transformations on feasible loops. In a loop nest this unrolls the outer loop by some factor and fuses the resulting multiple inner loops. This flag is enabled by default at `-O3`. It is also enabled by `-fprofile-use` and `-fauto-profile`.

`-ftree-loop-im`

Perform loop invariant motion on trees. This pass moves only invariants that are hard to handle at RTL level (function calls, operations that expand to nontrivial sequences of insns). With `-funswitch-loops` it also moves operands of conditions that are invariant out of the loop, so that we can use just trivial invariantness analysis in loop unswitching. The pass also includes store motion.

`-ftree-loop-ivcanon`

Create a canonical counter for number of iterations in loops for which determining number of iterations requires complicated analysis. Later optimizations then may determine the number easily. Useful especially in connection with unrolling.

`-ftree-scev-cprop`

Perform final value replacement. If a variable is modified in a loop in such a way that its value when exiting the loop can be determined using only its initial value and the number of loop iterations, replace uses of the final value by such a computation, provided it is sufficiently cheap. This reduces data dependencies and may allow further simplifications. Enabled by default at `-O1` and higher.

`-fivopts`

Perform induction variable optimizations (strength reduction, induction variable merging and induction variable elimination) on trees.

`-ftree-parallelize-loops=n`

Parallelize loops, i.e., split their iteration space to run in n threads. This is only possible for loops whose iterations are independent and can be arbitrarily reordered. The optimization is only profitable on multiprocessor machines, for loops that are CPU-intensive, rather than constrained e.g. by memory bandwidth. This option implies `-pthread`, and thus is only supported on targets that have support for `-pthread`.

`-ftree-pta`

Perform function-local points-to analysis on trees. This flag is enabled by default at `-O1` and higher, except for `-Og`.

`-ftree-sra`

Perform scalar replacement of aggregates. This pass replaces structure references with scalars to prevent committing structures to memory too early. This flag is enabled by default at `-O1` and higher, except for `-Og`.

`-fstore-merging`

Perform merging of narrow stores to consecutive memory addresses. This pass merges contiguous stores of immediate values narrower than a word into fewer wider stores to reduce the number of instructions. This is enabled by default at `-O2` and higher as well as `-Os`.

`-ftree-ter`

Perform temporary expression replacement during the SSA->normal phase. Single use/single def temporaries are replaced at their use location with their defining expression. This results in non-GIMPLE code, but gives the expanders much more complex trees to work on resulting in better RTL generation. This is enabled by default at `-O1` and higher.

`-ftree-slsr`

Perform straight-line strength reduction on trees. This recognizes related expressions involving multiplications and replaces them by less expensive calculations when possible. This is enabled by default at `-O1` and higher.

`-ftree-vectorize`

Perform vectorization on trees. This flag enables `-ftree-loop-vectorize` and `-ftree-slp-vectorize` if not explicitly specified.

`-ftree-loop-vectorize`

Perform loop vectorization on trees. This flag is enabled by default at `-O2` and by `-ftree-vectorize`, `-fprofile-use`, and `-fauto-profile`.

`-ftree-slp-vectorize`

Perform basic block vectorization on trees. This flag is enabled by default at `-O2` and by `-ftree-vectorize`, `-fprofile-use`, and `-fauto-profile`.

`-ftrivial-auto-var-init=`*`choice`*

Initialize automatic variables with either a pattern or with zeroes to increase the security and predictability of a program by preventing uninitialized memory disclosure and use. GCC still considers an automatic variable that doesn't have an explicit initializer as uninitialized, `-Wuninitialized` and `-Wanalyzer-use-of-uninitialized-value` will still report warning messages on such automatic variables and the compiler will perform optimization as if the variable were uninitialized. With this option, GCC will also initialize any padding of automatic variables that have structure or union types to zeroes. However, the current implementation cannot initialize automatic variables that are declared between the controlling expression and the first case of a `switch` statement. Using `-Wtrivial-auto-var-init` to report all such cases.

The three values of *choice* are:

- '`uninitialized`' doesn't initialize any automatic variables. This is C and C++'s default.
- '`pattern`' Initialize automatic variables with values which will likely transform logic bugs into crashes down the line, are easily recognized in a crash dump and without being values that programmers can rely on for useful program semantics. The current value is byte-repeatable pattern with byte "0xFE". The values used for pattern initialization might be changed in the future.
- '`zero`' Initialize automatic variables with zeroes.

The default is '`uninitialized`'.

You can control this behavior for a specific variable by using the variable attribute `uninitialized` (see Variable Attributes).

`-fvect-cost-model=`*`model`*

Alter the cost model used for vectorization. The *model* argument should be one of '`unlimited`', '`dynamic`', '`cheap`' or '`very-cheap`'. With the '`unlimited`' model the vectorized code-path is assumed to be profitable while with the '`dynamic`' model a runtime check guards the vectorized code-path to enable it only for iteration counts that will likely execute faster than when executing the original scalar loop. The '`cheap`' model disables vectorization of loops where doing so would be cost prohibitive for example due to required runtime checks for data dependence or alignment but otherwise is equal to the '`dynamic`' model. The '`very-cheap`' model only allows vectorization if the vector code would entirely replace the scalar code that is being vectorized. For example, if each iteration of a vectorized loop would only be able to handle exactly four iterations of the scalar loop, the '`very-cheap`' model would only allow vectorization if the scalar iteration count is known to be a multiple of four.

The default cost model depends on other optimization flags and is either '`dynamic`' or '`cheap`'.

`-fsimd-cost-model=`*`model`*

Alter the cost model used for vectorization of loops marked with the OpenMP simd directive. The *model* argument should be one of '`unlimited`', '`dynamic`', '`cheap`'. All values of *model* have the same meaning as described in `-fvect-cost-model` and by default a cost model defined with `-fvect-cost-model` is used.

`-ftree-vrp`

Perform Value Range Propagation on trees. This is similar to the constant propagation pass, but instead of values, ranges of values are propagated. This allows the optimizers to remove unnecessary range checks like array bound checks and null pointer checks. This is enabled by default at `-O2` and higher. Null pointer check elimination is only done if `-fdelete-null-pointer-checks` is enabled.

`-fsplit-paths`

Split paths leading to loop backedges. This can improve dead code elimination and common subexpression elimination. This is enabled by default at `-O3` and above.

`-fsplit-ivs-in-unroller`

Enables expression of values of induction variables in later iterations of the unrolled loop using the value in the first iteration. This breaks long dependency chains, thus improving efficiency of the scheduling passes.

A combination of `-fweb` and CSE is often sufficient to obtain the same effect. However, that is not reliable in cases where the loop body is more complicated than a single basic block. It also does not work at all on some architectures due to restrictions in the CSE pass.

This optimization is enabled by default.

`-fvariable-expansion-in-unroller`

With this option, the compiler creates multiple copies of some local variables when unrolling a loop, which can result in superior code.

This optimization is enabled by default for PowerPC targets, but disabled by default otherwise.

`-fpartial-inlining`

Inline parts of functions. This option has any effect only when inlining itself is turned on by the `-finline-functions` or `-finline-small-functions` options.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-fpredictive-commoning`

Perform predictive commoning optimization, i.e., reusing computations (especially memory loads and stores) performed in previous iterations of loops.

This option is enabled at level `-O3`. It is also enabled by `-fprofile-use` and `-fauto-profile`.

`-fprefetch-loop-arrays`

If supported by the target machine, generate instructions to prefetch memory to improve the performance of loops that access large arrays.

This option may generate better or worse code; results are highly dependent on the structure of loops within the source code.

Disabled at level `-Os`.

`-fno-printf-return-value`

Do not substitute constants for known return value of formatted output functions such as `sprintf`, `snprintf`, `vsprintf`, and `vsnprintf` (but not `printf` of `fprintf`). This transformation allows GCC to optimize or even eliminate branches based on the known return value of these functions called with arguments that are either constant, or whose values are known to be in a range that makes determining the exact return value possible. For example, when `-fprintf-return-value` is in effect, both the branch and the body of the `if` statement (but not the call to `snprint`) can be optimized away when `i` is a 32-bit or smaller integer because the return value is guaranteed to be at most 8.

```
char buf[9];
if (snprintf (buf, "%08x", i) >= sizeof buf)
  …
```

The `-fprintf-return-value` option relies on other optimizations and yields best results with `-O2` and above. It works in tandem with the `-Wformat-overflow` and `-Wformat-truncation` options. The `-fprintf-return-value` option is enabled by default.

`-fno-peephole`
`-fno-peephole2`

Disable any machine-specific peephole optimizations. The difference between `-fno-peephole` and `-fno-peephole2` is in how they are implemented in the compiler; some targets use one, some use the other, a few use both.

`-fpeephole` is enabled by default. `-fpeephole2` enabled at levels `-O2`, `-O3`, `-Os`.

`-fno-guess-branch-probability`

Do not guess branch probabilities using heuristics.

GCC uses heuristics to guess branch probabilities if they are not provided by profiling feedback (`-fprofile-arcs`). These heuristics are based on the control flow graph. If some branch probabilities are specified by `__builtin_expect`, then the heuristics are used to guess branch probabilities for the rest of the control flow graph, taking the `__builtin_expect` info into account. The interactions between the heuristics and `__builtin_expect` can be complex, and in some cases, it may be useful to disable the heuristics so that the effects of `__builtin_expect` are easier to understand.

It is also possible to specify expected probability of the expression with `__builtin_expect_with_probability` built-in function.

The default is `-fguess-branch-probability` at levels `-O`, `-O2`, `-O3`, `-Os`.

`-freorder-blocks`

Reorder basic blocks in the compiled function in order to reduce number of taken branches and improve code locality.

Enabled at levels `-O1`, `-O2`, `-O3`, `-Os`.

`-freorder-blocks-algorithm=`*`algorithm`*

Use the specified algorithm for basic block reordering. The *algorithm* argument can be ‘`simple`’, which does not increase code size (except sometimes due to secondary effects like alignment), or ‘`stc`’, the “software trace cache” algorithm, which tries to put all often executed code together, minimizing the number of branches executed by making extra copies of code.

The default is ‘`simple`’ at levels `-O1`, `-Os`, and ‘`stc`’ at levels `-O2`, `-O3`.

`-freorder-blocks-and-partition`

In addition to reordering basic blocks in the compiled function, in order to reduce number of taken branches, partitions hot and cold basic blocks into separate sections of the assembly and `.o` files, to improve paging and cache locality performance.

This optimization is automatically turned off in the presence of exception handling or unwind tables (on targets using setjump/longjump or target specific scheme), for linkonce sections, for functions with a user-defined section attribute and on any architecture that does not support named sections. When `-fsplit-stack` is used this option is not enabled by default (to avoid linker errors), but may be enabled explicitly (if using a working linker).

Enabled for x86 at levels `-O2`, `-O3`, `-Os`.

`-freorder-functions`

Reorder functions in the object file in order to improve code locality. This is implemented by using special subsections `.text.hot` for most frequently executed functions and `.text.unlikely` for unlikely executed functions. Reordering is done by the linker so object file format must support named sections and linker must place them in a reasonable way.

This option isn’t effective unless you either provide profile feedback (see `-fprofile-arcs` for details) or manually annotate functions with `hot` or `cold` attributes (see Common Function Attributes).

Enabled at levels `-O2`, `-O3`, `-Os`.

`-fstrict-aliasing`

Allow the compiler to assume the strictest aliasing rules applicable to the language being compiled. For C (and C++), this activates optimizations based on the type of expressions. In particular, an object of one type is assumed never to reside at the same address as an object of a different type, unless the types are almost the same. For example, an `unsigned int` can alias an `int`, but not a `void*` or a `double`. A character type may alias any other type.

Pay special attention to code like this:

```
union a_union {
  int i;
  double d;
};

int f() {
  union a_union t;
  t.d = 3.0;
  return t.i;
}
```

The practice of reading from a different union member than the one most recently written to (called “type-punning”) is common. Even with `-fstrict-aliasing`, type-punning is allowed, provided the memory is accessed through the union type. So, the code above works as expected. See Structures unions enumerations and bit-fields implementation. However, this code might not:

```
int f() {
  union a_union t;
  int* ip;
  t.d = 3.0;
  ip = &t.i;
  return *ip;
}
```

Similarly, access by taking the address, casting the resulting pointer and dereferencing the result has undefined behavior, even if the cast uses a union type, e.g.:

```
int f() {
  double d = 3.0;
  return ((union a_union *) &d)->i;
}
```

The `-fstrict-aliasing` option is enabled at levels `-O2`, `-O3`, `-Os`.

`-fipa-strict-aliasing`

Controls whether rules of `-fstrict-aliasing` are applied across function boundaries. Note that if multiple functions gets inlined into a single function the memory accesses are no longer considered to be crossing a function boundary.

The `-fipa-strict-aliasing` option is enabled by default and is effective only in combination with `-fstrict-aliasing`.

`-falign-functions`
`-falign-functions=`*`n`*
`-falign-functions=`*`n`*`:`*`m`*
`-falign-functions=`*`n`*`:`*`m`*`:`*`n2`*
`-falign-functions=`*`n`*`:`*`m`*`:`*`n2`*`:`*`m2`*

Align the start of functions to the next power-of-two greater than or equal to *n*, skipping up to *m*-1 bytes. This ensures that at least the first *m* bytes of the function can be fetched by the CPU without crossing an *n*-byte alignment boundary.

If *m* is not specified, it defaults to *n*.

Examples: `-falign-functions=32` aligns functions to the next 32-byte boundary, `-falign-functions=24` aligns to the next 32-byte boundary only if this can be done by skipping 23 bytes or less, `-falign-functions=32:7` aligns to the next 32-byte boundary only if this can be done by skipping 6 bytes or less.

The second pair of *n2*:*m2* values allows you to specify a secondary alignment: `-falign-functions=64:7:32:3` aligns to the next 64-byte boundary if this can be done by skipping 6 bytes or less, otherwise aligns to the next 32-byte boundary if this can be done by skipping 2 bytes or less. If *m2* is not specified, it defaults to *n2*.

Some assemblers only support this flag when *n* is a power of two; in that case, it is rounded up.

`-fno-align-functions` and `-falign-functions=1` are equivalent and mean that functions are not aligned.

If *n* is not specified or is zero, use a machine-dependent default. The maximum allowed *n* option value is 65536.

Enabled at levels `-O2`, `-O3`.

`-flimit-function-alignment`

If this option is enabled, the compiler tries to avoid unnecessarily overaligning functions. It attempts to instruct the assembler to align by the amount specified by `-falign-functions`, but not to skip more bytes than the size of the function.

`-falign-labels`
`-falign-labels=`*`n`*
`-falign-labels=`*`n`*`:`*`m`*
`-falign-labels=`*`n`*`:`*`m`*`:`*`n2`*

`-falign-labels=`*n*`:`*m*`:`*n2*`:`*m2*

Align all branch targets to a power-of-two boundary.

Parameters of this option are analogous to the `-falign-functions` option. `-fno-align-labels` and `-falign-labels=1` are equivalent and mean that labels are not aligned.

If `-falign-loops` or `-falign-jumps` are applicable and are greater than this value, then their values are used instead.

If *n* is not specified or is zero, use a machine-dependent default which is very likely to be '`1`', meaning no alignment. The maximum allowed *n* option value is 65536.

Enabled at levels `-O2`, `-O3`.

`-falign-loops`
`-falign-loops=`*n*
`-falign-loops=`*n*`:`*m*
`-falign-loops=`*n*`:`*m*`:`*n2*
`-falign-loops=`*n*`:`*m*`:`*n2*`:`*m2*

Align loops to a power-of-two boundary. If the loops are executed many times, this makes up for any execution of the dummy padding instructions.

If `-falign-labels` is greater than this value, then its value is used instead.

Parameters of this option are analogous to the `-falign-functions` option. `-fno-align-loops` and `-falign-loops=1` are equivalent and mean that loops are not aligned. The maximum allowed *n* option value is 65536.

If *n* is not specified or is zero, use a machine-dependent default.

Enabled at levels `-O2`, `-O3`.

`-falign-jumps`
`-falign-jumps=`*n*
`-falign-jumps=`*n*`:`*m*
`-falign-jumps=`*n*`:`*m*`:`*n2*
`-falign-jumps=`*n*`:`*m*`:`*n2*`:`*m2*

Align branch targets to a power-of-two boundary, for branch targets where the targets can only be reached by jumping. In this case, no dummy operations need be executed.

If `-falign-labels` is greater than this value, then its value is used instead.

Parameters of this option are analogous to the `-falign-functions` option. `-fno-align-jumps` and `-falign-jumps=1` are equivalent and mean that loops are not aligned.

If *n* is not specified or is zero, use a machine-dependent default. The maximum allowed *n* option value is 65536.

Enabled at levels `-O2`, `-O3`.

`-fno-allocation-dce`

Do not remove unused C++ allocations in dead code elimination.

`-fallow-store-data-races`

Allow the compiler to perform optimizations that may introduce new data races on stores, without proving that the variable cannot be concurrently accessed by other threads. Does not affect optimization of local data. It is safe to use this option if it is known that global data will not be accessed by multiple threads.

Examples of optimizations enabled by `-fallow-store-data-races` include hoisting or if-conversions that may cause a value that was already in memory to be re-written with that same value. Such re-writing is safe in a single threaded context but may be unsafe in a multi-threaded context. Note that on some processors, if-conversions may be required in order to enable vectorization.

Enabled at level `-Ofast`.

`-funit-at-a-time`

This option is left for compatibility reasons. `-funit-at-a-time` has no effect, while `-fno-unit-at-a-time` implies `-fno-toplevel-reorder` and `-fno-section-anchors`.

Enabled by default.

`-fno-toplevel-reorder`

Do not reorder top-level functions, variables, and `asm` statements. Output them in the same order that they appear in the input file. When this option is used, unreferenced static variables are not removed. This option is intended to support existing code that relies on a particular ordering. For new code, it is better to use attributes when possible.

`-ftoplevel-reorder` is the default at `-O1` and higher, and also at `-O0` if `-fsection-anchors` is explicitly requested. Additionally `-fno-toplevel-reorder` implies `-fno-section-anchors`.

`-funreachable-traps`

With this option, the compiler turns calls to `__builtin_unreachable` into traps, instead of using them for optimization. This also affects any such calls implicitly generated by the compiler.

This option has the same effect as `-fsanitize=unreachable -fsanitize-trap=unreachable`, but does not affect the values of those options. If `-fsanitize=unreachable` is enabled, that option takes priority over this one.

This option is enabled by default at `-O0` and `-Og`.

`-fweb`

Constructs webs as commonly used for register allocation purposes and assign each web individual pseudo register. This allows the register allocation pass to operate on pseudos directly, but also strengthens several other optimization passes, such as CSE, loop optimizer and trivial dead code remover. It can, however, make debugging impossible, since variables no longer stay in a “home register”.

Enabled by default with `-funroll-loops`.

`-fwhole-program`

Assume that the current compilation unit represents the whole program being compiled. All public functions and variables with the exception of `main` and those merged by attribute `externally_visible` become static functions and in effect are optimized more aggressively by interprocedural optimizers.

With `-flto` this option has a limited use. In most cases the precise list of symbols used or exported from the binary is known the resolution info passed to the link-time optimizer by the linker plugin. It is still useful if no linker plugin is used or during incremental link step when final code is produced (with `-flto -flinker-output=nolto-rel`).

`-flto[=`*`n`*`]`

This option runs the standard link-time optimizer. When invoked with source code, it generates GIMPLE (one of GCC's internal representations) and writes it to special ELF sections in the object file. When the object files are linked together, all the function bodies are read from these ELF sections and instantiated as if they had been part of the same translation unit.

To use the link-time optimizer, `-flto` and optimization options should be specified at compile time and during the final link. It is recommended that you compile all the files participating in the same link with the same options and also specify those options at link time. For example:

```
gcc -c -O2 -flto foo.c
gcc -c -O2 -flto bar.c
gcc -o myprog -flto -O2 foo.o bar.o
```

The first two invocations to GCC save a bytecode representation of GIMPLE into special ELF sections inside `foo.o` and `bar.o`. The final invocation reads the GIMPLE bytecode from `foo.o` and `bar.o`, merges the two files into a single internal image, and compiles the result as usual. Since both `foo.o` and `bar.o` are merged into a single image, this causes all the interprocedural analyses and optimizations in GCC to work across the two files as if they were a single one. This means, for example, that the inliner is able to inline functions in `bar.o` into functions in `foo.o` and vice-versa.

Another (simpler) way to enable link-time optimization is:

```
gcc -o myprog -flto -O2 foo.c bar.c
```

The above generates bytecode for `foo.c` and `bar.c`, merges them together into a single GIMPLE representation and optimizes them as usual to produce `myprog`.

The important thing to keep in mind is that to enable link-time optimizations you need to use the GCC driver to perform the link step. GCC automatically performs link-time optimization if any of the objects involved were compiled with the `-flto` command-line option. You can always override the automatic decision to do link-time optimization by passing `-fno-lto` to the link command.

To make whole program optimization effective, it is necessary to make certain whole program assumptions. The compiler needs to know what functions and variables can be accessed by libraries and runtime outside of the link-time optimized unit. When supported by the linker, the linker plugin (see `-fuse-linker-plugin`) passes information to the compiler about used and externally visible symbols. When the linker plugin is not available, `-fwhole-program` should be used to allow the compiler to make these assumptions, which leads to more aggressive optimization decisions.

When a file is compiled with `-flto` without `-fuse-linker-plugin`, the generated object file is larger than a regular object file because it contains GIMPLE bytecodes and the usual final code (see `-ffat-lto-objects`). This means that object files with LTO information can be linked as normal object files; if `-fno-lto` is passed to the linker, no interprocedural optimizations are applied. Note that when `-fno-fat-lto-objects` is enabled the compile stage is faster but you cannot perform a regular, non-LTO link on them.

When producing the final binary, GCC only applies link-time optimizations to those files that contain bytecode. Therefore, you can mix and match object files and libraries with GIMPLE bytecodes and final object code. GCC automatically selects which files to optimize in LTO mode and which files to link without further processing.

Generally, options specified at link time override those specified at compile time, although in some cases GCC attempts to infer link-time options from the settings used to compile the input files.

If you do not specify an optimization level option `-O` at link time, then GCC uses the highest optimization level used when compiling the object files. Note that it is generally ineffective to specify an optimization level option only at link time and not at compile time, for two reasons. First, compiling without optimization suppresses compiler passes that gather information needed for effective optimization at link time. Second, some early optimization passes can be performed only at compile time and not at link time.

There are some code generation flags preserved by GCC when generating bytecodes, as they need to be used during the final link. Currently, the following options and their settings are taken from the first object file that explicitly specifies them: `-fcommon`, `-fexceptions`, `-fnon-call-exceptions`, `-fgnu-tm` and all the `-m` target flags.

The following options `-fPIC`, `-fpic`, `-fpie` and `-fPIE` are combined based on the following scheme:

```
-fPIC + -fpic = -fpic
-fPIC + -fno-pic = -fno-pic
-fpic/-fPIC + (no option) = (no option)
-fPIC + -fPIE = -fPIE
-fpic + -fPIE = -fpie
-fPIC/-fpic + -fpie = -fpie
```

Certain ABI-changing flags are required to match in all compilation units, and trying to override this at link time with a conflicting value is ignored. This includes options such as `-freg-struct-return` and `-fpcc-struct-return`.

Other options such as `-ffp-contract`, `-fno-strict-overflow`, `-fwrapv`, `-fno-trapv` or `-fno-strict-aliasing` are passed through to the link stage and merged conservatively for conflicting translation units. Specifically `-fno-strict-overflow`, `-fwrapv` and `-fno-trapv` take precedence; and for example `-ffp-contract=off` takes precedence over `-ffp-contract=fast`. You can override them at link time.

Diagnostic options such as `-Wstringop-overflow` are passed through to the link stage and their setting matches that of the compile-step at function granularity. Note that this matters only for diagnostics emitted during optimization. Note that code transforms such as inlining can lead to warnings being enabled or disabled for regions if code not consistent with the setting at compile time.

When you need to pass options to the assembler via `-Wa` or `-Xassembler` make sure to either compile such translation units with `-fno-lto` or consistently use the same assembler options on all translation units. You can alternatively also specify assembler options at LTO link time.

To enable debug info generation you need to supply `-g` at compile time. If any of the input files at link time were built with debug info generation enabled the link will enable debug info generation as well. Any elaborate debug info settings like the dwarf level `-gdwarf-5` need to be explicitly repeated at the linker command line and mixing different settings in different translation units is discouraged.

If LTO encounters objects with C linkage declared with incompatible types in separate translation units to be linked together (undefined behavior according to ISO C99 6.2.7), a non-fatal diagnostic may be issued. The behavior is still undefined at run time. Similar diagnostics may be raised for other languages.

Another feature of LTO is that it is possible to apply interprocedural optimizations on files written in different languages:

```
gcc -c -flto foo.c
g++ -c -flto bar.cc
gfortran -c -flto baz.f90
g++ -o myprog -flto -O3 foo.o bar.o baz.o -lgfortran
```

Notice that the final link is done with `g++` to get the C++ runtime libraries and `-lgfortran` is added to get the Fortran runtime libraries. In general, when mixing languages in LTO mode, you should use the same link command options as when mixing languages in a regular (non-LTO) compilation.

If object files containing GIMPLE bytecode are stored in a library archive, say `libfoo.a`, it is possible to extract and use them in an LTO link if you are using a linker with plugin support. To create static libraries suitable for LTO, use `gcc-ar` and `gcc-ranlib` instead of `ar` and `ranlib`; to show the symbols of object files with GIMPLE bytecode, use `gcc-nm`. Those commands require that `ar`, `ranlib` and `nm` have been compiled with plugin support. At link time, use the flag `-fuse-linker-plugin` to ensure that the library participates in the LTO optimization process:

```
gcc -o myprog -O2 -flto -fuse-linker-plugin a.o b.o -lfoo
```

With the linker plugin enabled, the linker extracts the needed GIMPLE files from `libfoo.a` and passes them on to the running GCC to make them part of the aggregated GIMPLE image to be optimized.

If you are not using a linker with plugin support and/or do not enable the linker plugin, then the objects inside `libfoo.a` are extracted and linked as usual, but they do not participate in the LTO optimization process. In order to make a static library suitable for both LTO optimization and usual linkage, compile its object files with `-flto -ffat-lto-objects`.

Link-time optimizations do not require the presence of the whole program to operate. If the program does not require any symbols to be exported, it is possible to combine `-flto` and `-fwhole-program` to allow the interprocedural optimizers to use more aggressive assumptions which may lead to improved optimization opportunities. Use of `-fwhole-program` is not needed when linker plugin is active (see `-fuse-linker-plugin`).

The current implementation of LTO makes no attempt to generate bytecode that is portable between different types of hosts. The bytecode files are versioned and there is a strict version check, so bytecode files generated in one version of GCC do not work with an older or newer version of GCC.

Link-time optimization does not work well with generation of debugging information on systems other than those using a combination of ELF and DWARF.

If you specify the optional *n*, the optimization and code generation done at link time is executed in parallel using *n* parallel jobs by utilizing an installed `make` program. The environment variable `MAKE` may be used to override the program used.

You can also specify `-flto=jobserver` to use GNU make’s job server mode to determine the number of parallel jobs. This is useful when the Makefile calling GCC is already executing in parallel. You must prepend a ‘`+`’ to the command recipe in the parent Makefile for this to work. This option likely only works if `MAKE` is GNU make. Even without the option value, GCC tries to automatically detect a running GNU make’s job server.

Use `-flto=auto` to use GNU make’s job server, if available, or otherwise fall back to autodetection of the number of CPU threads present in your system.

`-flto-partition=`*`alg`*

Specify the partitioning algorithm used by the link-time optimizer. The value is either ‘`1to1`’ to specify a partitioning mirroring the original source files or ‘`balanced`’ to specify partitioning into equally sized chunks (whenever possible) or ‘`max`’ to create new partition for every symbol where possible. Specifying ‘`none`’ as an algorithm disables partitioning and streaming completely. The default value is ‘`balanced`’. While ‘`1to1`’ can be used as an workaround for various code ordering issues, the ‘`max`’ partitioning is intended for internal testing only. The value ‘`one`’ specifies that exactly one partition should be used while the value ‘`none`’ bypasses partitioning and executes the link-time optimization step directly from the WPA phase.

`-flto-compression-level=`*`n`*

This option specifies the level of compression used for intermediate language written to LTO object files, and is only meaningful in conjunction with LTO mode (`-flto`). GCC currently supports two LTO compression algorithms. For zstd, valid values are 0 (no compression) to 19 (maximum compression), while zlib supports values from 0 to 9. Values outside this range are clamped to either minimum or maximum of the supported values. If the option is not given, a default balanced compression setting is used.

`-fuse-linker-plugin`

Enables the use of a linker plugin during link-time optimization. This option relies on plugin support in the linker, which is available in gold or in GNU ld 2.21 or newer.

This option enables the extraction of object files with GIMPLE bytecode out of library archives. This improves the quality of optimization by exposing more code to the link-time optimizer. This information specifies what symbols can be accessed externally (by non-LTO object or during dynamic linking). Resulting code quality improvements on binaries (and shared libraries that use hidden visibility) are similar to `-fwhole-program`. See `-flto` for a description of the effect of this flag and how to use it.

This option is enabled by default when LTO support in GCC is enabled and GCC was configured for use with a linker supporting plugins (GNU ld 2.21 or newer or gold).

`-ffat-lto-objects`

Fat LTO objects are object files that contain both the intermediate language and the object code. This makes them usable for both LTO linking and normal linking. This option is effective only when compiling with `-flto` and is ignored at link time.

`-fno-fat-lto-objects` improves compilation time over plain LTO, but requires the complete toolchain to be aware of LTO. It requires a linker with linker plugin support for basic functionality. Additionally, `nm`, `ar` and `ranlib` need to support linker plugins to allow a full-featured build environment (capable of building static libraries etc). GCC provides the `gcc-ar`, `gcc-nm`, `gcc-ranlib` wrappers to pass the right options to these tools. With non fat LTO makefiles need to be modified to use them.

Note that modern binutils provide plugin auto-load mechanism. Installing the linker plugin into `$libdir/bfd-plugins` has the same effect as usage of the command wrappers (`gcc-ar`, `gcc-nm` and `gcc-ranlib`).

The default is `-fno-fat-lto-objects` on targets with linker plugin support.

`-fcompare-elim`

After register allocation and post-register allocation instruction splitting, identify arithmetic instructions that compute processor flags similar to a comparison operation based on that arithmetic. If possible, eliminate the explicit comparison operation.

This pass only applies to certain targets that cannot explicitly represent the comparison operation before register allocation is complete.

Enabled at levels `-O1`, `-O2`, `-O3`, `-Os`.

`-fcprop-registers`

After register allocation and post-register allocation instruction splitting, perform a copy-propagation pass to try to reduce scheduling dependencies and occasionally eliminate the copy.

Enabled at levels `-O1`, `-O2`, `-O3`, `-Os`.

`-fprofile-correction`

Profiles collected using an instrumented binary for multi-threaded programs may be inconsistent due to missed counter updates. When this option is specified, GCC uses heuristics to correct or smooth out such inconsistencies. By default, GCC emits an error message when an inconsistent profile is detected.

This option is enabled by `-fauto-profile`.

`-fprofile-partial-training`

With `-fprofile-use` all portions of programs not executed during train run are optimized agressively for size rather than speed. In some cases it is not practical to train all possible hot paths in the program. (For example, program may contain functions specific for a given hardware and trianing may not cover all hardware configurations program is run on.) With `-fprofile-partial-training` profile feedback will be ignored for all functions not executed during the train run leading them to be optimized as if they were compiled without profile feedback. This leads to better performance when train run is not representative but also leads to significantly bigger code.

`-fprofile-use`
`-fprofile-use=`*`path`*

Enable profile feedback-directed optimizations, and the following optimizations, many of which are generally profitable only with profile feedback available:

```
-fbranch-probabilities  -fprofile-values
-funroll-loops  -fpeel-loops  -ftracer  -fvpt
-finline-functions  -fipa-cp  -fipa-cp-clone  -fipa-bit-cp
-fpredictive-commoning  -fsplit-loops  -funswitch-loops
-fgcse-after-reload  -ftree-loop-vectorize  -ftree-slp-vectorize
-fvect-cost-model=dynamic  -ftree-loop-distribute-patterns
-fprofile-reorder-functions
```

Before you can use this option, you must first generate profiling information. See Instrumentation Options, for information about the `-fprofile-generate` option.

By default, GCC emits an error message if the feedback profiles do not match the source code. This error can be turned into a warning by using `-Wno-error=coverage-mismatch`. Note this may result in poorly optimized code. Additionally, by default, GCC also emits a warning message if the feedback profiles do not exist (see `-Wmissing-profile`).

If *path* is specified, GCC looks at the *path* to find the profile feedback data files. See `-fprofile-dir`.

`-fauto-profile`
`-fauto-profile=`*`path`*

Enable sampling-based feedback-directed optimizations, and the following optimizations, many of which are generally profitable only with profile feedback available:

```
-fbranch-probabilities  -fprofile-values
-funroll-loops  -fpeel-loops  -ftracer  -fvpt
-finline-functions  -fipa-cp  -fipa-cp-clone  -fipa-bit-cp
-fpredictive-commoning  -fsplit-loops  -funswitch-loops
-fgcse-after-reload  -ftree-loop-vectorize  -ftree-slp-vectorize
-fvect-cost-model=dynamic  -ftree-loop-distribute-patterns
-fprofile-correction
```

path is the name of a file containing AutoFDO profile information. If omitted, it defaults to `fbdata.afdo` in the current directory.

Producing an AutoFDO profile data file requires running your program with the `perf` utility on a supported GNU/Linux target system. For more information, see https://perf.wiki.kernel.org/.

E.g.

```
perf record -e br_inst_retired:near_taken -b -o perf.data \
    -- your_program
```

Then use the `create_gcov` tool to convert the raw profile data to a format that can be used by GCC. You must also supply the unstripped binary for your program to this tool. See https://github.com/google/autofdo.

E.g.

```
create_gcov --binary=your_program.unstripped --profile=perf.data \
    --gcov=profile.afdo
```

The following options control compiler behavior regarding floating-point arithmetic. These options trade off between speed and correctness. All must be specifically enabled.

`-ffloat-store`

Do not store floating-point variables in registers, and inhibit other options that might change whether a floating-point value is taken from a register or memory.

This option prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a `double` is supposed to have. Similarly for the x86 architecture. For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use `-ffloat-store` for such programs, after modifying them to store all pertinent intermediate computations into variables.

`-fexcess-precision=`*`style`*

This option allows further control over excess precision on machines where floating-point operations occur in a format with more precision or range than the IEEE standard and interchange floating-point types. By default, `-fexcess-precision=fast` is in effect; this means that operations may be carried out in a wider precision than the types specified in the source if that would result in faster code, and it is unpredictable when rounding to the types specified in the source code takes place. When compiling C or C++, if `-fexcess-precision=standard` is specified then excess precision follows the rules specified in ISO C99 or C++; in particular, both casts and assignments cause values to be rounded to their semantic types (whereas `-ffloat-store` only affects assignments). This option is enabled by default for C or C++ if a strict conformance option such as `-std=c99` or `-std=c++17` is used. `-ffast-math` enables `-fexcess-precision=fast` by default regardless of whether a strict conformance option is used.

`-fexcess-precision=standard` is not implemented for languages other than C or C++. On the x86, it has no effect if `-mfpmath=sse` or `-mfpmath=sse+387` is specified; in the former case, IEEE semantics apply without excess precision, and in the latter, rounding is unpredictable.

`-ffast-math`

Sets the options `-fno-math-errno`, `-funsafe-math-optimizations`, `-ffinite-math-only`, `-fno-rounding-math`, `-fno-signaling-nans`, `-fcx-limited-range` and `-fexcess-precision=fast`.

This option causes the preprocessor macro `__FAST_MATH__` to be defined.

This option is not turned on by any `-O` option besides `-Ofast` since it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions. It may, however, yield faster code for programs that do not require the guarantees of these specifications.

`-fno-math-errno`

Do not set `errno` after calling math functions that are executed with a single instruction, e.g., `sqrt`. A program that relies on IEEE exceptions for math error handling may want to use this flag for speed while maintaining IEEE arithmetic compatibility.

This option is not turned on by any `-O` option since it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions. It may, however, yield faster code for programs that do not require the guarantees of these specifications.

The default is `-fmath-errno`.

On Darwin systems, the math library never sets `errno`. There is therefore no reason for the compiler to consider the possibility that it might, and `-fno-math-errno` is the default.

`-funsafe-math-optimizations`

Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid and (b) may violate IEEE or ANSI standards. When used at link time, it may include libraries or startup files that change the default FPU control word or other similar optimizations.

This option is not turned on by any `-O` option since it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions. It may, however, yield faster code for programs that do not require the guarantees of these specifications. Enables `-fno-signed-zeros`, `-fno-trapping-math`, `-fassociative-math` and `-freciprocal-math`.

The default is `-fno-unsafe-math-optimizations`.

`-fassociative-math`

Allow re-association of operands in series of floating-point operations. This violates the ISO C and C++ language standard by possibly changing computation result. NOTE: re-ordering may change the sign of zero as well as ignore NaNs and inhibit or create underflow or overflow (and thus cannot be used on code that relies on rounding behavior like `(x + 2**52) - 2**52`. May also reorder floating-point comparisons and thus may not be used when ordered comparisons are required. This option requires that both `-fno-signed-zeros` and `-fno-trapping-math` be in effect. Moreover, it doesn't make much sense with `-frounding-math`. For Fortran the option is automatically enabled when both `-fno-signed-zeros` and `-fno-trapping-math` are in effect.

The default is `-fno-associative-math`.

`-freciprocal-math`

Allow the reciprocal of a value to be used instead of dividing by the value if this enables optimizations. For example `x / y` can be replaced with `x * (1/y)`, which is useful if `(1/y)` is subject to common subexpression elimination. Note that this loses precision and increases the number of flops operating on the value.

The default is `-fno-reciprocal-math`.

`-ffinite-math-only`

Allow optimizations for floating-point arithmetic that assume that arguments and results are not NaNs or +-Infs.

This option is not turned on by any `-O` option since it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions. It may, however, yield faster code for programs that do not require the guarantees of these specifications.

The default is `-fno-finite-math-only`.

`-fno-signed-zeros`

Allow optimizations for floating-point arithmetic that ignore the signedness of zero. IEEE arithmetic specifies the behavior of distinct +0.0 and -0.0 values, which then prohibits simplification of expressions such as x+0.0 or 0.0*x (even with `-ffinite-math-only`). This option implies that the sign of a zero result isn’t significant.

The default is `-fsigned-zeros`.

`-fno-trapping-math`

Compile code assuming that floating-point operations cannot generate user-visible traps. These traps include division by zero, overflow, underflow, inexact result and invalid operation. This option requires that `-fno-signaling-nans` be in effect. Setting this option may allow faster code if one relies on “non-stop” IEEE arithmetic, for example.

This option should never be turned on by any `-O` option since it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions.

The default is `-ftrapping-math`.

Future versions of GCC may provide finer control of this setting using C99’s `FENV_ACCESS` pragma. This command-line option will be used along with `-frounding-math` to specify the default state for `FENV_ACCESS`.

`-frounding-math`

Disable transformations and optimizations that assume default floating-point rounding behavior. This is round-to-zero for all floating point to integer conversions, and round-to-nearest for all other arithmetic truncations. This option should be specified for programs that change the FP rounding mode dynamically, or that may be executed with a non-default rounding mode. This option disables constant folding of floating-point expressions at compile time (which may be affected by rounding mode) and arithmetic transformations that are unsafe in the presence of sign-dependent rounding modes.

The default is `-fno-rounding-math`.

This option is experimental and does not currently guarantee to disable all GCC optimizations that are affected by rounding mode. Future versions of GCC may provide finer control of this setting using C99’s `FENV_ACCESS` pragma. This command-line option will be used along with `-ftrapping-math` to specify the default state for `FENV_ACCESS`.

`-fsignaling-nans`

Compile code assuming that IEEE signaling NaNs may generate user-visible traps during floating-point operations. Setting this option disables optimizations that may change the number of exceptions visible with signaling NaNs. This option implies `-ftrapping-math`.

This option causes the preprocessor macro `__SUPPORT_SNAN__` to be defined.

The default is `-fno-signaling-nans`.

This option is experimental and does not currently guarantee to disable all GCC optimizations that affect signaling NaN behavior.

`-fno-fp-int-builtin-inexact`

Do not allow the built-in functions `ceil`, `floor`, `round` and `trunc`, and their `float` and `long double` variants, to generate code that raises the “inexact” floating-point exception for noninteger arguments. ISO C99 and C11 allow these functions to raise the “inexact” exception, but ISO/IEC TS 18661-

1:2014, the C bindings to IEEE 754-2008, as integrated into ISO C2X, does not allow these functions to do so.

The default is `-ffp-int-builtin-inexact`, allowing the exception to be raised, unless C2X or a later C standard is selected. This option does nothing unless `-ftrapping-math` is in effect.

Even if `-fno-fp-int-builtin-inexact` is used, if the functions generate a call to a library function then the "inexact" exception may be raised if the library implementation does not follow TS 18661.

`-fsingle-precision-constant`

Treat floating-point constants as single precision instead of implicitly converting them to double-precision constants.

`-fcx-limited-range`

When enabled, this option states that a range reduction step is not needed when performing complex division. Also, there is no checking whether the result of a complex multiplication or division is `NaN + I*NaN`, with an attempt to rescue the situation in that case. The default is `-fno-cx-limited-range`, but is enabled by `-ffast-math`.

This option controls the default setting of the ISO C99 `CX_LIMITED_RANGE` pragma. Nevertheless, the option applies to all languages.

`-fcx-fortran-rules`

Complex multiplication and division follow Fortran rules. Range reduction is done as part of complex division, but there is no checking whether the result of a complex multiplication or division is `NaN + I*NaN`, with an attempt to rescue the situation in that case.

The default is `-fno-cx-fortran-rules`.

The following options control optimizations that may improve performance, but are not enabled by any `-O` options. This section includes experimental options that may produce broken code.

`-fbranch-probabilities`

After running a program compiled with `-fprofile-arcs` (see Instrumentation Options), you can compile it a second time using `-fbranch-probabilities`, to improve optimizations based on the number of times each branch was taken. When a program compiled with `-fprofile-arcs` exits, it saves arc execution counts to a file called `*sourcename*.gcda` for each source file. The information in this data file is very dependent on the structure of the generated code, so you must use the same source code and the same optimization options for both compilations. See details about the file naming in `-fprofile-arcs`.

With `-fbranch-probabilities`, GCC puts a '`REG_BR_PROB`' note on each '`JUMP_INSN`' and '`CALL_INSN`'. These can be used to improve optimization. Currently, they are only used in one place: in `reorg.cc`, instead of guessing which path a branch is most likely to take, the '`REG_BR_PROB`' values are used to exactly determine which path is taken more often.

Enabled by `-fprofile-use` and `-fauto-profile`.

`-fprofile-values`

If combined with `-fprofile-arcs`, it adds code so that some data about values of expressions in the program is gathered.

With `-fbranch-probabilities`, it reads back the data gathered from profiling values of expressions for usage in optimizations.

Enabled by `-fprofile-generate`, `-fprofile-use`, and `-fauto-profile`.

`-fprofile-reorder-functions`

Function reordering based on profile instrumentation collects first time of execution of a function and orders these functions in ascending order.

Enabled with `-fprofile-use`.

`-fvpt`

If combined with `-fprofile-arcs`, this option instructs the compiler to add code to gather information about values of expressions.

With `-fbranch-probabilities`, it reads back the data gathered and actually performs the optimizations based on them. Currently the optimizations include specialization of division operations using the knowledge about the value of the denominator.

Enabled with `-fprofile-use` and `-fauto-profile`.

`-frename-registers`

Attempt to avoid false dependencies in scheduled code by making use of registers left over after register allocation. This optimization most benefits processors with lots of registers. Depending on the debug information format adopted by the target, however, it can make debugging impossible, since variables no longer stay in a “home register”.

Enabled by default with `-funroll-loops`.

`-fschedule-fusion`

Performs a target dependent pass over the instruction stream to schedule instructions of same type together because target machine can execute them more efficiently if they are adjacent to each other in the instruction flow.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-ftracer`

Perform tail duplication to enlarge superblock size. This transformation simplifies the control flow of the function allowing other optimizations to do a better job.

Enabled by `-fprofile-use` and `-fauto-profile`.

`-funroll-loops`

Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. `-funroll-loops` implies `-frerun-cse-after-loop`, `-fweb` and `-frename-registers`. It also turns on complete loop peeling (i.e. complete removal of loops with a small constant number of iterations). This option makes code larger, and may or may not make it run faster.

Enabled by `-fprofile-use` and `-fauto-profile`.

`-funroll-all-loops`

Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This usually makes programs run more slowly. `-funroll-all-loops` implies the same options as `-funroll-loops`.

`-fpeel-loops`

Peels loops for which there is enough information that they do not roll much (from profile feedback or static analysis). It also turns on complete loop peeling (i.e. complete removal of loops with small constant number of iterations).

Enabled by `-O3`, `-fprofile-use`, and `-fauto-profile`.

`-fmove-loop-invariants`

Enables the loop invariant motion pass in the RTL loop optimizer. Enabled at level `-O1` and higher, except for `-Og`.

`-fmove-loop-stores`

Enables the loop store motion pass in the GIMPLE loop optimizer. This moves invariant stores to after the end of the loop in exchange for carrying the stored value in a register across the iteration. Note for this option to have an effect `-ftree-loop-im` has to be enabled as well. Enabled at level `-O1` and higher, except for `-Og`.

`-fsplit-loops`

Split a loop into two if it contains a condition that’s always true for one side of the iteration space and false for the other.

Enabled by `-fprofile-use` and `-fauto-profile`.

`-funswitch-loops`

Move branches with loop invariant conditions out of the loop, with duplicates of the loop on both branches (modified according to result of the condition).

Enabled by `-fprofile-use` and `-fauto-profile`.

`-fversion-loops-for-strides`

If a loop iterates over an array with a variable stride, create another version of the loop that assumes the stride is always one. For example:

```
for (int i = 0; i < n; ++i)
  x[i * stride] = …;
```

becomes:

```
if (stride == 1)
  for (int i = 0; i < n; ++i)
    x[i] = …;
else
  for (int i = 0; i < n; ++i)
    x[i * stride] = …;
```

This is particularly useful for assumed-shape arrays in Fortran where (for example) it allows better vectorization assuming contiguous accesses. This flag is enabled by default at `-O3`. It is also enabled by `-fprofile-use` and `-fauto-profile`.

`-ffunction-sections`
`-fdata-sections`

Place each function or data item into its own section in the output file if the target supports arbitrary sections. The name of the function or the name of the data item determines the section’s name in the output file.

Use these options on systems where the linker can perform optimizations to improve locality of reference in the instruction space. Most systems using the ELF object format have linkers with such optimizations. On AIX, the linker rearranges sections (CSECTs) based on the call graph. The performance impact varies.

Together with a linker garbage collection (linker `--gc-sections` option) these options may lead to smaller statically-linked executables (after stripping).

On ELF/DWARF systems these options do not degenerate the quality of the debug information. There could be issues with other object files/debug info formats.

Only use these options when there are significant benefits from doing so. When you specify these options, the assembler and linker create larger object and executable files and are also slower. These options affect code generation. They prevent optimizations by the compiler and assembler using relative locations inside a translation unit since the locations are unknown until link time. An example of such an optimization is relaxing calls to short call instructions.

`-fstdarg-opt`

Optimize the prologue of variadic argument functions with respect to usage of those arguments.

`-fsection-anchors`

Try to reduce the number of symbolic address calculations by using shared "anchor" symbols to address nearby objects. This transformation can help to reduce the number of GOT entries and GOT accesses on some targets.

For example, the implementation of the following function `foo`:

```
static int a, b, c;
int foo (void) { return a + b + c; }
```

usually calculates the addresses of all three variables, but if you compile it with `-fsection-anchors`, it accesses the variables from a common anchor point instead. The effect is similar to the following pseudocode (which isn't valid C):

```
int foo (void)
{
  register int *xr = &x;
  return xr[&a - &x] + xr[&b - &x] + xr[&c - &x];
}
```

Not all targets support this option.

`-fzero-call-used-regs=`*`choice`*

Zero call-used registers at function return to increase program security by either mitigating Return-Oriented Programming (ROP) attacks or preventing information leakage through registers.

The possible values of *choice* are the same as for the `zero_call_used_regs` attribute (see Function Attributes). The default is '`skip`'.

You can control this behavior for a specific function by using the function attribute `zero_call_used_regs` (see Function Attributes).

`--param` *`name=value`*

In some places, GCC uses various constants to control the amount of optimization that is done. For example, GCC does not inline functions that contain more than a certain number of instructions. You can control some of these constants on the command line using the `--param` option.

The names of specific parameters, and the meaning of the values, are tied to the internals of the compiler, and are subject to change without notice in future releases.

In order to get minimal, maximal and default value of a parameter, one can use `--help=param -Q` options.

In each case, the *value* is an integer. The following choices of *name* are recognized for all targets:

`predictable-branch-outcome`

> When branch is predicted to be taken with probability lower than this threshold (in percent), then it is considered well predictable.

`max-rtl-if-conversion-insns`

> RTL if-conversion tries to remove conditional branches around a block and replace them with conditionally executed instructions. This parameter gives the maximum number of instructions in a block which should be considered for if-conversion. The compiler will also use other heuristics to decide whether if-conversion is likely to be profitable.

`max-rtl-if-conversion-predictable-cost`

> RTL if-conversion will try to remove conditional branches around a block and replace them with conditionally executed instructions. These parameters give the maximum permissible cost for the sequence that would be generated by if-conversion depending on whether the branch is statically determined to be predictable or not. The units for this parameter are the same as those for the GCC internal seq_cost metric. The compiler will try to provide a reasonable default for this parameter using the BRANCH_COST target macro.

`max-crossjump-edges`

> The maximum number of incoming edges to consider for cross-jumping. The algorithm used by `-fcrossjumping` is *O(N^2)* in the number of edges incoming to each block. Increasing values mean more aggressive optimization, making the compilation time increase with probably small improvement in executable size.

`min-crossjump-insns`

> The minimum number of instructions that must be matched at the end of two blocks before cross-jumping is performed on them. This value is ignored in the case where all instructions in the block being cross-jumped from are matched.

`max-grow-copy-bb-insns`

> The maximum code size expansion factor when copying basic blocks instead of jumping. The expansion is relative to a jump instruction.

`max-goto-duplication-insns`

> The maximum number of instructions to duplicate to a block that jumps to a computed goto. To avoid *O(N^2)* behavior in a number of passes, GCC factors computed gotos early in the compilation process, and unfactors them as late as possible. Only computed jumps at the end of a basic blocks with no more than max-goto-duplication-insns are unfactored.

`max-delay-slot-insn-search`

> The maximum number of instructions to consider when looking for an instruction to fill a delay slot. If more than this arbitrary number of instructions are searched, the time savings from filling the delay slot are minimal, so stop searching. Increasing values mean more aggressive optimization, making the compilation time increase with probably small improvement in execution time.

`max-delay-slot-live-search`

When trying to fill delay slots, the maximum number of instructions to consider when searching for a block with valid live register information. Increasing this arbitrarily chosen value means more aggressive optimization, increasing the compilation time. This parameter should be removed when the delay slot code is rewritten to maintain the control-flow graph.

`max-gcse-memory`

The approximate maximum amount of memory in `kB` that can be allocated in order to perform the global common subexpression elimination optimization. If more memory than specified is required, the optimization is not done.

`max-gcse-insertion-ratio`

If the ratio of expression insertions to deletions is larger than this value for any expression, then RTL PRE inserts or removes the expression and thus leaves partially redundant computations in the instruction stream.

`max-pending-list-length`

The maximum number of pending dependencies scheduling allows before flushing the current state and starting over. Large functions with few branches or calls can create excessively large lists which needlessly consume memory and resources.

`max-modulo-backtrack-attempts`

The maximum number of backtrack attempts the scheduler should make when modulo scheduling a loop. Larger values can exponentially increase compilation time.

`max-inline-functions-called-once-loop-depth`

Maximal loop depth of a call considered by inline heuristics that tries to inline all functions called once.

`max-inline-functions-called-once-insns`

Maximal estimated size of functions produced while inlining functions called once.

`max-inline-insns-single`

Several parameters control the tree inliner used in GCC. This number sets the maximum number of instructions (counted in GCC’s internal representation) in a single function that the tree inliner considers for inlining. This only affects functions declared inline and methods implemented in a class declaration (C++).

`max-inline-insns-auto`

When you use `-finline-functions` (included in `-O3`), a lot of functions that would otherwise not be considered for inlining by the compiler are investigated. To those functions, a different (more restrictive) limit compared to functions declared inline can be applied (`--param max-inline-insns-auto`).

`max-inline-insns-small`

This is bound applied to calls which are considered relevant with `-finline-small-functions`.

`max-inline-insns-size`

This is bound applied to calls which are optimized for size. Small growth may be desirable to anticipate optimization oppurtunities exposed by inlining.

`uninlined-function-insns`

Number of instructions accounted by inliner for function overhead such as function prologue and epilogue.

`uninlined-function-time`

Extra time accounted by inliner for function overhead such as time needed to execute function prologue and epilogue.

`inline-heuristics-hint-percent`

The scale (in percents) applied to `inline-insns-single`, `inline-insns-single-O2`, `inline-insns-auto` when inline heuristics hints that inlining is very profitable (will enable later optimizations).

`uninlined-thunk-insns`
`uninlined-thunk-time`

Same as `--param uninlined-function-insns` and `--param uninlined-function-time` but applied to function thunks.

`inline-min-speedup`

When estimated performance improvement of caller + callee runtime exceeds this threshold (in percent), the function can be inlined regardless of the limit on `--param max-inline-insns-single` and `--param max-inline-insns-auto`.

`large-function-insns`

The limit specifying really large functions. For functions larger than this limit after inlining, inlining is constrained by `--param large-function-growth`. This parameter is useful primarily to avoid extreme compilation time caused by non-linear algorithms used by the back end.

`large-function-growth`

Specifies maximal growth of large function caused by inlining in percents. For example, parameter value 100 limits large function growth to 2.0 times the original size.

`large-unit-insns`

The limit specifying large translation unit. Growth caused by inlining of units larger than this limit is limited by `--param inline-unit-growth`. For small units this might be too tight. For example, consider a unit consisting of function A that is inline and B that just calls A three times. If B is small relative to A, the growth of unit is 300\% and yet such inlining is very sane. For very large units consisting of small inlineable functions, however, the overall unit growth limit is needed to avoid exponential explosion of code size. Thus for smaller units, the size is increased to `--param large-unit-insns` before applying `--param inline-unit-growth`.

`lazy-modules`

Maximum number of concurrently open C++ module files when lazy loading.

`inline-unit-growth`

Specifies maximal overall growth of the compilation unit caused by inlining. For example, parameter value 20 limits unit growth to 1.2 times the original size. Cold functions (either marked cold via an attribute or by profile feedback) are not accounted into the unit size.

`ipa-cp-unit-growth`

Specifies maximal overall growth of the compilation unit caused by interprocedural constant propagation. For example, parameter value 10 limits unit growth to 1.1 times the original size.

`ipa-cp-large-unit-insns`

The size of translation unit that IPA-CP pass considers large.

`large-stack-frame`

The limit specifying large stack frames. While inlining the algorithm is trying to not grow past this limit too much.

`large-stack-frame-growth`

Specifies maximal growth of large stack frames caused by inlining in percents. For example, parameter value 1000 limits large stack frame growth to 11 times the original size.

`max-inline-insns-recursive`
`max-inline-insns-recursive-auto`

Specifies the maximum number of instructions an out-of-line copy of a self-recursive inline function can grow into by performing recursive inlining.

`--param max-inline-insns-recursive` applies to functions declared inline. For functions not declared inline, recursive inlining happens only when `-finline-functions` (included in `-O3`) is enabled; `--param max-inline-insns-recursive-auto` applies instead.

`max-inline-recursive-depth`
`max-inline-recursive-depth-auto`

Specifies the maximum recursion depth used for recursive inlining.

`--param max-inline-recursive-depth` applies to functions declared inline. For functions not declared inline, recursive inlining happens only when `-finline-functions` (included in `-O3`) is enabled; `--param max-inline-recursive-depth-auto` applies instead.

`min-inline-recursive-probability`

Recursive inlining is profitable only for function having deep recursion in average and can hurt for function having little recursion depth by increasing the prologue size or complexity of function body to other optimizers.

When profile feedback is available (see `-fprofile-generate`) the actual recursion depth can be guessed from the probability that function recurses via a given call expression. This parameter limits inlining only to call expressions whose probability exceeds the given threshold (in percents).

`early-inlining-insns`

Specify growth that the early inliner can make. In effect it increases the amount of inlining for code having a large abstraction penalty.

`max-early-inliner-iterations`

Limit of iterations of the early inliner. This basically bounds the number of nested indirect calls the early inliner can resolve. Deeper chains are still handled by late inlining.

`comdat-sharing-probability`

Probability (in percent) that C++ inline function with comdat visibility are shared across multiple compilation units.

`modref-max-bases`
`modref-max-refs`
`modref-max-accesses`

Specifies the maximal number of base pointers, references and accesses stored for a single function by mod/ref analysis.

`modref-max-tests`

Specifies the maxmal number of tests alias oracle can perform to disambiguate memory locations using the mod/ref information. This parameter ought to be bigger than `--param modref-max-bases` and `--param modref-max-refs`.

`modref-max-depth`

Specifies the maximum depth of DFS walk used by modref escape analysis. Setting to 0 disables the analysis completely.

`modref-max-escape-points`

Specifies the maximum number of escape points tracked by modref per SSA-name.

`modref-max-adjustments`

Specifies the maximum number the access range is enlarged during modref dataflow analysis.

`profile-func-internal-id`

A parameter to control whether to use function internal id in profile database lookup. If the value is 0, the compiler uses an id that is based on function assembler name and filename, which makes old profile data more tolerant to source changes such as function reordering etc.

`min-vect-loop-bound`

The minimum number of iterations under which loops are not vectorized when `-ftree-vectorize` is used. The number of iterations after vectorization needs to be greater than the value specified by this option to allow vectorization.

`gcse-cost-distance-ratio`

Scaling factor in calculation of maximum distance an expression can be moved by GCSE optimizations. This is currently supported only in the code hoisting pass. The bigger the ratio, the more aggressive code hoisting is with simple expressions, i.e., the expressions that have cost less than `gcse-unrestricted-cost`. Specifying 0 disables hoisting of simple expressions.

`gcse-unrestricted-cost`

Cost, roughly measured as the cost of a single typical machine instruction, at which GCSE optimizations do not constrain the distance an expression can travel. This is currently supported only in the code hoisting pass. The lesser the cost, the more aggressive code hoisting is. Specifying 0 allows all expressions to travel unrestricted distances.

`max-hoist-depth`

The depth of search in the dominator tree for expressions to hoist. This is used to avoid quadratic behavior in hoisting algorithm. The value of 0 does not limit on the search, but may slow down compilation of huge functions.

`max-tail-merge-comparisons`

The maximum amount of similar bbs to compare a bb with. This is used to avoid quadratic behavior in tree tail merging.

`max-tail-merge-iterations`

The maximum amount of iterations of the pass over the function. This is used to limit compilation time in tree tail merging.

`store-merging-allow-unaligned`

Allow the store merging pass to introduce unaligned stores if it is legal to do so.

`max-stores-to-merge`

The maximum number of stores to attempt to merge into wider stores in the store merging pass.

`max-store-chains-to-track`

The maximum number of store chains to track at the same time in the attempt to merge them into wider stores in the store merging pass.

`max-stores-to-track`

The maximum number of stores to track at the same time in the attemt to to merge them into wider stores in the store merging pass.

`max-unrolled-insns`

The maximum number of instructions that a loop may have to be unrolled. If a loop is unrolled, this parameter also determines how many times the loop code is unrolled.

`max-average-unrolled-insns`

The maximum number of instructions biased by probabilities of their execution that a loop may have to be unrolled. If a loop is unrolled, this parameter also determines how many times the loop code is unrolled.

`max-unroll-times`

The maximum number of unrollings of a single loop.

`max-peeled-insns`

The maximum number of instructions that a loop may have to be peeled. If a loop is peeled, this parameter also determines how many times the loop code is peeled.

`max-peel-times`

The maximum number of peelings of a single loop.

`max-peel-branches`

The maximum number of branches on the hot path through the peeled sequence.

`max-completely-peeled-insns`

The maximum number of insns of a completely peeled loop.

`max-completely-peel-times`

The maximum number of iterations of a loop to be suitable for complete peeling.

`max-completely-peel-loop-nest-depth`

The maximum depth of a loop nest suitable for complete peeling.

`max-unswitch-insns`

The maximum number of insns of an unswitched loop.

`max-unswitch-depth`

The maximum depth of a loop nest to be unswitched.

`lim-expensive`

The minimum cost of an expensive expression in the loop invariant motion.

`min-loop-cond-split-prob`

When FDO profile information is available, `min-loop-cond-split-prob` specifies minimum threshold for probability of semi-invariant condition statement to trigger loop split.

`iv-consider-all-candidates-bound`

Bound on number of candidates for induction variables, below which all candidates are considered for each use in induction variable optimizations. If there are more candidates than this, only the most relevant ones are considered to avoid quadratic time complexity.

`iv-max-considered-uses`

The induction variable optimizations give up on loops that contain more induction variable uses.

`iv-always-prune-cand-set-bound`

If the number of candidates in the set is smaller than this value, always try to remove unnecessary ivs from the set when adding a new one.

`avg-loop-niter`

Average number of iterations of a loop.

`dse-max-object-size`

Maximum size (in bytes) of objects tracked bytewise by dead store elimination. Larger values may result in larger compilation times.

`dse-max-alias-queries-per-store`

Maximum number of queries into the alias oracle per store. Larger values result in larger compilation times and may result in more removed dead stores.

`scev-max-expr-size`

Bound on size of expressions used in the scalar evolutions analyzer. Large expressions slow the analyzer.

`scev-max-expr-complexity`

Bound on the complexity of the expressions in the scalar evolutions analyzer. Complex expressions slow the analyzer.

`max-tree-if-conversion-phi-args`

Maximum number of arguments in a PHI supported by TREE if conversion unless the loop is marked with simd pragma.

`vect-max-layout-candidates`

The maximum number of possible vector layouts (such as permutations) to consider when optimizing to-be-vectorized code.

`vect-max-version-for-alignment-checks`

The maximum number of run-time checks that can be performed when doing loop versioning for alignment in the vectorizer.

`vect-max-version-for-alias-checks`

The maximum number of run-time checks that can be performed when doing loop versioning for alias in the vectorizer.

`vect-max-peeling-for-alignment`

The maximum number of loop peels to enhance access alignment for vectorizer. Value -1 means no limit.

`max-iterations-to-track`

The maximum number of iterations of a loop the brute-force algorithm for analysis of the number of iterations of the loop tries to evaluate.

`hot-bb-count-fraction`

The denominator n of fraction 1/n of the maximal execution count of a basic block in the entire program that a basic block needs to at least have in order to be considered hot. The default is 10000, which means that a basic block is considered hot if its execution count is greater than 1/10000 of the maximal execution count. 0 means that it is never considered hot. Used in non-LTO mode.

`hot-bb-count-ws-permille`

The number of most executed permilles, ranging from 0 to 1000, of the profiled execution of the entire program to which the execution count of a basic block must be part of in order to be considered hot. The default is 990, which means that a basic block is considered hot if its execution count contributes to the upper 990 permilles, or 99.0%, of the profiled execution of the entire program. 0 means that it is never considered hot. Used in LTO mode.

`hot-bb-frequency-fraction`

The denominator n of fraction 1/n of the execution frequency of the entry block of a function that a basic block of this function needs to at least have in order to be considered hot. The default is 1000, which means that a basic block is considered hot in a function if it is executed more frequently than 1/1000 of the frequency of the entry block of the function. 0 means that it is never considered hot.

`unlikely-bb-count-fraction`

The denominator n of fraction 1/n of the number of profiled runs of the entire program below which the execution count of a basic block must be in order for the basic block to be considered unlikely executed. The default is 20, which means that a basic block is considered unlikely executed if it is executed in fewer than 1/20, or 5%, of the runs of the program. 0 means that it is always considered unlikely executed.

`max-predicted-iterations`

The maximum number of loop iterations we predict statically. This is useful in cases where a function contains a single loop with known bound and another loop with unknown bound. The known number of iterations is predicted correctly, while the unknown number of iterations average to roughly 10. This means that the loop without bounds appears artificially cold relative to the other one.

`builtin-expect-probability`

Control the probability of the expression having the specified value. This parameter takes a percentage (i.e. 0 ... 100) as input.

`builtin-string-cmp-inline-length`

The maximum length of a constant string for a builtin string cmp call eligible for inlining.

`align-threshold`

Select fraction of the maximal frequency of executions of a basic block in a function to align the basic block.

`align-loop-iterations`

A loop expected to iterate at least the selected number of iterations is aligned.

`tracer-dynamic-coverage`
`tracer-dynamic-coverage-feedback`

This value is used to limit superblock formation once the given percentage of executed instructions is covered. This limits unnecessary code size expansion.

The `tracer-dynamic-coverage-feedback` parameter is used only when profile feedback is available. The real profiles (as opposed to statically estimated ones) are much less balanced allowing the threshold to be larger value.

`tracer-max-code-growth`

Stop tail duplication once code growth has reached given percentage. This is a rather artificial limit, as most of the duplicates are eliminated later in cross jumping, so it may be set to much higher values than is the desired code growth.

`tracer-min-branch-ratio`

Stop reverse growth when the reverse probability of best edge is less than this threshold (in percent).

`tracer-min-branch-probability`
`tracer-min-branch-probability-feedback`

Stop forward growth if the best edge has probability lower than this threshold.

Similarly to `tracer-dynamic-coverage` two parameters are provided. `tracer-min-branch-probability-feedback` is used for compilation with profile feedback and `tracer-min-branch-probability` compilation without. The value for compilation with profile feedback needs to be more conservative (higher) in order to make tracer effective.

`stack-clash-protection-guard-size`

Specify the size of the operating system provided stack guard as 2 raised to *num* bytes. Higher values may reduce the number of explicit probes, but a value larger than the operating system provided guard will leave code vulnerable to stack clash style attacks.

`stack-clash-protection-probe-interval`

Stack clash protection involves probing stack space as it is allocated. This param controls the maximum distance between probes into the stack as 2 raised to *num* bytes. Higher values may reduce the number of explicit probes, but a value larger than the operating system provided guard will leave code vulnerable to stack clash style attacks.

`max-cse-path-length`

The maximum number of basic blocks on path that CSE considers.

`max-cse-insns`

The maximum number of instructions CSE processes before flushing.

`ggc-min-expand`

GCC uses a garbage collector to manage its own memory allocation. This parameter specifies the minimum percentage by which the garbage collector’s heap should be allowed to expand between collections. Tuning this may improve compilation speed; it has no effect on code generation.

The default is 30% + 70% * (RAM/1GB) with an upper bound of 100% when RAM >= 1GB. If `getrlimit` is available, the notion of “RAM” is the smallest of actual RAM and `RLIMIT_DATA` or `RLIMIT_AS`. If GCC is not able to calculate RAM on a particular platform, the lower bound of 30% is used. Setting this parameter and `ggc-min-heapsize` to zero causes a full collection to occur at every opportunity. This is extremely slow, but can be useful for debugging.

`ggc-min-heapsize`

Minimum size of the garbage collector’s heap before it begins bothering to collect garbage. The first collection occurs after the heap expands by `ggc-min-expand`% beyond `ggc-min-heapsize`. Again, tuning this may improve compilation speed, and has no effect on code generation.

The default is the smaller of RAM/8, RLIMIT_RSS, or a limit that tries to ensure that RLIMIT_DATA or RLIMIT_AS are not exceeded, but with a lower bound of 4096 (four megabytes) and an upper bound of 131072 (128 megabytes). If GCC is not able to calculate RAM on a particular platform, the lower bound is used. Setting this parameter very large effectively disables garbage collection. Setting this parameter and `ggc-min-expand` to zero causes a full collection to occur at every opportunity.

`max-reload-search-insns`

The maximum number of instruction reload should look backward for equivalent register. Increasing values mean more aggressive optimization, making the compilation time increase with probably slightly better performance.

`max-cselib-memory-locations`

The maximum number of memory locations cselib should take into account. Increasing values mean more aggressive optimization, making the compilation time increase with probably slightly better performance.

`max-sched-ready-insns`

The maximum number of instructions ready to be issued the scheduler should consider at any given time during the first scheduling pass. Increasing values mean more thorough searches, making the compilation time increase with probably little benefit.

`max-sched-region-blocks`

The maximum number of blocks in a region to be considered for interblock scheduling.

`max-pipeline-region-blocks`

The maximum number of blocks in a region to be considered for pipelining in the selective scheduler.

`max-sched-region-insns`

The maximum number of insns in a region to be considered for interblock scheduling.

`max-pipeline-region-insns`

The maximum number of insns in a region to be considered for pipelining in the selective scheduler.

`min-spec-prob`

The minimum probability (in percents) of reaching a source block for interblock speculative scheduling.

`max-sched-extend-regions-iters`

The maximum number of iterations through CFG to extend regions. A value of 0 disables region extensions.

`max-sched-insn-conflict-delay`

The maximum conflict delay for an insn to be considered for speculative motion.

`sched-spec-prob-cutoff`

The minimal probability of speculation success (in percents), so that speculative insns are scheduled.

`sched-state-edge-prob-cutoff`

The minimum probability an edge must have for the scheduler to save its state across it.

`sched-mem-true-dep-cost`

Minimal distance (in CPU cycles) between store and load targeting same memory locations.

`selsched-max-lookahead`

The maximum size of the lookahead window of selective scheduling. It is a depth of search for available instructions.

`selsched-max-sched-times`

The maximum number of times that an instruction is scheduled during selective scheduling. This is the limit on the number of iterations through which the instruction may be pipelined.

`selsched-insns-to-rename`

The maximum number of best instructions in the ready list that are considered for renaming in the selective scheduler.

`sms-min-sc`

The minimum value of stage count that swing modulo scheduler generates.

`max-last-value-rtl`

The maximum size measured as number of RTLs that can be recorded in an expression in combiner for a pseudo register as last known value of that register.

`max-combine-insns`

The maximum number of instructions the RTL combiner tries to combine.

`integer-share-limit`

Small integer constants can use a shared data structure, reducing the compiler's memory usage and increasing its speed. This sets the maximum value of a shared integer constant.

`ssp-buffer-size`

The minimum size of buffers (i.e. arrays) that receive stack smashing protection when `-fstack-protector` is used.

`min-size-for-stack-sharing`

The minimum size of variables taking part in stack slot sharing when not optimizing.

`max-jump-thread-duplication-stmts`

Maximum number of statements allowed in a block that needs to be duplicated when threading jumps.

`max-jump-thread-paths`

The maximum number of paths to consider when searching for jump threading opportunities. When arriving at a block, incoming edges are only considered if the number of paths to be searched so far multiplied by the number of incoming edges does not exhaust the specified maximum number of paths to consider.

`max-fields-for-field-sensitive`

Maximum number of fields in a structure treated in a field sensitive manner during pointer analysis.

`prefetch-latency`

Estimate on average number of instructions that are executed before prefetch finishes. The distance prefetched ahead is proportional to this constant. Increasing this number may also lead to less streams being prefetched (see `simultaneous-prefetches`).

`simultaneous-prefetches`

Maximum number of prefetches that can run at the same time.

`l1-cache-line-size`

The size of cache line in L1 data cache, in bytes.

`l1-cache-size`

The size of L1 data cache, in kilobytes.

`l2-cache-size`

The size of L2 data cache, in kilobytes.

`prefetch-dynamic-strides`

Whether the loop array prefetch pass should issue software prefetch hints for strides that are non-constant. In some cases this may be beneficial, though the fact the stride is non-constant may make it hard to predict when there is clear benefit to issuing these hints.

Set to 1 if the prefetch hints should be issued for non-constant strides. Set to 0 if prefetch hints should be issued only for strides that are known to be constant and below `prefetch-minimum-stride`.

`prefetch-minimum-stride`

Minimum constant stride, in bytes, to start using prefetch hints for. If the stride is less than this threshold, prefetch hints will not be issued.

This setting is useful for processors that have hardware prefetchers, in which case there may be conflicts between the hardware prefetchers and the software prefetchers. If the hardware prefetchers have a maximum stride they can handle, it should be used here to improve the use of software prefetchers.

A value of -1 means we don’t have a threshold and therefore prefetch hints can be issued for any constant stride.

This setting is only useful for strides that are known and constant.

`destructive-interference-size`
`constructive-interference-size`

The values for the C++17 variables `std::hardware_destructive_interference_size` and `std::hardware_constructive_interference_size`. The destructive interference size is the minimum recommended offset between two independent concurrently-accessed objects; the constructive interference size is the maximum recommended size of contiguous memory accessed together. Typically both will be the size of an L1 cache line for the target, in bytes. For a generic target covering a range of L1 cache line sizes, typically the constructive interference size will be the small end of the range and the destructive size will be the large end.

The destructive interference size is intended to be used for layout, and thus has ABI impact. The default value is not expected to be stable, and on some targets varies with `-mtune`, so use of this variable in a context where ABI stability is important, such as the public interface of a library, is strongly discouraged; if it is used in that context, users can stabilize the value using this option.

The constructive interference size is less sensitive, as it is typically only used in a ‘`static_assert`’ to make sure that a type fits within a cache line.

See also `-Winterference-size`.

`loop-interchange-max-num-stmts`

The maximum number of stmts in a loop to be interchanged.

`loop-interchange-stride-ratio`

The minimum ratio between stride of two loops for interchange to be profitable.

`min-insn-to-prefetch-ratio`

The minimum ratio between the number of instructions and the number of prefetches to enable prefetching in a loop.

`prefetch-min-insn-to-mem-ratio`

The minimum ratio between the number of instructions and the number of memory references to enable prefetching in a loop.

`use-canonical-types`

Whether the compiler should use the “canonical” type system. Should always be 1, which uses a more efficient internal mechanism for comparing types in C++ and Objective-C++. However, if bugs in the canonical type system are causing compilation failures, set this value to 0 to disable canonical types.

`switch-conversion-max-branch-ratio`

Switch initialization conversion refuses to create arrays that are bigger than `switch-conversion-max-branch-ratio` times the number of branches in the switch.

`max-partial-antic-length`

Maximum length of the partial antic set computed during the tree partial redundancy elimination optimization (`-ftree-pre`) when optimizing at `-O3` and above. For some sorts of source code the enhanced partial redundancy elimination optimization can run away, consuming all of the memory available on the host machine. This parameter sets a limit on the length of the sets that are computed, which prevents the runaway behavior. Setting a value of 0 for this parameter allows an unlimited set length.

`rpo-vn-max-loop-depth`

Maximum loop depth that is value-numbered optimistically. When the limit hits the innermost *rpo-vn-max-loop-depth* loops and the outermost loop in the loop nest are value-numbered optimistically and the remaining ones not.

`sccvn-max-alias-queries-per-access`

Maximum number of alias-oracle queries we perform when looking for redundancies for loads and stores. If this limit is hit the search is aborted and the load or store is not considered redundant. The number of queries is algorithmically limited to the number of stores on all paths from the load to the function entry.

`ira-max-loops-num`

IRA uses regional register allocation by default. If a function contains more loops than the number given by this parameter, only at most the given number of the most frequently-executed loops form regions for regional register allocation.

`ira-max-conflict-table-size`

Although IRA uses a sophisticated algorithm to compress the conflict table, the table can still require excessive amounts of memory for huge functions. If the conflict table for a function could be more than the size in MB given by this parameter, the register allocator instead uses a faster, simpler, and lower-quality algorithm that does not require building a pseudo-register conflict table.

`ira-loop-reserved-regs`

IRA can be used to evaluate more accurate register pressure in loops for decisions to move loop invariants (see `-O3`). The number of available registers reserved for some other purposes is given by this parameter. Default of the parameter is the best found from numerous experiments.

`ira-consider-dup-in-all-alts`

Make IRA to consider matching constraint (duplicated operand number) heavily in all available alternatives for preferred register class. If it is set as zero, it means IRA only respects the matching constraint when it’s in the only available alternative with an appropriate register class. Otherwise, it means IRA will check all available alternatives for preferred register class even if it has found some choice with an appropriate register class and respect the found qualified matching constraint.

`lra-inheritance-ebb-probability-cutoff`

LRA tries to reuse values reloaded in registers in subsequent insns. This optimization is called inheritance. EBB is used as a region to do this optimization. The parameter defines a minimal fall-through edge probability in percentage used to add BB to inheritance EBB in LRA. The default value was chosen from numerous runs of SPEC2000 on x86-64.

`loop-invariant-max-bbs-in-loop`

Loop invariant motion can be very expensive, both in compilation time and in amount of needed compile-time memory, with very large loops. Loops with more basic blocks than this parameter won't have loop invariant motion optimization performed on them.

`loop-max-datarefs-for-datadeps`

Building data dependencies is expensive for very large loops. This parameter limits the number of data references in loops that are considered for data dependence analysis. These large loops are no handled by the optimizations using loop data dependencies.

`max-vartrack-size`

Sets a maximum number of hash table slots to use during variable tracking dataflow analysis of any function. If this limit is exceeded with variable tracking at assignments enabled, analysis for that function is retried without it, after removing all debug insns from the function. If the limit is exceeded even without debug insns, var tracking analysis is completely disabled for the function. Setting the parameter to zero makes it unlimited.

`max-vartrack-expr-depth`

Sets a maximum number of recursion levels when attempting to map variable names or debug temporaries to value expressions. This trades compilation time for more complete debug information. If this is set too low, value expressions that are available and could be represented in debug information may end up not being used; setting this higher may enable the compiler to find more complex debug expressions, but compile time and memory use may grow.

`max-debug-marker-count`

Sets a threshold on the number of debug markers (e.g. begin stmt markers) to avoid complexity explosion at inlining or expanding to RTL. If a function has more such gimple stmts than the set limit, such stmts will be dropped from the inlined copy of a function, and from its RTL expansion.

`min-nondebug-insn-uid`

Use uids starting at this parameter for nondebug insns. The range below the parameter is reserved exclusively for debug insns created by `-fvar-tracking-assignments`, but debug insns may get (non-overlapping) uids above it if the reserved range is exhausted.

`ipa-sra-deref-prob-threshold`

IPA-SRA replaces a pointer which is known not be NULL with one or more new parameters only when the probability (in percent, relative to function entry) of it being dereferenced is higher than this parameter.

`ipa-sra-ptr-growth-factor`

IPA-SRA replaces a pointer to an aggregate with one or more new parameters only when their cumulative size is less or equal to `ipa-sra-ptr-growth-factor` times the size of the original pointer parameter.

`ipa-sra-ptrwrap-growth-factor`

Additional maximum allowed growth of total size of new parameters that ipa-sra replaces a pointer to an aggregate with, if it points to a local variable that the caller only writes to and passes it as an argument to other functions.

`ipa-sra-max-replacements`

Maximum pieces of an aggregate that IPA-SRA tracks. As a consequence, it is also the maximum number of replacements of a formal parameter.

`sra-max-scalarization-size-Ospeed`
`sra-max-scalarization-size-Osize`

The two Scalar Reduction of Aggregates passes (SRA and IPA-SRA) aim to replace scalar parts of aggregates with uses of independent scalar variables. These parameters control the maximum size, in storage units, of aggregate which is considered for replacement when compiling for speed (`sra-max-scalarization-size-Ospeed`) or size (`sra-max-scalarization-size-Osize`) respectively.

`sra-max-propagations`

The maximum number of artificial accesses that Scalar Replacement of Aggregates (SRA) will track, per one local variable, in order to facilitate copy propagation.

`tm-max-aggregate-size`

When making copies of thread-local variables in a transaction, this parameter specifies the size in bytes after which variables are saved with the logging functions as opposed to save/restore code sequence pairs. This option only applies when using `-fgnu-tm`.

`graphite-max-nb-scop-params`

To avoid exponential effects in the Graphite loop transforms, the number of parameters in a Static Control Part (SCoP) is bounded. A value of zero can be used to lift the bound. A variable whose value is unknown at compilation time and defined outside a SCoP is a parameter of the SCoP.

`loop-block-tile-size`

Loop blocking or strip mining transforms, enabled with `-floop-block` or `-floop-strip-mine`, strip mine each loop in the loop nest by a given number of iterations. The strip length can be changed using the `loop-block-tile-size` parameter.

`ipa-jump-function-lookups`

Specifies number of statements visited during jump function offset discovery.

`ipa-cp-value-list-size`

IPA-CP attempts to track all possible values and types passed to a function’s parameter in order to propagate them and perform devirtualization. `ipa-cp-value-list-size` is the maximum number of values and types it stores per one formal parameter of a function.

`ipa-cp-eval-threshold`

IPA-CP calculates its own score of cloning profitability heuristics and performs those cloning opportunities with scores that exceed `ipa-cp-eval-threshold`.

`ipa-cp-max-recursive-depth`

Maximum depth of recursive cloning for self-recursive function.

`ipa-cp-min-recursive-probability`

Recursive cloning only when the probability of call being executed exceeds the parameter.

`ipa-cp-profile-count-base`

When using `-fprofile-use` option, IPA-CP will consider the measured execution count of a call graph edge at this percentage position in their histogram as the basis for its heuristics calculation.

`ipa-cp-recursive-freq-factor`

The number of times interprocedural copy propagation expects recursive functions to call themselves.

`ipa-cp-recursion-penalty`

Percentage penalty the recursive functions will receive when they are evaluated for cloning.

`ipa-cp-single-call-penalty`

Percentage penalty functions containing a single call to another function will receive when they are evaluated for cloning.

`ipa-max-agg-items`

IPA-CP is also capable to propagate a number of scalar values passed in an aggregate. `ipa-max-agg-items` controls the maximum number of such values per one parameter.

`ipa-cp-loop-hint-bonus`

When IPA-CP determines that a cloning candidate would make the number of iterations of a loop known, it adds a bonus of `ipa-cp-loop-hint-bonus` to the profitability score of the candidate.

`ipa-max-loop-predicates`

The maximum number of different predicates IPA will use to describe when loops in a function have known properties.

`ipa-max-aa-steps`

During its analysis of function bodies, IPA-CP employs alias analysis in order to track values pointed to by function parameters. In order not spend too much time analyzing huge functions, it gives up and consider all memory clobbered after examining `ipa-max-aa-steps` statements modifying memory.

`ipa-max-switch-predicate-bounds`

Maximal number of boundary endpoints of case ranges of switch statement. For switch exceeding this limit, IPA-CP will not construct cloning cost predicate, which is used to estimate cloning benefit, for default case of the switch statement.

`ipa-max-param-expr-ops`

IPA-CP will analyze conditional statement that references some function parameter to estimate benefit for cloning upon certain constant value. But if number of operations in a parameter expression exceeds `ipa-max-param-expr-ops`, the expression is treated as complicated one, and is not handled by IPA analysis.

`lto-partitions`

Specify desired number of partitions produced during WHOPR compilation. The number of partitions should exceed the number of CPUs used for compilation.

`lto-min-partition`

Size of minimal partition for WHOPR (in estimated instructions). This prevents expenses of splitting very small programs into too many partitions.

`lto-max-partition`

Size of max partition for WHOPR (in estimated instructions). to provide an upper bound for individual size of partition. Meant to be used only with balanced partitioning.

`lto-max-streaming-parallelism`

Maximal number of parallel processes used for LTO streaming.

`cxx-max-namespaces-for-diagnostic-help`

The maximum number of namespaces to consult for suggestions when C++ name lookup fails for an identifier.

`sink-frequency-threshold`

The maximum relative execution frequency (in percents) of the target block relative to a statement's original block to allow statement sinking of a statement. Larger numbers result in more aggressive statement sinking. A small positive adjustment is applied for statements with memory operands as those are even more profitable so sink.

`max-stores-to-sink`

The maximum number of conditional store pairs that can be sunk. Set to 0 if either vectorization (`-ftree-vectorize`) or if-conversion (`-ftree-loop-if-convert`) is disabled.

`case-values-threshold`

The smallest number of different values for which it is best to use a jump-table instead of a tree of conditional branches. If the value is 0, use the default for the machine.

`jump-table-max-growth-ratio-for-size`

The maximum code size growth ratio when expanding into a jump table (in percent). The parameter is used when optimizing for size.

`jump-table-max-growth-ratio-for-speed`

The maximum code size growth ratio when expanding into a jump table (in percent). The parameter is used when optimizing for speed.

`tree-reassoc-width`

Set the maximum number of instructions executed in parallel in reassociated tree. This parameter overrides target dependent heuristics used by default if has non zero value.

`sched-pressure-algorithm`

Choose between the two available implementations of `-fsched-pressure`. Algorithm 1 is the original implementation and is the more likely to prevent instructions from being reordered. Algorithm 2 was designed to be a compromise between the relatively conservative approach taken by algorithm 1 and the rather aggressive approach taken by the default scheduler. It relies more heavily on having a regular register file and accurate register pressure classes. See `haifa-sched.cc` in the GCC sources for more details.

The default choice depends on the target.

`max-slsr-cand-scan`

Set the maximum number of existing candidates that are considered when seeking a basis for a new straight-line strength reduction candidate.

`asan-globals`

Enable buffer overflow detection for global objects. This kind of protection is enabled by default if you are using `-fsanitize=address` option. To disable global objects protection use `--param asan-globals=0`.

`asan-stack`

Enable buffer overflow detection for stack objects. This kind of protection is enabled by default when using `-fsanitize=address`. To disable stack protection use `--param asan-stack=0` option.

`asan-instrument-reads`

Enable buffer overflow detection for memory reads. This kind of protection is enabled by default when using `-fsanitize=address`. To disable memory reads protection use `--param asan-instrument-reads=0`.

`asan-instrument-writes`

Enable buffer overflow detection for memory writes. This kind of protection is enabled by default when using `-fsanitize=address`. To disable memory writes protection use `--param asan-instrument-writes=0` option.

`asan-memintrin`

Enable detection for built-in functions. This kind of protection is enabled by default when using `-fsanitize=address`. To disable built-in functions protection use `--param asan-memintrin=0`.

`asan-use-after-return`

Enable detection of use-after-return. This kind of protection is enabled by default when using the `-fsanitize=address` option. To disable it use `--param asan-use-after-return=0`.

Note: By default the check is disabled at run time. To enable it, add `detect_stack_use_after_return=1` to the environment variable `ASAN_OPTIONS`.

`asan-instrumentation-with-call-threshold`

If number of memory accesses in function being instrumented is greater or equal to this number, use callbacks instead of inline checks. E.g. to disable inline code use `--param asan-instrumentation-with-call-threshold=0`.

`hwasan-instrument-stack`

Enable hwasan instrumentation of statically sized stack-allocated variables. This kind of instrumentation is enabled by default when using `-fsanitize=hwaddress` and disabled by default when using `-fsanitize=kernel-hwaddress`. To disable stack instrumentation use `--param hwasan-instrument-stack=0`, and to enable it use `--param hwasan-instrument-stack=1`.

`hwasan-random-frame-tag`

When using stack instrumentation, decide tags for stack variables using a deterministic sequence beginning at a random tag for each frame. With this parameter unset tags are chosen using the same sequence but beginning from 1. This is enabled by default for `-fsanitize=hwaddress` and unavailable for `-fsanitize=kernel-hwaddress`. To disable it use `--param hwasan-random-frame-tag=0`.

`hwasan-instrument-allocas`

Enable hwasan instrumentation of dynamically sized stack-allocated variables. This kind of instrumentation is enabled by default when using `-fsanitize=hwaddress` and disabled by default when using `-fsanitize=kernel-hwaddress`. To disable instrumentation of such variables use `--param hwasan-instrument-allocas=0`, and to enable it use `--param hwasan-instrument-allocas=1`.

`hwasan-instrument-reads`

Enable hwasan checks on memory reads. Instrumentation of reads is enabled by default for both `-fsanitize=hwaddress` and `-fsanitize=kernel-hwaddress`. To disable checking memory reads use `--param hwasan-instrument-reads=0`.

`hwasan-instrument-writes`

Enable hwasan checks on memory writes. Instrumentation of writes is enabled by default for both `-fsanitize=hwaddress` and `-fsanitize=kernel-hwaddress`. To disable checking memory writes use `--param hwasan-instrument-writes=0`.

`hwasan-instrument-mem-intrinsics`

Enable hwasan instrumentation of builtin functions. Instrumentation of these builtin functions is enabled by default for both `-fsanitize=hwaddress` and `-fsanitize=kernel-hwaddress`. To disable instrumentation of builtin functions use `--param hwasan-instrument-mem-intrinsics=0`.

`use-after-scope-direct-emission-threshold`

If the size of a local variable in bytes is smaller or equal to this number, directly poison (or unpoison) shadow memory instead of using run-time callbacks.

`tsan-distinguish-volatile`

Emit special instrumentation for accesses to volatiles.

`tsan-instrument-func-entry-exit`

Emit instrumentation calls to __tsan_func_entry() and __tsan_func_exit().

`max-fsm-thread-path-insns`

Maximum number of instructions to copy when duplicating blocks on a finite state automaton jump thread path.

`threader-debug`

threader-debug=[none|all] Enables verbose dumping of the threader solver.

`parloops-chunk-size`

Chunk size of omp schedule for loops parallelized by parloops.

`parloops-schedule`

Schedule type of omp schedule for loops parallelized by parloops (static, dynamic, guided, auto, runtime).

`parloops-min-per-thread`

The minimum number of iterations per thread of an innermost parallelized loop for which the parallelized variant is preferred over the single threaded one. Note that for a parallelized loop nest the minimum number of iterations of the outermost loop per thread is two.

`max-ssa-name-query-depth`

Maximum depth of recursion when querying properties of SSA names in things like fold routines. One level of recursion corresponds to following a use-def chain.

`max-speculative-devirt-maydefs`

The maximum number of may-defs we analyze when looking for a must-def specifying the dynamic type of an object that invokes a virtual call we may be able to devirtualize speculatively.

`evrp-sparse-threshold`

Maximum number of basic blocks before EVRP uses a sparse cache.

`ranger-debug`

Specifies the type of debug output to be issued for ranges.

`evrp-switch-limit`

Specifies the maximum number of switch cases before EVRP ignores a switch.

`unroll-jam-min-percent`

The minimum percentage of memory references that must be optimized away for the unroll-and-jam transformation to be considered profitable.

`unroll-jam-max-unroll`

The maximum number of times the outer loop should be unrolled by the unroll-and-jam transformation.

`max-rtl-if-conversion-unpredictable-cost`

Maximum permissible cost for the sequence that would be generated by the RTL if-conversion pass for a branch that is considered unpredictable.

`max-variable-expansions-in-unroller`

If `-fvariable-expansion-in-unroller` is used, the maximum number of times that an individual variable will be expanded during loop unrolling.

`partial-inlining-entry-probability`

Maximum probability of the entry BB of split region (in percent relative to entry BB of the function) to make partial inlining happen.

`max-tracked-strlens`

Maximum number of strings for which strlen optimization pass will track string lengths.

`gcse-after-reload-partial-fraction`

The threshold ratio for performing partial redundancy elimination after reload.

`gcse-after-reload-critical-fraction`

The threshold ratio of critical edges execution count that permit performing redundancy elimination after reload.

`max-loop-header-insns`

The maximum number of insns in loop header duplicated by the copy loop headers pass.

`vect-epilogues-nomask`

Enable loop epilogue vectorization using smaller vector size.

`vect-partial-vector-usage`

Controls when the loop vectorizer considers using partial vector loads and stores as an alternative to falling back to scalar code. 0 stops the vectorizer from ever using partial vector loads and stores. 1 allows partial vector loads and stores if vectorization removes the need for the code to iterate. 2 allows partial vector loads and stores in all loops. The parameter only has an effect on targets that support partial vector loads and stores.

`vect-inner-loop-cost-factor`

The maximum factor which the loop vectorizer applies to the cost of statements in an inner loop relative to the loop being vectorized. The factor applied is the maximum of the estimated number of iterations of the inner loop and this parameter. The default value of this parameter is 50.

`vect-induction-float`

Enable loop vectorization of floating point inductions.

`avoid-fma-max-bits`

Maximum number of bits for which we avoid creating FMAs.

`sms-loop-average-count-threshold`

A threshold on the average loop count considered by the swing modulo scheduler.

`sms-dfa-history`

The number of cycles the swing modulo scheduler considers when checking conflicts using DFA.

`graphite-allow-codegen-errors`

Whether codegen errors should be ICEs when `-fchecking`.

`sms-max-ii-factor`

A factor for tuning the upper bound that swing modulo scheduler uses for scheduling a loop.

`lra-max-considered-reload-pseudos`

The max number of reload pseudos which are considered during spilling a non-reload pseudo.

`max-pow-sqrt-depth`

Maximum depth of sqrt chains to use when synthesizing exponentiation by a real constant.

`max-dse-active-local-stores`

Maximum number of active local stores in RTL dead store elimination.

`asan-instrument-allocas`

Enable asan allocas/VLAs protection.

`max-iterations-computation-cost`

Bound on the cost of an expression to compute the number of iterations.

`max-isl-operations`

Maximum number of isl operations, 0 means unlimited.

`graphite-max-arrays-per-scop`

Maximum number of arrays per scop.

`max-vartrack-reverse-op-size`

Max. size of loc list for which reverse ops should be added.

`fsm-scale-path-stmts`

Scale factor to apply to the number of statements in a threading path when comparing to the number of (scaled) blocks.

`uninit-control-dep-attempts`

Maximum number of nested calls to search for control dependencies during uninitialized variable analysis.

`fsm-scale-path-blocks`

Scale factor to apply to the number of blocks in a threading path when comparing to the number of (scaled) statements.

`sched-autopref-queue-depth`

Hardware autoprefetcher scheduler model control flag. Number of lookahead cycles the model looks into; at ' ' only enable instruction sorting heuristic.

`loop-versioning-max-inner-insns`

The maximum number of instructions that an inner loop can have before the loop versioning pass considers it too big to copy.

`loop-versioning-max-outer-insns`

The maximum number of instructions that an outer loop can have before the loop versioning pass considers it too big to copy, discounting any instructions in inner loops that directly benefit from versioning.

`ssa-name-def-chain-limit`

The maximum number of SSA_NAME assignments to follow in determining a property of a variable such as its value. This limits the number of iterations or recursive calls GCC performs when optimizing certain statements or when determining their validity prior to issuing diagnostics.

`store-merging-max-size`

Maximum size of a single store merging region in bytes.

`hash-table-verification-limit`

The number of elements for which hash table verification is done for each searched element.

`max-find-base-term-values`

Maximum number of VALUEs handled during a single find_base_term call.

`analyzer-max-enodes-per-program-point`

The maximum number of exploded nodes per program point within the analyzer, before terminating analysis of that point.

`analyzer-max-constraints`

The maximum number of constraints per state.

`analyzer-min-snodes-for-call-summary`

The minimum number of supernodes within a function for the analyzer to consider summarizing its effects at call sites.

`analyzer-max-enodes-for-full-dump`

The maximum depth of exploded nodes that should appear in a dot dump before switching to a less verbose format.

`analyzer-max-recursion-depth`

The maximum number of times a callsite can appear in a call stack within the analyzer, before terminating analysis of a call that would recurse deeper.

`analyzer-max-svalue-depth`

The maximum depth of a symbolic value, before approximating the value as unknown.

`analyzer-max-infeasible-edges`

The maximum number of infeasible edges to reject before declaring a diagnostic as infeasible.

`gimple-fe-computed-hot-bb-threshold`

The number of executions of a basic block which is considered hot. The parameter is used only in GIMPLE FE.

`analyzer-bb-explosion-factor`

The maximum number of ’after supernode’ exploded nodes within the analyzer per supernode, before terminating analysis.

`ranger-logical-depth`

Maximum depth of logical expression evaluation ranger will look through when evaluating outgoing edge ranges.

`relation-block-limit`

Maximum number of relations the oracle will register in a basic block.

`min-pagesize`

Minimum page size for warning purposes.

`openacc-kernels`

Specify mode of OpenACC 'kernels' constructs handling. With `--param=openacc-kernels=decompose`, OpenACC 'kernels' constructs are decomposed into parts, a sequence of compute constructs, each then handled individually. This is work in progress. With `--param=openacc-kernels=parloops`, OpenACC 'kernels' constructs are handled by the '`parloops`' pass, en bloc. This is the current default.

`openacc-privatization`

Specify mode of OpenACC privatization diagnostics for `-fopt-info-omp-note` and applicable `-fdump-tree-*-details`. With `--param=openacc-privatization=quiet`, don't diagnose. This is the current default. With `--param=openacc-privatization=noisy`, do diagnose.

The following choices of *name* are available on AArch64 targets:

`aarch64-sve-compare-costs`

When vectorizing for SVE, consider using "unpacked" vectors for smaller elements and use the cost model to pick the cheapest approach. Also use the cost model to choose between SVE and Advanced SIMD vectorization.

Using unpacked vectors includes storing smaller elements in larger containers and accessing elements with extending loads and truncating stores.

`aarch64-float-recp-precision`

The number of Newton iterations for calculating the reciprocal for float type. The precision of division is proportional to this param when division approximation is enabled. The default value is 1.

`aarch64-double-recp-precision`

The number of Newton iterations for calculating the reciprocal for double type. The precision of division is propotional to this param when division approximation is enabled. The default value is 2.

`aarch64-autovec-preference`

Force an ISA selection strategy for auto-vectorization. Accepts values from 0 to 4, inclusive.

'`0`'

Use the default heuristics.

'`1`'

Use only Advanced SIMD for auto-vectorization.

'`2`'

Use only SVE for auto-vectorization.

'`3`'

Use both Advanced SIMD and SVE. Prefer Advanced SIMD when the costs are deemed equal.

'`4`'

Use both Advanced SIMD and SVE. Prefer SVE when the costs are deemed equal.

The default value is 0.

`aarch64-loop-vect-issue-rate-niters`

The tuning for some AArch64 CPUs tries to take both latencies and issue rates into account when deciding whether a loop should be vectorized using SVE, vectorized using Advanced SIMD, or not vectorized at all. If this parameter is set to *n*, GCC will not use this heuristic for loops that are known to execute in fewer than *n* Advanced SIMD iterations.

`aarch64-vect-unroll-limit`

The vectorizer will use available tuning information to determine whether it would be beneficial to unroll the main vectorized loop and by how much. This parameter set's the upper bound of how much the vectorizer will unroll the main loop. The default value is four.

The following choices of *name* are available on i386 and x86_64 targets:

`x86-stlf-window-ninsns`

Instructions number above which STFL stall penalty can be compensated.

Next: Instrumentation Options, Previous: Debugging Options, Up: Invoking GCC [Contents][Index]