Next: Preprocessor Options, Previous: Debugging Options, Up: Invoking GCC

3.10 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 only enabled if an `-O` level is set on the command line. Otherwise they are disabled, even if individual optimization flags are specified.

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
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fguess-branch-probability
-fif-conversion2
-fif-conversion
-fipa-pure-const
-fipa-profile
-fipa-reference
-fmerge-constants
-fsplit-wide-types
-ftree-bit-ccp
```

```
-ftree-builtin-call-dce
-ftree-ccp
-ftree-ch
-ftree-copyrename
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-slsr
-ftree-sra
-ftree-pta
-ftree-ter
-funit-at-a-time
```

`-O` also turns on `-fomit-frame-pointer` on machines where doing so does not interfere with debugging.

`-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 `-O`. It also turns on the following optimization flags:

```
-fthread-jumps
-falign-functions  -falign-jumps
-falign-loops  -falign-labels
-fcaller-saves
-fcrossjumping
-fcse-follow-jumps  -fcse-skip-blocks
-fdelete-null-pointer-checks
-fdevirtualize
-fexpensive-optimizations
-fgcse  -fgcse-lm
-fhoist-adjacent-loads
-finline-small-functions
-findirect-inlining
-fipa-sra
-foptimize-sibling-calls
-fpartial-inlining
-fpeephole2
-fregmove
-freorder-blocks  -freorder-functions
-frerun-cse-after-loop
-fsched-interblock  -fsched-spec
-fschedule-insns  -fschedule-insns2
-fstrict-aliasing -fstrict-overflow
-ftree-switch-conversion -ftree-tail-merge
-ftree-pre
-ftree-vrp
```

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 `-finline-functions`, `-funswitch-loops`, `-fpredictive-commoning`, `-fgcse-after-`

`reload`, `-ftree-vectorize`, `-fvect-cost-model`, `-ftree-partial-pre` and `-fipa-cp-clone` options.

`-O0`
: Reduce compilation time and make debugging produce the expected results. This is the default.

`-Os`
: Optimize for size. `-Os` enables all `-O2` optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

  `-Os` disables the following optimization flags:

```
-falign-functions  -falign-jumps  -falign-loops
-falign-labels  -freorder-blocks  -freorder-blocks-and-partition
-fprefetch-loop-arrays  -ftree-vect-loop-version
```

`-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` and the Fortran-specific `-fno-protect-parens` and `-fstack-arrays`.

`-Og`
: Optimize debugging experience. `-Og` enables optimizations that do not interfere with debugging. It 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.

  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-default-inline`
: Do not make member functions inline by default merely because they are defined inside the class scope (C++ only). Otherwise, when you specify `-O`, member functions defined inside class scope are compiled inline by default; i.e., you don't need to add ‘`inline`’ in front of the member function name.

`-fno-defer-pop`
: Always pop the arguments to each function call as soon as that function returns. For machines that must pop arguments after a function call, the compiler normally lets arguments accumulate on the stack for several function calls and pops them all at once.

  Disabled at levels `-O`, `-O2`, `-O3`, `-Os`.

`-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 `-O`, `-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`
Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. **It also makes debugging impossible on some machines.**

On some machines, such as the VAX, this flag has no effect, because the standard calling sequence automatically handles the frame pointer and nothing is saved by pretending it doesn't exist. The machine-description macro `FRAME_POINTER_REQUIRED` controls whether a target machine supports this flag. See Register Usage.

Starting with GCC version 4.6, the default setting (when not optimizing for size) for 32-bit GNU/Linux x86 and 32-bit Darwin x86 targets has been changed to `-fomit-frame-pointer`. The default can be reverted to `-fno-omit-frame-pointer` by configuring GCC with the `--enable-frame-pointer` configure option.

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

`-foptimize-sibling-calls`
Optimize sibling and tail recursive calls.

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

`-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 level `-O2`.

`-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 level `-O2`.

`-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 level `-O3`.

`-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`.

`-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-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 `-O`, `-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`
Do not use “decrement and branch” instructions on a count register, but instead generate a sequence of instructions that decrement a register, compare it against zero, 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.

The default is `-fbranch-count-reg`.

`-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`.

`-fmudflap -fmudflapth -fmudflapir`
For front-ends that support it (C and C++), instrument all risky pointer/array dereferencing operations, some standard library string/heap functions, and some other associated constructs with range/validity tests. Modules so instrumented should be immune to buffer overflows, invalid heap use, and some other classes of C/C++ programming errors. The instrumentation relies on a separate runtime library (`libmudflap`), which is linked into a program if `-fmudflap` is given at link time. Run-time behavior of the instrumented program is controlled by the `MUDFLAP_OPTIONS` environment variable. See `env MUDFLAP_OPTIONS=-help a.out` for its options.

Use `-fmudflapth` instead of `-fmudflap` to compile and to link if your program is multi-threaded. Use `-fmudflapir`, in addition to `-fmudflap` or `-fmudflapth`, if instrumentation should ignore pointer reads. This produces less instrumentation (and therefore faster execution) and still provides some protection against outright memory corrupting writes, but allows erroneously read data to propagate within a program.

`-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 `-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 `-O`, `-O2`, `-O3`, `-Os`.

`-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.

`-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.

`-funsafe-loop-optimizations`
This option tells the loop optimizer to assume that loop indices do not overflow, and that loops with nontrivial exit condition are not infinite. This enables a wider range of loop optimizations even if the loop optimizer itself cannot prove that these assumptions are valid. If you use `-Wunsafe-loop-optimizations`, the compiler warns you if it finds this kind of loop.

`-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 `-O` and higher on architectures that support this.

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

`-fdse`
Perform dead store elimination (DSE) on RTL. Enabled by default at `-O` 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 `if-conversion2`.

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

`-fif-conversion2`
Use conditional execution (where available) to transform conditional jumps into branch-less equivalents.

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

`-fdelete-null-pointer-checks`
Assume that programs cannot safely dereference null pointers, and that no code or data element resides there. This 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 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.

Some targets, especially embedded ones, disable this option at all levels. Otherwise it is enabled at all levels: `-O0`, `-O1`, `-O2`, `-O3`, `-Os`. Passes that use the 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`.

`-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 x86 at levels `-O2`, `-O3`.

`-foptimize-register-move`
`-fregmove`

Attempt to reassign register numbers in move instructions and as operands of other simple instructions in order to maximize the amount of register tying. This is especially helpful on machines with two-operand instructions.

Note `-fregmove` and `-foptimize-register-move` are the same optimization.

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

`-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.

`-fira-verbose=`*n*
Control the verbosity of the dump file for the integrated register allocator. The default value is 5. If the value *n* is greater or equal to 10, the dump output is sent to stderr using the same format as *n* minus 10.

`-fdelayed-branch`

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

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

`-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`
Don't schedule instructions across basic blocks. This is normally enabled by default when scheduling before register allocation, i.e. with `-fschedule-insns` or at `-O2` or higher.

`-fno-sched-spec`
Don't allow speculative motion of non-load instructions. This is normally enabled by default 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.

`-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.

`-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.

`-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 `-O` 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 `-O` 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 `-O` and higher.

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

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

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

`-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 `-O` 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`.

`-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 ipcp-unit-growth=`*`value`*). This flag is enabled by default at `-O3`.

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

`-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 `-O` 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 side-effect free. This flag is enabled by default at `-O2` and higher if `-Os` is not also specified.

`-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 `-O` 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 `-O` 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 `-O` 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 `-O` and higher.

`-ftree-loop-linear`

Perform loop interchange transformations on tree. Same as `-floop-interchange`. To use this code transformation, GCC has to be configured with `--with-ppl` and `--with-cloog` to enable the Graphite loop transformation infrastructure.

`-floop-interchange`
Perform loop interchange transformations on loops. Interchanging two nested loops switches the inner and outer loops. For example, given a loop like:

```
DO J = 1, M
  DO I = 1, N
    A(J, I) = A(J, I) * C
  ENDDO
ENDDO
```

loop interchange transforms the loop as if it were written:

```
DO I = 1, N
  DO J = 1, M
    A(J, I) = A(J, I) * C
  ENDDO
ENDDO
```

which can be beneficial when `N` is larger than the caches, because in Fortran, the elements of an array are stored in memory contiguously by column, and the original loop iterates over rows, potentially creating at each access a cache miss. This optimization applies to all the languages supported by GCC and is not limited to Fortran. To use this code transformation, GCC has to be configured with `--with-ppl` and `--with-cloog` to enable the Graphite loop transformation infrastructure.

`-floop-strip-mine`
Perform loop strip mining transformations on loops. Strip mining splits a loop into two nested loops. The outer loop has strides equal to the strip size and the inner loop has strides of the original loop within a strip. The strip length can be changed using the `loop-block-tile-size` parameter. For example, given a loop like:

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

loop strip mining transforms the loop as if it were written:

```
DO II = 1, N, 51
  DO I = II, min (II + 50, N)
    A(I) = A(I) + C
  ENDDO
ENDDO
```

This optimization applies to all the languages supported by GCC and is not limited to Fortran. To use this code transformation, GCC has to be configured with `--with-ppl` and `--with-cloog` to enable the Graphite loop transformation infrastructure.

`-floop-block`
Perform loop blocking transformations on loops. Blocking strip mines each loop in the loop nest such that the memory accesses of the element loops fit inside caches. The strip length can be changed using the `loop-block-tile-size` parameter. For example, given a loop like:

```
DO I = 1, N
  DO J = 1, M
    A(J, I) = B(I) + C(J)
```

```
  ENDDO
ENDDO
```

loop blocking transforms the loop as if it were written:

```
DO II = 1, N, 51
  DO JJ = 1, M, 51
    DO I = II, min (II + 50, N)
      DO J = JJ, min (JJ + 50, M)
        A(J, I) = B(I) + C(J)
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

which can be beneficial when `M` is larger than the caches, because the innermost loop iterates over a smaller amount of data which can be kept in the caches. This optimization applies to all the languages supported by GCC and is not limited to Fortran. To use this code transformation, GCC has to be configured with `--with-ppl` and `--with-cloog` 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 CLooG, 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.

`-fcheck-data-deps`
: Compare the results of several data dependence analyzers. This option is used for debugging the data dependence analyzers.

`-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-if-convert-stores`
: Attempt to also if-convert conditional jumps containing memory writes. This transformation can be unsafe for multi-threaded programs as it transforms conditional memory writes into unconditional memory writes. For example,

```
for (i = 0; i < N; i++)
  if (cond)
    A[i] = expr;
```

is transformed to

```
for (i = 0; i < N; i++)
  A[i] = cond ? expr : A[i];
```

potentially producing data races.

`-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
```

`-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 `-O3`.

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.

`-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.

`-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 `-O` and higher.

`-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 `-O` and higher.

`-ftree-copyrename`
Perform copy renaming on trees. This pass attempts to rename compiler temporaries to other variables at copy locations, usually resulting in variable names which more closely resemble the original variables. This flag is enabled by default at `-O` and higher.

`-ftree-coalesce-inlined-vars`
Tell the copyrename pass (see `-ftree-copyrename`) to attempt to combine small user-defined variables too, but only if they were inlined from other functions. It is a more limited form of `-ftree-coalesce-vars`. This may harm debug information of such inlined variables, but it will keep variables of the inlined-into function apart from each other, such that they are more likely to contain the expected values in a debugging session. This was the default in GCC versions older than 4.7.

`-ftree-coalesce-vars`
Tell the copyrename pass (see `-ftree-copyrename`) to attempt to combine small user-defined variables too, 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, including inlined ones. This option is enabled by default.

`-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 `-O` 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 `-O` and higher.

`-ftree-vectorize`
Perform loop vectorization on trees. This flag is enabled by default at `-O3`.

`-ftree-slp-vectorize`
Perform basic block vectorization on trees. This flag is enabled by default at `-O3` and when `-ftree-vectorize` is enabled.

`-ftree-vect-loop-version`
Perform loop versioning when doing loop vectorization on trees. When a loop appears to be vectorizable except that data alignment or data dependence cannot be determined at compile time, then vectorized and non-vectorized versions of the loop are generated along with run-time checks for alignment or dependence to control which version is executed. This option is enabled by default except at level `-Os` where it is disabled.

`-fvect-cost-model`
Enable cost model for vectorization. This option is enabled by default at `-O3`.

`-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.

`-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.

`-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`. This option makes code larger, and may or may not make it run faster.

`-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`,

`-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.

`-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 level `-O2`.

`-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`.

`-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-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.

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 `-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, for linkonce sections, for functions with a user-defined section attribute and on any architecture that does not support named sections.

`-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.

Also profile feedback must be available to make this option effective. See `-fprofile-arcs` for details.

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`.

`-fstrict-overflow`

Allow the compiler to assume strict signed overflow rules, depending on the language being compiled. For C (and C++) this means that overflow when doing arithmetic with signed numbers is undefined, which means that the compiler may assume that it does not happen. This permits various optimizations. For example, the compiler assumes that an expression like `i + 10 > i` is always true for signed `i`. This assumption is only valid if signed overflow is undefined, as the expression is false if `i + 10` overflows when using twos complement arithmetic. When this option is in effect any attempt to determine whether an operation on signed numbers overflows must be written carefully to not actually involve overflow.

This option also allows the compiler to assume strict pointer semantics: given a pointer to an object, if adding an offset to that pointer does not produce a pointer to the same object, the addition is undefined. This permits the compiler to conclude that `p + u > p` is always true for a pointer `p` and unsigned integer `u`. This assumption is only valid because pointer wraparound is undefined, as the expression is false if `p + u` overflows using twos complement arithmetic.

See also the `-fwrapv` option. Using `-fwrapv` means that integer signed overflow is fully defined: it wraps. When `-fwrapv` is used, there is no difference between `-fstrict-overflow` and `-fno-strict-overflow` for integers. With `-fwrapv` certain types of overflow are permitted. For example, if the compiler gets an overflow when doing arithmetic on constants, the overflowed value can still be used with `-fwrapv`, but not otherwise.

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

`-falign-functions`
`-falign-functions=`*n*

Align the start of functions to the next power-of-two greater than *n*, skipping up to *n* bytes. For instance, `-falign-functions=32` aligns functions to the next 32-byte boundary, but

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

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

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

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

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

`-falign-labels`
`-falign-labels=`*n*
Align all branch targets to a power-of-two boundary, skipping up to *n* bytes like `-falign-functions`. This option can easily make code slower, because it must insert dummy operations for when the branch target is reached in the usual flow of the code.

`-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.

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

`-falign-loops`
`-falign-loops=`*n*
Align loops to a power-of-two boundary, skipping up to *n* bytes like `-falign-functions`. If the loops are executed many times, this makes up for any execution of the dummy operations.

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

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

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

`-falign-jumps`
`-falign-jumps=`*n*
Align branch targets to a power-of-two boundary, for branch targets where the targets can only be reached by jumping, skipping up to *n* bytes like `-falign-functions`. In this case, no dummy operations need be executed.

`-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.

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

`-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.

Enabled at level `-O0`. When disabled explicitly, it also implies `-fno-section-anchors`, which is otherwise enabled at `-O0` on some targets.

`-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.

This option should not be used in combination with `-flto`. Instead relying on a linker plugin should provide safer and more precise information.

`-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` needs to be specified at compile time and during the final link. 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 only important thing to keep in mind is that to enable link-time optimizations the `-flto` flag needs to be passed to both the compile and the link commands.

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.

Note that when a file is compiled with `-flto`, the generated object file is larger than a regular object file because it contains GIMPLE bytecodes and the usual final code. This means that object files with LTO information can be linked as normal object files; if `-flto` is not passed to the linker, no interprocedural optimizations are applied.

Additionally, the optimization flags used to compile individual files are not necessarily related to those used at link time. For instance,

```
gcc -c -O0 -flto foo.c
gcc -c -O0 -flto bar.c
gcc -o myprog -flto -O3 foo.o bar.o
```

This produces individual object files with unoptimized assembler code, but the resulting binary `myprog` is optimized at `-O3`. If, instead, the final binary is generated without `-flto`, then `myprog` is not optimized.

When producing the final binary with `-flto`, 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.

There are some code generation flags preserved by GCC when generating bytecodes, as they need to be used during the final link stage. Currently, the following options are saved into the GIMPLE bytecode files: `-fPIC`, `-fcommon` and all the `-m` target flags.

At link time, these options are read in and reapplied. Note that the current implementation makes no attempt to recognize conflicting values for these options. If different files have conflicting option values (e.g., one file is compiled with `-fPIC` and another isn't), the compiler simply uses the last value read from the bytecode files. It is recommended, then, that you compile all the files participating in the same link with the same options.

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.

Another feature of LTO is that it is possible to apply interprocedural optimizations on files written in different languages. This requires support in the language front end. Currently, the C, C++ and Fortran front ends are capable of emitting GIMPLE bytecodes, so something like this should work:

```
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; all you need to add is `-flto` to all the compile and link commands.

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 enable this feature, use the flag `-fuse-linker-plugin` at link time:

```
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.

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 will not work with an older/newer version of GCC.

Link-time optimization does not work well with generation of debugging information. Combining `-flto` with `-g` is currently experimental and expected to produce wrong results.

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. The default value for *n* is 1.

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.

This option is disabled by default.

`-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.

`-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`). Valid values are 0 (no compression) to 9 (maximum compression). Values outside this range are clamped to either 0 or 9. If the option is not given, a default balanced compression setting is used.

`-flto-report`
Prints a report with internal details on the workings of the link-time optimizer. The contents of this report vary from version to version. It is meant to be useful to GCC developers when processing object files in LTO mode (via `-flto`).

Disabled by default.

`-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.

The default is `-ffat-lto-objects` but this default is intended to change in future releases when linker plugin enabled environments become more common.

`-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 `-O`, `-O2`, `-O3`, `-Os`.

`-fuse-ld=bfd`
: Use the `bfd` linker instead of the default linker.

`-fuse-ld=gold`
: Use the `gold` linker instead of the default linker.

`-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 `-O`, `-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.

`-fprofile-dir=`*path*
: Set the directory to search for the profile data files in to *path*. This option affects only the profile data generated by `-fprofile-generate`, `-ftest-coverage`, `-fprofile-arcs` and used by `-fprofile-use` and `-fbranch-probabilities` and its related options. Both absolute and relative paths can be used. By default, GCC uses the current directory as *path*, thus the profile data file appears in the same directory as the object file.

`-fprofile-generate`
`-fprofile-generate=`*path*
: Enable options usually used for instrumenting application to produce profile useful for later recompilation with profile feedback based optimization. You must use `-fprofile-generate` both when compiling and when linking your program.

  The following options are enabled: `-fprofile-arcs`, `-fprofile-values`, `-fvpt`.

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

`-fprofile-use`
`-fprofile-use=`*path*
: Enable profile feedback directed optimizations, and optimizations generally profitable only with profile feedback available.

  The following options are enabled: `-fbranch-probabilities`, `-fvpt`, `-funroll-loops`, `-fpeel-loops`, `-ftracer`, `-ftree-vectorize`, `ftree-loop-distribute-patterns`

  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 `-Wcoverage-mismatch`. Note this may result in poorly optimized code.

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

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 registers have more precision than the IEEE `float` and `double` types and the processor does not support operations rounding to those types. By default, `-fexcess-precision=fast` is in effect; this means that operations are carried out in the precision of the registers and that it is unpredictable when rounding to the types specified in the source code takes place. When compiling C, if `-fexcess-precision=standard` is specified then excess precision follows the rules specified in ISO C99; 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 if a strict conformance option such as `-std=c99` is used.

`-fexcess-precision=standard` is not implemented for languages other than C, and has no effect if `-funsafe-math-optimizations` or `-ffast-math` is specified. On the x86, it also 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 `-fno-math-errno`, `-funsafe-math-optimizations`, `-ffinite-math-only`, `-fno-rounding-math`, `-fno-signaling-nans` and `-fcx-limited-range`.

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`.

`-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 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.

`-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 Options for Debugging Your Program or `gcc`), 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.

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.c`, 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.

`-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 with `-fprofile-generate` and `-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.

`-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` and `-fpeel-loops`.

`-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 with `-fprofile-use`.

`-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 with `-fprofile-use`.

`-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). It also turns on complete loop peeling (i.e. complete removal of loops with small constant number of iterations).

Enabled with `-fprofile-use`.

`-fmove-loop-invariants`
Enables the loop invariant motion pass in the RTL loop optimizer. Enabled at level `-O1`

`-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).

`-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 and SPARC processors running Solaris 2 have linkers with such optimizations. AIX may have these optimizations in the future.

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. You cannot use `gprof` on all systems if you specify this option, and you may have problems with debugging if you specify both this option and `-g`.

`-fbranch-target-load-optimize`
Perform branch target register load optimization before prologue / epilogue threading. The use of target registers can typically be exposed only during reload, thus hoisting loads out of loops and doing inter-block scheduling needs a separate optimization pass.

`-fbranch-target-load-optimize2`
Perform branch target register load optimization after prologue / epilogue threading.

`-fbtr-bb-exclusive`
When performing branch target register load optimization, don't reuse branch target registers within any basic block.

`-fstack-protector`
Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call `alloca`, and functions with buffers larger than 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits.

`-fstack-protector-all`
Like `-fstack-protector` except that all functions are protected.

`-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.

`--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 each case, the *value* is an integer. The allowable choices for *name* are:

`predictable-branch-outcome`

When branch is predicted to be taken with probability lower than this threshold (in percent), then it is considered well predictable. The default is 10.

`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. The default value is 5.

`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. The default value is 8.

`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. The default value is 8.

`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 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. The default value is 20.

`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-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++). The default value is 400.

`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. The default value is 40.

`inline-min-speedup`
When estimated performance improvement of caller + callee runtime exceeds this threshold (in precent), the function can be inlined regardless 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. The default value is 2700.

`large-function-growth`
Specifies maximal growth of large function caused by inlining in percents. The default value is 100 which 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`. The default is 10000.

`inline-unit-growth`

Specifies maximal overall growth of the compilation unit caused by inlining. The default value is 30 which limits unit growth to 1.3 times the original size.

`ipcp-unit-growth`
Specifies maximal overall growth of the compilation unit caused by interprocedural constant propagation. The default value is 10 which limits unit growth to 1.1 times the original size.

`large-stack-frame`
The limit specifying large stack frames. While inlining the algorithm is trying to not grow past this limit too much. The default value is 256 bytes.

`large-stack-frame-growth`
Specifies maximal growth of large stack frames caused by inlining in percents. The default value is 1000 which 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.

For functions declared inline, `--param max-inline-insns-recursive` is taken into account. For functions not declared inline, recursive inlining happens only when `-finline-functions` (included in `-O3`) is enabled and `--param max-inline-insns-recursive-auto` is used. The default value is 450.

`max-inline-recursive-depth`
`max-inline-recursive-depth-auto`
Specifies the maximum recursion depth used for recursive inlining.

For functions declared inline, `--param max-inline-recursive-depth` is taken into account. For functions not declared inline, recursive inlining happens only when `-finline-functions` (included in `-O3`) is enabled and `--param max-inline-recursive-depth-auto` is used. The default value is 8.

`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 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). The default value is 10.

`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. The default value is 10.

`max-early-inliner-iterations`
`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`
`comdat-sharing-probability`
Probability (in percent) that C++ inline function with comdat visibility are shared across multiple compilation units. The default value is 20.

`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. The default value is 0.

`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. The default value is 10.

`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. The default value is 3.

`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. The default value is 30.

`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. The default value is 10.

`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. The default value is 2.

`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-level`
The maximum number of branches unswitched in a single loop.

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

`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.

`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.

`omega-max-vars`
The maximum number of variables in an Omega constraint system. The default value is 128.

`omega-max-geqs`
The maximum number of inequalities in an Omega constraint system. The default value is 256.

`omega-max-eqs`
The maximum number of equalities in an Omega constraint system. The default value is 128.

`omega-max-wild-cards`
The maximum number of wildcard variables that the Omega solver is able to insert. The default value is 18.

`omega-hash-table-size`
The size of the hash table in the Omega solver. The default value is 550.

`omega-max-keys`
The maximal number of keys used by the Omega solver. The default value is 500.

`omega-eliminate-redundant-constraints`
When set to 1, use expensive methods to eliminate all redundant constraints. The default value is 0.

`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. See option `-ftree-vect-loop-version` for more information.

`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. See option `-ftree-vect-loop-version` for more information.

`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-ws-permille`
A basic block profile count is considered hot if it contributes to the given permillage (i.e. 0...1000) of the entire profiled execution.

`hot-bb-frequency-fraction`
Select fraction of the entry block frequency of executions of basic block in function given basic block needs to have to be considered hot.

`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.

`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` 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-ratio`
`tracer-min-branch-ratio-feedback`
Stop forward growth if the best edge has probability lower than this threshold.

Similarly to `tracer-dynamic-coverage` two values are present, one for compilation for profile feedback and one for compilation without. The value for compilation with profile feedback needs to be more conservative (higher) in order to make tracer effective.

`max-cse-path-length`
The maximum number of basic blocks on path that CSE considers. The default is 10.

`max-cse-insns`
The maximum number of instructions CSE processes before flushing. The default is 1000.

`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. The default value is 100.

`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. The default value is 500.

`reorder-blocks-duplicate`
`reorder-blocks-duplicate-feedback`
Used by the basic block reordering pass to decide whether to use unconditional branch or duplicate the code on its destination. Code is duplicated when its estimated size is smaller than this value multiplied by the estimated size of unconditional jump in the hot spots of the program.

The `reorder-block-duplicate-feedback` is used only when profile feedback is available. It may be set to higher values than `reorder-block-duplicate` since information about the hot spots is more accurate.

`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. The default value is 100.

`max-sched-region-blocks`
The maximum number of blocks in a region to be considered for interblock scheduling. The default value is 10.

`max-pipeline-region-blocks`
The maximum number of blocks in a region to be considered for pipelining in the selective scheduler. The default value is 15.

`max-sched-region-insns`
The maximum number of insns in a region to be considered for interblock scheduling. The default value is 100.

`max-pipeline-region-insns`
The maximum number of insns in a region to be considered for pipelining in the selective scheduler. The default value is 200.

`min-spec-prob`
The minimum probability (in percents) of reaching a source block for interblock speculative scheduling. The default value is 40.

`max-sched-extend-regions-iters`
The maximum number of iterations through CFG to extend regions. A value of 0 (the default) disables region extensions.

`max-sched-insn-conflict-delay`
The maximum conflict delay for an insn to be considered for speculative motion. The default value is 3.

`sched-spec-prob-cutoff`

The minimal probability of speculation success (in percents), so that speculative insns are scheduled. The default value is 40.

`sched-spec-state-edge-prob-cutoff`
The minimum probability an edge must have for the scheduler to save its state across it. The default value is 10.

`sched-mem-true-dep-cost`
Minimal distance (in CPU cycles) between store and load targeting same memory locations. The default value is 1.

`selsched-max-lookahead`
The maximum size of the lookahead window of selective scheduling. It is a depth of search for available instructions. The default value is 50.

`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. The default value is 2.

`selsched-max-insns-to-rename`
The maximum number of best instructions in the ready list that are considered for renaming in the selective scheduler. The default value is 2.

`sms-min-sc`
The minimum value of stage count that swing modulo scheduler generates. The default value is 2.

`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. The default is 10000.

`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. The default value is 256.

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

`max-jump-thread-duplication-stmts`
Maximum number of statements allowed in a block that needs to be duplicated when threading jumps.

`max-fields-for-field-sensitive`
Maximum number of fields in a structure treated in a field sensitive manner during pointer analysis. The default is zero for `-O0` and `-O1`, and 100 for `-Os`, `-O2`, and `-O3`.

`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 cache, in bytes.

`l1-cache-size`
The size of L1 cache, in kilobytes.

`l2-cache-size`
The size of L2 cache, in kilobytes.

`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. By default, this 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.

`sccvn-max-scc-size`
: Maximum size of a strongly connected component (SCC) during SCCVN processing. If this limit is hit, SCCVN processing for the whole function is not done and optimizations depending on it are disabled. The default maximum SCC size is 10000.

`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. The default maxmimum number of queries is 1000.

`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. The default value of the parameter is 100.

`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. The default value of the parameter is 2000.

`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. The default value of the parameter is 2, which is the minimal number of registers needed by typical instructions. This value is the best found from numerous experiments.

`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. The default value of the parameter is 1000 for `-O1` and 10000 for `-O2` and above.

`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. The default value is 1000.

`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. The default is 12.

`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-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.

`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. The default value is 10 parameters. A variable whose value is unknown at compilation time and defined outside a SCoP is a parameter of the SCoP.

`graphite-max-bbs-per-function`
To avoid exponential effects in the detection of SCoPs, the size of the functions analyzed by Graphite is bounded. The default value is 100 basic blocks.

`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. The default value is 51 iterations.

`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.

`lto-partitions`
Specify desired number of partitions produced during WHOPR compilation. The number of partitions should exceed the number of CPUs used for compilation. The default value is 32.

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

`cxx-max-namespaces-for-diagnostic-help`
The maximum number of namespaces to consult for suggestions when C++ name lookup fails for an identifier. The default is 1000.

`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. The default value is 75. 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 stores paires that can be sunk. Set to 0 if either vectorization (`-ftree-vectorize`) or if-conversion (`-ftree-loop-if-convert`) is disabled. The default is 2.

`allow-load-data-races`
: Allow optimizers to introduce new data races on loads. Set to 1 to allow, otherwise to 0. This option is enabled by default unless implicitly set by the `-fmemory-model=` option.

`allow-store-data-races`
: Allow optimizers to introduce new data races on stores. Set to 1 to allow, otherwise to 0. This option is enabled by default unless implicitly set by the `-fmemory-model=` option.

`allow-packed-load-data-races`
: Allow optimizers to introduce new data races on packed data loads. Set to 1 to allow, otherwise to 0. This option is enabled by default unless implicitly set by the `-fmemory-model=` option.

`allow-packed-store-data-races`
: Allow optimizers to introduce new data races on packed data stores. Set to 1 to allow, otherwise to 0. This option is enabled by default unless implicitly set by the `-fmemory-model=` option.

`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. The default is 0.

`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.c` 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 will be considered when seeking a basis for a new straight-line strength reduction candidate.