CPP-not-lang

2022年11月3日 13:31

not language
beyond language

compiler option

__asm

jit, asmjit

mprotect

VirtualProtect

ABI

调度 原文是 scheduling，应该是 时序安排 的意思。

矢量化(vectorization) parallelization

==========

https://gcc.gnu.org/onlinedocs/gcc/index.html

Using the GNU Compiler Collection (GCC)

==========

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
3.11 Options That Control Optimization

这些选项 控制着 优化的各种分类。

没有 任何优化选项时，编译器的目标是 降低编译的cost，且 使得 调试(debugging) 产生 期

望的结果。语句是独立的：如果你 在语句间 的 断点上 stop 程序，那么你可以 为 任何变量 分配 新的值 或 修改 程序计数器 到 方法中的 任何其他 语句，并从 源码中获得 你期望的结果。

打开优化标记 使得 编译器 尝试 提高 性能 and/or 代码大小，代价是 编译期耗时 和 debug 的能力。

编译器 基于 对程序的knowledge 来执行优化。 一次编译 多个文件 并输出到 一个单独的 文件，允许 编译器 在对 它们中的 每个文件 进行编译时 使用到 编译器已知的 关于所有文件的信息。

不是 所有的 优化 通过 一个标记 来 直接控制。 只有本节列举的 优化 有 flag。

大部分优化 可以通过 -O0 或 在命令行中不设置 -O 等级 来完全禁止， 即使 指定了 单独的优化标记。 类似的，-Og 抑制了许多优化过程。

根据 目标的不同 和 GCC的配置的不同，可以在每个 -O 级别 启用一组 与这里列出的 稍有不同 的优化选项 集合。
你可以调用 GCC， 通过 -Q --help=optimizers 来 找到 每个 level 启用的 优化 集合。
例如

-O
-O1
进行优化。优化的编译 需要更多时间，对于大型函数 需要更多内存。
使用 -O， 编译器 尝试 降低 代码大小 和 执行时间(。。估计是指编译器的执行时间。)，不会执行 任何 消耗大量 编译时间的 优化。 (。。怎么判断 消耗大量，应该是实现设定好的，就是下面的才会开启)

-O 打开了下面的 优化选项
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions-called-once
-fipa-modref
-fipa-profile
-fipa-pure-const
-fipa-reference
-fipa-reference-addressable
-fmerge-constants

-fmove-loop-invariants
-fmove-loop-stores
-fomit-frame-pointer
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-pta
-ftree-scev-cprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-ter
-funit-at-a-time

。。。应该 -O 就是 -O1 。。。等价的。。

-O2

优化更多。GCC 执行几乎所有 支持的 优化，而不考虑 空间-时间 的权衡。和 -O 对比，这个选项 增加了 编译时间， 提升了 编译后 代码的 性能。

-O2 打开了 -O1 指定的 所有 优化标记， 还开启了 下列的 优化 标记
-falign-functions -falign-jumps
-falign-labels -falign-loops
-fcaller-saves
-fcode-hoisting
-fcrossjumping
-fcse-follow-jumps -fcse-skip-blocks
-fdelete-null-pointer-checks
-fdevirtualize -fdevirtualize-speculatively
-fexpensive-optimizations
-ffinite-loops
-fgcse -fgcse-lm
-fhoist-adjacent-loads
-finline-functions

-finline-small-functions
-findirect-inlining
-fipa-bit-cp -fipa-cp -fipa-icf
-fipa-ra -fipa-sra -fipa-vrp
-fisolate-erroneous-paths-dereference
-flra-remat
-foptimize-sibling-calls
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-freorder-blocks-algorithm=stc
-freorder-blocks-and-partition -freorder-functions
-frerun-cse-after-loop
-fschedule-insns -fschedule-insns2
-fsched-interblock -fsched-spec
-fstore-merging
-fstrict-aliasing
-fthread-jumps
-ftree-builtin-call-dce
-ftree-loop-vectorize
-ftree-pre
-ftree-slp-vectorize
-ftree-switch-conversion -ftree-tail-merge
-ftree-vrp
-fvect-cost-model=very-cheap

注意 在 使用 computed gotos 的 程序上 调用 -O2 时 开启的 -fgcse 会 有 warning

-O3

更进一步优化。-O3 打开 所有 -O2 指定的优化， 还会开启下面的 优化标记
-fgcse-after-reload
-fipa-cp-clone
-floop-interchange
-floop-unroll-and-jam
-fpeel-loops
-fpredictive-commoning
-fsplit-loops
-fsplit-paths
-ftree-loop-distribution
-ftree-partial-pre
-funswitch-loops
-fvect-cost-model=dynamic
-fversion-loops-for-strides

-O0

减少 编译时间，使得 debugging 产生 预期的结果。这是 默认设置

-Os

为size 优化。 -Os 激活 所有 -O2 ， 除了 下面的 会增加 code size 的 选项：

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

也激活 -finline-functions, 导致 编译器 调整 代码大小 而不是 执行速度，并执行 进一步的优化 来 降低 code size。

-Ofast

不考虑 严格的 标准编译。 -Ofast 启用 所有的 -O3 优化项。 也启用 不是对 所有 标准编译的 代码 有效的 优化。 它打开 -ffast-math, -fallow-store-data-races ，和特定于Fortran 的 -fstack-arrays 除非 -fmax-stack-var-size 被指定， 和 -fno-protect-parens。 关闭了 -fsemantic-interposition 。

-Og

优化 debug 体验。
-Og 应该是 标准的 edit-compile-debug 周期的 优化level，提供了 合理的 优化level，同时 保持了 快速编译 和 一个 良好的 debug 体验。
这是 比 -O0 更好的 选择， 来生成 用于debug 的 代码， 因为 一些控制 debug 信息的 编译器过程 在 -O0 中被禁用了。

就像 -O0，-Og 完全 禁用了 许多优化过程，因此 控制它们的 单个选项 不会起效。
-Og 启用了 -O1 的所有 优化标记， 除了 下面的 干涉了 debug 的 优化flag：

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

-Oz

优化 size 而不是 速度。 这可能 增加 执行的指令的数量 如果 这些指令 需要更少的字节 来 编码。
-Oz 的行为 类似 -Os 包括 启用 大部分 -O2 优化。

如果你使用 多个 -O 选项，带 或不带 level数字， 最后一个 选项 起效。

-f*flag* 格式的 选项 指明了 和机器无关的 标记。大部分flag 有 正反 形式； -ffoo 的反面形式 是 -fno-foo。在下面的表格中，只列出了一种格式 -- 你通常使用的那种。 你可以得到另一种格式，通过 remove 或 增加 ’no-’ 前缀

下面的选项 控制了 特定的优化。它们 要么被 -O 选项激活，要么 和 -O选项有关。
当你需要 对优化微调 的时候，可以使用 下面的 标记。 需要微调的 情况很少见。

-fno-defer-pop

对于 在 函数调用后 必须 pop 参数的 机器， 总是在 每个函数 返回后 立即 pop 参数。在 -O1 或更高 等级， -fdefer-pop 是默认值； 它允许 编译器 在 stack 上累积 多个 函数调用的 参数，并且 一次性 pop掉。

-fforward-propagate

在 RTL 上 执行 向前的 传播 阶段 (forward propagation pass)。 这个阶段 尝试 组合 2条语句，并检查 结果是否可以简化。如果 循环展开 (loop unrolling) 是可用的，2个 阶段 会被执行，第二个 阶段 会在 loop unrolling 后 执行。

在 -O1，-O2，-O3，-Os 中默认启用。

-ffp-contract=style

-ffp-contract=off 禁用 浮点表达式收缩

-ffp-contract=fast 启用 浮点表达式收缩，例如，如果 目标 有本地的支持，那么 形成融合的 乘-加 操作。

-ffp-contract=on 允许 浮点表达式收缩，如果 语言标准 允许。这个目前 没有实现，被视为 -ffp-contract=off

默认值是 -ffp-contract=fast

-fomit-frame-pointer

省略 函数中的 帧指针，如果 函数不需要的话。

这避免了 保存，设置，恢复 帧指针 的 命令， 在许多target上，它也提供了 一个 额外的 可用寄存器。

在一些 target上，这个标记 无效， 因为 标准 调用 序列 总是使用 帧指针，所以 无法忽略 帧指针。

注意，-fno-omit-frame-pointer 不能保证 所有的 函数中 都用到 帧指针。 一些target 总是 在 leaf 函数上 忽略 帧指针。

在 -O1 或更高 中 默认激活。

-foptimize-sibling-calls

优化 同级递归 和 尾递归 调用 (sibling and tail recursive call)

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

-foptimize-strlen

优化 各个 标准C的string函数 (如，strlen，strchr，strcpy)，和 它们的 _FORTIFY_SOURCE 对应函数(counterparts) 到 更快的替代函数

Enabled at levels -O2, -O3.

-fno-inline

除了 用 always_inline 标记的 函数外，不 内联展开 任何函数。 这是 不优化时的 默认设置。
对 单个 函数 标记为 noinline 来 避免 内联

-finline-small-functions
当函数 体 小于 预期的 函数调用代码时，集成 函数 到 它们的调用者中（ 这样，程序的 总体size 就变小了)。
编译器 试探性地 决定 哪些 函数 足够简单，值得集成到 调用者中。
这个 内联 会 应用到 所有的 函数， 即使 没有 声明 inline。

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

-findirect-inlining
通过 previous inlining， 发现 间接 内联 调用。只有当 通过 -finline-functions or -finline-small-functions 启用 inlining 时，这个 选项才有用。

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

-finline-functions
考虑所有 inlining 函数，即使没有被声明为 inline。 编译器 试探性地决定 哪些函数值得 以这种方式集成。
如果对于某个函数的 所有调用 都是 integrated，且 这个函数 被声明为 static，那么该函数 通常不会 作为 汇编代码 输出。

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

-finline-functions-called-once
考虑将 所有 只被调用一次的 static 函数 内联到 它们的调用者中，即使 它们没有被 inline 描述。 如果 对一个函数的 调用 被集成了，那么这个 函数本身 不会作为 汇编输出。

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

-fearly-inlining
在执行 -fprofile-generate 命令 和 真正的 inlining 阶段 之前， 内联 那些 被标记为 always_inline 的函数， 和那些 函数体 看起来 比 函数调用代码 小 的 函数。
这样做可以 显著 降低 分析成本， 并且 在有 大量 nested wrapper function 链 的程序上 内联速度更快。

Enabled by default.

。。fearly 。。。 early。。。

-fipa-sra
执行 聚合 的 过程间 标量替换 (perform interprocedural scalar replacement of aggregates)， 移除 未使用的 参数， 使用 值传递形参 替换 址传递形参。

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

-finline-limit=n
默认下，GCC 限制了 能被 inline 的 函数的 size。 这个标记 允许 对 这个限制 进行粗略的(coarse) 控制。 n 是 可以内联的 函数的 伪指令 的数量。

inlining 被许多参数 控制， 可以通过 --param name=value 来指定 特定的参数。
-finline-limit=n 设置了 如下的参数
max-inline-insns-single 被设置为 n/2
max-inline-insns-auto 被设置为 n/2
查看后续文档 获得 更多的 可以独立设置的参数，和 参数的默认值。

注意，不带=n 的 -finline-limit 会导致 默认的行为
注意，在这个上下文中，伪指令 表示 函数大小的 抽象程度。 绝不代表 汇编指令的 数量。因此， 它的确切 含义 会 因版本 而不同。

-fno-keep-inline-dllexport
是 -fkeep-inline-functions 更 细粒度版本，仅适用于 被 dllexport 或 declspec 声明 的 函数。

-fkeep-inline-functions
在C中，将 声明为 inline 的 static 函数 发到(emit) 一个 对象文件中， 即使 函数已经 内联到 所有的 caller中。 这个开关 不影响 GNU C90 中使用 extern_inline 的函数，在 C++中，将所有 内联函数 发送到 一个 对象文件中。

-fkeep-static-functions
将 static 函数 发送(emit) 到 对象文件，即使 函数 没有被用到。

-fkeep-static-consts
将 声明为 static const 的 变量 emit，当 优化没有打开时， 即使 变量 没有被 引用。
GCC 默认启用这个选项。 如果 你希望 强制 编译器 检查 变量是否被引用，不管 优化是否打开，使用 -fno-keep-static-consts 选项。

-fmerge-constants
尝试跨 编译单元 合并 相同的 常量 (字符串，浮点数 常量)。
如果 汇编器 或 linker 支持，那么 就默认 启用 这个 选项 来优化编译。 使用 -fno-merge-constants 来 抑制 这种行为

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

-fmerge-all-constants

尝试 合并 相同的常量 和 相同的 变量。

这个选项 包含了 -fmerge-constants 。 除了 -fmerge-constants 的行为， 还考虑，例如：常量初始化的数组 或 使用 整型/浮点型类型初始化的常量。 像 C，C++ 这样的语言需要 每个变量 (包括 在 递归调用中 同一个变量的 多个实例) 有着 唯一的 地址，所以 使用这个 选项 会导致 不一致的(non-conforming(不合规，非标准)) 行为

-fmodulo-sched

在第一个 scheduling 阶段前 立即执行 swing modulo scheduling。 这个阶段 查看 最内部的 loop 并且 通过 重叠不同的迭代(overlapping different iterations) 来 重排序它们的执行

-fmodulo-sched-allow-regmoves

在 允许 寄存器移动(register move) 的情况下，执行更积极的 基于SMS 的 modulo scheduling。 通过设置这个 flag，一些 反依赖边(anti-dependences edges) 被移除，这个会触发 生成 一个 基于生命周期分析(life-range analysis) 的 reg-moves。这个选项只有在 -fmodulo-sched 启用时 才有效。

-fno-branch-count-reg

禁用 一个优化阶段，这个阶段中 在 count register 上 扫描 "decrement and branch" 指令 的 使用 次数， 而不是 在 寄存器上 进行递减 的指令序列 ，将其 和 0 比较，然后 根据结果 选择 分支。

Disable the optimization pass that scans for opportunities to use “decrement and branch” instructions on a count register instead of instruction sequences that decrement a register, compare it against zero, and then branch based upon the result.

这个选项 只在 支持 此类指令 的 架构 (包括 x86，PowerPC，IA-64，S/390) 上有意义。

注意，-fno-branch-count-reg 选项 不会 把 其他优化阶段 引入的 指令流中的 decrement and branch 指令 移除。

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

-fno-function-cse

不将 函数地址放到 寄存器中；使得 每条 调用 常量方法(constant function) 的指令显式包含了 函数的 地址。

这个选项 会导致 代码低效， 但是 不使用这个 选项的话，执行的优化可能会 混淆 一些改变 汇编输出的 hack

默认是 -ffunction-cse

-fno-zero-initialized-in-bss

如果目标支持 BSS section， GCC 会默认 将 初始化为0的变量 放到 BSS。这个可以 解约 最终的代码 所用空间。

这个选项 关闭了 这种行为，因为 一些 程序 显示依赖于 进入 data section 的变量，所以 生成的 可执行文件 可以 找到 那个section 的 beginning， and/or 基于that 做出一些 假设。

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

-fthread-jumps

执行优化 来 检查 查看 是否 将branch jump 到 一个位置，这个位置 是 第一个找到的另一个比较的位置， 如果是，第一个branch 被重定向到 第二个branch 或 紧随它的 下一个点，取决于 条件是 true 还是 false

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

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

-fsplit-wide-types

当 使用一个 会占用 多个寄存器的 类型时， 比如 在32位系统上 使用 long long，将寄存器分开 并 独立分配。 这个 通常会 为这些 类型 生成 更好的代码， 但是 debug 更难。

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

-fsplit-wide-types-early

early 完全拆分 wide 类型。 这个选项无效，除非 -fsplit-wide-types 是 on。

在某些target 上 是默认的

-fcse-follow-jumps

在 通用子表达式消除 (CSE, common subexpression elimination)中， 当 jump 的目标对于 任何其他路径 都不可达时，扫描 jump 语句。 例如，当CSE 遇到 带 else 的 if 语句， 当条件是 false 时， CSE 会 follow jump。

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

-fcse-skip-blocks

类似于 -fcse-follow-jumps，但是 导致 CSE 跟随 那些conditionally 跳过块 的 jump。当 CSE 遇到 一个 不带 else 的 if 时，-fcse-skip-blocks 导致 CSE 跟随 跳过 if 的body 的 jump。

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

-frerun-cse-after-loop

在 loop 优化 执行后， 再次执行 通用子表达式消除 CSE

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

-fgcse

执行一个 全局 CSE 阶段。这个阶段 也执行 全局 常量 和 复制 扩散 (global constant and copy propagation)

注意：当 编译一个 使用了 计算的goto(computed gotos) 的程序时，一个GCC扩展，你可能 获得 更好的 运行时 性能 如果你 通过 增加 -fno-gcse 到 命令行 来 禁用 全局 CSE 阶段。

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

-fgcse-lm

当这个启用时， 全局CSE 尝试 将 仅由 存储 终止的 load 移动到 自身中。(move loads that are only killed by stores into themselves)。这允许 一个 包含 load/store 序列 的 loop 被转换为 loop外的 load 加上 loop内的 copy/store。

Enabled by default when -fgcse is enabled.

-fgcse-sm

启用这个标记时，在 全局 CSE 后 执行 一个 store motion 阶段。 这个阶段 尝试 将 store 移动到 loop 外。 和 -fgcse-lm 结合使用时， 包含 load/store 序列的 loop 能被 转为 loop 前 load 加上 loop 后 store。

Not enabled at any optimization level.

-fgcse-las

这个标记启用时，全局CSE 阶段 消除 在一个内存地址 上store 后，对这个内存地址 的后续的 冗余load (包括 部分 和 全部 冗余)

Not enabled at any optimization level.

-fgcse-after-reload

启用时，在 reload 后 执行一个 冗余load消除 阶段。 这个阶段的目的是 消除 冗余的

溢出(spilling)。

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

-faggressive-loop-optimizations
这个选项 告诉 loop 优化器 去 使用 语言限制(language constraints) 来 得出 loop 的迭代的 次数 界限。 这个假设 loop 代码 不会 导致 未定义的行为，如 导致 有符号整数 的 溢出 或 越界的数组访问。 loop迭代次数的界限 用于 指导 loop 展开和剥离(unrolling and peeling) 和 循环退出 测试优化。

默认启用

-funconstrained-commons
这个选项 告诉 编译器，在 公共块 (common block) (如 Frotran) 中 声明的变量 稍后可能会被 较长的尾部数组(longer trailing array) 覆盖。 这防止 一些 依赖于 已知的数组 边界的 优化。
。。C++ 有这个吗？。。怎么区分 是什么语言的。。

-fcrossjumping
执行 cross-jumping (交叉跳转) 转换。这个 转换 统一了 等效代码 并 节约了代码长度。 生成的代码 可能比 没有 交叉跳转 的 执行得更好， 也可能不会。

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

-fauto-inc-dec
将 地址的增量和减量 与 内存访问 结合。如果架构不支持这种命令，则跳过 这个阶段。

Enabled by default at -O1 and higher on architectures that support this.

-fdce
在 RTL 上 执行 DCE (dead code elimination)

Enabled by default at -O1 and higher.

-fdse
在 RTL 上执行 DSE (dead store elimination)

Enabled by default at -O1 and higher.

-fif-conversion
尝试将 条件跳转 转换为 无分支的等效。这包括 使用 条件移动，min，max，设置标记，abs，和一些 可以通过 标准算术实现的 技巧。 在(可以使用conditional execution的)

芯片(chip)上 使用 conditional execution 是通过 -fif-conversion2 控制的。

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

-fif-conversion2
使用 conditional execution (如果可用) 来转换 conditional jump 为 无分支的等价。

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

-fdeclone-ctor-dtor
C++ ABI 要求 构造器和析构器 有多个 入口点：一个用于 基本子对象base subobject，一个用于完整对象complete object， 一个用于稍后 调用 delete运算符 的虚拟析构函数。对于 有 虚拟base 的 架构，base 和 complete 用的 构造器 是用 clone 实现的，这意味着 函数 有2个 副本。 通过这个 选项，base 和 complete 被改为 调用公共实现。

Enabled by -Os.

-fdelete-null-pointer-checks
假设 程序 不能安全地 对 空指针 反引用，并且 在地址0 处 没有 代码或数据。 这个选项允许 简单常量折叠 优化 在所有 优化level。 此外，GCC 的其他优化阶段 使用这个标记 来控制 全局 数据流 分析，从而消除 对空指针的 无用检查；这些 假设 对地址0 的内存访问 总是 导致 trap(陷阱)，所以 如果 在 指针被反引用后 进行检查，它不可能为 null。

注意，在一些环境中 这个假设不成立。 使用 -fno-delete-null-pointer-checks 来禁用这个优化。

This option is enabled by default on most targets
Nios II ELF 上，默认关闭
AVR 和 MSP430， 完全不可用

在不同优化等级上 的 使用数据流信息 的 阶段 被独立 启用。

-fdevirtualize
尝试 将 虚拟函数的 call 转换为 直接call。这个 在 过程内(procedure) 和 过程间(interprocedurally) 都可以 完成，作为 间接inlining(indirect inlining) (-findirect-inlining) 和 过程间常量传播(interprocedural constant propagation) (-fipa-cp) 的一部分。

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

-fdevirtualize-speculatively
尝试将 虚拟函数的 call 转换为 推测性的直接call。 基于 对类型继承图 的分析，确定

给定call 的可能 target 集合。 如果 集合较小，最好大小为1，则将 call 变为 一个在直接 和 间接 call 之间的 条件选择。 推测性的call 允许 更多的 优化，比如 内联。当它们看起来 对 进一步的优化 无用时，它们被转换为 原始的形式。

-fdevirtualize-at-ltrans
在本地转换模式下 允许 link-time 优化器 时， 流式 传输 去虚拟化 所需的 信息。 这个选项 启用 更多的 去虚拟化 但 显著地 增加了 流式传输的数据 的 大小。 因此 它默认 禁用。

-fexpensive-optimizations
执行 一些 相对昂贵的 小优化

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

-free
尝试 删除 冗余的 扩展指令。对于 x86-64 架构特别有用，这个架构在 写入 64位寄存器的 低 32位的half 后， 在 64位寄存器中 隐式进行 0扩展。

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

-fno-lifetime-dse
在C++中，对象的值 只受 在它的生命周期 内的 修改的 影响：当 构造器开始时，对象有不确定的值， 当对象被销毁时，对象生命周期内的 任何修改 都是无效的。通常 死区消除(dead store elimination) 利用了这一点。
如果你的代码 依赖的 对象存储的 值 在对象生命周期后 依然存在， 你可以使用这个flag 来 禁用这个优化。
要在 构造器 启动前 保留 存储(例如，因为 你的 new操作 清空了 对象存储)， 但 在析构后 依然 视 对象为死的， 你可以使用 -flifetime-dse=1 。 默认可以 可以通过 -flifetime-dse=2 来显式选择。 -flifetime-dse=0 等价于 -fno-lifetime-dse 。

-flive-range-shrinkage
尝试 降低 寄存器压力 通过 缩小寄存器起效范围 (register live range shrinkage)。这对于 有 小或中等 数量的 寄存器 集合 的 快速处理器 有用。

-fira-algorithm=algorithm
为 集成寄存器 分配器 使用 指定的 着色算法。
algorithm参数 可以是 'priority'，这个指定了 Chow's priority coloring。 'CB' 指定了 Chaitin-Briggs coloring。 CB 不是所有的 架构都实现，但是 对于 支持CB 的架构，CB 是默认值，因为它 生成 了更好的 代码。

-fira-region=region
为 集成寄存器分配器 使用 指定的 区域。 region参数可以是 下面的 之一

'all'
使用所有loop 作为 寄存器分配器区域。 这个 对于 有 小的 和/或 不规则 的寄存器 集合 有 最好的结果

'mixed'
使用所有 loop 作为 区域，除了 那些 寄存器压力 较小 的 loop 。 对于 大多数架构 的 大多数case 有 最好的结果， 并且 如果 编译时 是 为了速度而优化的话(-O, -O2, …)， 这个 默认启用。

'one'
使用 所有函数 作为 单一的 区域。这个通常 产生 更小size 的代码，在 -Os，-O0 中默认启用。

-fira-hoist-pressure
在 code hoisting 阶段，使用 IRA 来 评估 寄存器压力，以决定 提升表达式(hoist expression)。 这个选项通常 生成 更小的 代码， 但会 降低 编译器速度。

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

-fira-loop-pressure
使用 IRA 来 评估 loop 中 寄存器压力 来 决定 是否移动 loop 的 不变量。 这个选项通常 导致 在 具有 大寄存器文件(large register files)(>= 32个寄存器) 的 机器上产生 更小 更快的 代码， 但是 降低 编译器 速度。

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

-fno-ira-share-save-slots
禁止 stack slot 的共享 用来保存 通过call 使用的 硬寄存器。 每个 硬寄存器 获得一个单独的 stack slot， 所以 函数 stack frame(栈帧) 更大
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
对 伪寄存器 禁用 stack slot 共享。每个 没有获得 硬寄存器的 伪寄存器 获得一个单独的 stack slot， 所以 函数的 栈帧 更大

-flra-remat
在 LRA中启用 CFG-sensitive rematerialization。LRA 不再 加载 溢出的 伪值，而是尝试重新计算值， 如果有利可图的话。
Enable CFG-sensitive rematerialization in LRA. Instead of loading values of spilled pseudos, LRA tries to rematerialize (recalculate) values if it is profitable.

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

-fdelayed-branch
如果 目标机器支持，尝试 重排序 指令 来 利用 延迟分支指令后 可用的 指令槽

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

-fschedule-insns
如果目标机器支持，尝试 重排序 指令 来 消除 由于 所需数据 不可用 而 导致的 执行暂停。 这有有助于 具有 缓慢的 浮点数 或 内存加载 指令 的机器， 允许 发出其他指令，直到 需要 加载 或 浮点指令 的结果。

Enabled at levels -O2, -O3.

-fschedule-insns2
类似 -fschedule-insns，但 在 完成寄存器分配后，要求一个 额外的 指令调度 (instruction scheduling) 阶段。 这在 寄存器数量相对较少 且 load指令 需要 一个以上周期 的 机器上 特别有用。

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

-fno-sched-interblock
禁用跨基本块 的指令调度，通常在 寄存器分配前的 调度 时启用，即使用 -fschedule-insns 或 在 -O2及更高

-fsched-pressure
在 寄存器分配前 启用 寄存器压力敏感 insn 调度。这只有在 寄存器分配 前 进行调度才有意义，例如使用 -fschedule-insns 或 -O2及更高。
使用这个选项 能够提升 产生的代码 和 降低 它的size 通过 防止 寄存器压力 增加 超过 可用硬件寄存器的 数量 和 寄存器分配中的 后续溢出(subsequent spills in register allocation.)。

-fsched-spec-load
允许 一些load指令 的 推测行为。 这只有在 寄存器分配前 调度才有意义，例如 with -fschedule-insns or at -O2 or higher.

-fsched-spec-load-dangerous
允许 更多loadl指令 的 推测动作。 只有在 寄存器分配前 调度 才有意义，例如 with -fschedule-insns or at -O2 or higher.

-fsched-stalled-insns
-fsched-stalled-insns=n
定义 在第二次调度 阶段 中， 有 多少 insns (如果有) 可以被 提前 从 暂停insn队列

移动到 就绪列表。
-fno-sched-stalled-insns 意味着 没有 insn 被 提前移动。
-fsched-stalled-insns=0 意味着 不限制 提前移动的 insn 的数量。
-fsched-stalled-insns ，不带参数 等价于 -fsched-stalled-insns=1

-fsched-stalled-insns-dep
-fsched-stalled-insns-dep=n
定义 需要检查 多少 insn groups(cycles) 来确定 是否存在 对暂停 insn 的 依赖关系，这个暂停insn 是 提前从 暂停(stalled) insn队列 中移除的 候选项。这 仅在 第二次 调度过程中，并且 仅在 使用 -fsched-stalled-insns 时有效。
-fno-sched-stalled-insns-dep 等价于 -fsched-stalled-insns-dep=0
-fsched-stalled-insns-dep 不带参数 等价于 -fsched-stalled-insns-dep=1

-fsched2-use-superblocks
在寄存器分配后 进行调度时，使用 超级块(superblock) 调度。这允许 跨越基本块 边界的 行为，从而 加快 调度。 这个 选项是 实现性的，因为 并非 GCC 使用的 所有 机器描述 都 对 CPU 建模得足够紧密，以避免算法的 不可靠结果。

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

-fsched-group-heuristic
在 调度器中 使用 组启发式。这种启发式 支持属于 调度组的 指令。

当 调度 被启用时(如 -fschedule-insns or -fschedule-insns2 or at -O2 or higher.)，这个被 默认启用。

-fsched-critical-path-heuristic
在 调度器中 启用 关键路径 试探。 这种启发式 方法 有利于 关键路径上的 指令。

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
在 调度器中 使用 推测指令试探。 这个启发式 方法 倾向于 依赖性较弱的 推测指令。

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
在 调度器中 使用 rank 试探，这种试探方法 倾向于 使用 更大size或频率 的 基本块的 指令。

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
在 调度器中 启用 last-instruction(最后命令) 试探。 这种试探方法 倾向于 较少依赖于 调度的 最后一条指令 的 指令。

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
在 调度器 中 启用 dependent-count 试探。 这种试探 倾向于 有更多指令依赖于它的指令。

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
在传统调度前 执行 modulo 调度。 如果一个 loop 是 modulo 调度的，后续的 调度 阶段 可能 修改 它的 调度。 使用这个 选项 来控制 that 行为。

-fselective-scheduling
调度 指令 使用 selective scheduling 算法。selective scheduling 代替 第一个 调度阶段 运行。

-fselective-scheduling2
调度指令使用selective scheduling算法， selective scheduling 代替 第二个 调度 阶段。

-fsel-sched-pipelining
在 selective scheduling 期间中 启用 最内层loop 的 software pipelining。

This option has no effect unless one of -fselective-scheduling or -fselective-scheduling2 is turned on.

-fsel-sched-pipelining-outer-loops
当在 selective scheduling 期间 进行 pipelining loop 时，也会 外层loop pipeline

This option has no effect unless -fsel-sched-pipelining is turned on.

-fsemantic-interposition

。。。差不多 1/5 。。。
。。g　https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
。。下面是 几个有 代码片段的 option。。

-ftree-loop-distribution

loop切分， 提升 大loop体 的 cache 性能，并允许 进一步的 loop 优化，比如 并行 或 矢量化 (parallelization or vertorization)

例如

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

被转换为

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

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

-ftree-loop-distribute-patterns

从loop 中拆分代码， 如果 这些代码 满足： 功能 可以通过 调用 库 来完成。

This flag is enabled by default at -O2 and higher, and by -fprofile-use and -fauto-profile.

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

被转换为

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

初始化loop 会被 转换为 一个 memset 为 0 的 call。

This flag is enabled by default at -O3. It is also enabled by -fprofile-use

and -fauto-profile.

-floop-interchange

执行 内外层 loop 交换。 这个 flag 可以提升 多层循环 的 cache 性能， 且 允许 进一步loop 优化，比如 平行向量化处理(vectorization)

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

被转换为

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

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

。。这个提升了什么？ 感觉只是 [j] 能顺序遍历。 但是 原先 [k] 也能顺序遍历的，改没了啊。 除非 预先加载了 c[i] 这一行 到 cache (指 寄存器之下，内存之上的 CPU cache组件)
。。是的。 "CPU从内存读取数据到CPU Cache的过程中,是一小块一小块来读取"
。。但是 a[i] 这一行 就不能用cache 了啊。 应该是 写的 代价 高于 读。

-fstrict-aliasing

允许 编译器 假设 适用于当前编译的 语言的 最严格 的 别名规则。对于 C C++，这个会激活 基于 表达式类型的 优化。特别是， 假设 一种类型的对象 永远不会 和 不同类型的对象 在 驻留在相同的地址，除非类型几乎相同。
例如，unsigned int 可以 别名为 int， 但是不能 别名为 void* 或 double。一个 character 类型可以 别名 任何其他类型

特别留意 类似下面的 代码

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

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

从一个 不同于 最近写 的 union的member 的 member上读取 是很常见的。 即使使用了 -fstrict-aliasing，也允许 类型双关 (type-punning)，前提是 通过 union 类型 来访问内存。 因此，上面的 代码能工作。 但是下面的 代码可能不能：
。。。？ 上面的 代码 f() 会返回什么？ 不是 默认的 0 ？ 但是 C应该没有默认值吧。 难道是 返回 3 ？？？？

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

类似的，通过 地址访问，强转 结果指针 和 反引用结果 会导致 未定义行为，即使 强转 使用了 union 类型，例如

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

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

-fversion-loops-for-strides

如果 loop 遍历时 使用了 一个变量 作为 步长，那么 创建另一个版本的循环： 会先 判断 步长是否为1。

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

变成

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

。。 i+=stride 行不行？

这对 Fortran 中的 假定形状数组(assumed-shape array) 很有用， 它允许 在 假定连续访问的 情况下 更好地 矢量化。

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

-fsection-anchors

尝试 降低 符号地址计算(symbolic address calculation) 的次数， 通过 使用 共享的"锚" 符号 来定位 附近的 对象。

这个转换可以帮助 减少 GOT 实体的数量 和 GOT访问 的数量， 在一些target上。

例如，下面的 foo 方法：

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

通常计算 3个变量的地址，但是 如果 你使用 -fsection-anchors 来编译它，它会通过一个 公共锚点 来访问 变量。 类似于下面的 伪代码(在C中是非法的)：

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

Not all targets support this option.

==========

==========

https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html#x86-Options

==========

==========

https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Dialect-Options.html#C_002b_002b-Dialect-Options

==========

https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html#C-Dialect-Options

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========