SUSE

SUSE Best Practices

Development Tools

# Advanced Optimization and New Capabilities of GCC 15

SUSE Linux Enterprise Server 16.0 and later

Martin Jambor, Toolchain Team Lead (SUSE)
Jan Hubička, Toolchain Developer (SUSE)
Richard Biener, Toolchain Developer (SUSE)
Michael Matz, Toolchain Developer (SUSE)
Venkataramanan Kumar, PMTS Software System Design Eng (AMD)
Kim Naru, Engineering Manager (AMD)

SUSE

The document at hands provides an overview of GCC 15 as the default compiler for users of SUSE Linux Enterprise 16.0. It focuses on the important optimization levels and options **Link Time Optimization (LTO)** and **Profile Guided Optimization (PGO)**. Their effects are demonstrated by compiling the SPEC CPU benchmark suite for AMD EPYC 8635P Processor.

**Disclaimer:** Documents published as part of the SUSE Best Practices series have been contributed voluntarily by SUSE employees and third parties. They are meant to serve as examples of how particular actions can be performed. They have been compiled with utmost attention to detail. However, this does not guarantee complete accuracy. SUSE cannot verify that actions described in these documents do what is claimed or whether actions described have unintended consequences. SUSE LLC, its affiliates, the authors, and the translators may not be held liable for possible errors or the consequences thereof.

# Contents

1 Overview 4

2 Compilers available in SUSE Linux Enterprise Server 16 5

3 Optimization levels and related options 11

4 Taking advantage of newer processors 13

5 Link Time Optimization (LTO) 14

6 Profile-Guided Optimization (PGO) 18

7 Performance evaluation: SPEC CPU 2017 19

8 Legal notice 40

9 GNU Free Documentation License 41

# 1 Overview

The first release of the GNU Compiler Collection (GCC) with the major version 15, GCC 15.1, took place in May 2025. In June of the same year, the openSUSE Tumbleweed Linux distribution began using this compiler to build its packages. GCC 15.2, with fixes to over 100 bugs, was released in August. When SUSE Linux Enterprise Server 16.0 was released in November 2025, this compiler was provided for building user applications and libraries. This new version introduces many new features. These include implementation of parts of the most recent versions of various language specifications (particularly C23, C++23, and C++26), along with their extensions (such as OpenMP and OpenACC). Like all new major releases, GCC 15 introduces support for new capabilities of a range of computer architectures and generic improvements in code optimization. Although GCC 15 is the third major release supporting AMD CPUs based on the Zen 5 core, both general and targeted optimizations allow it to better leverage processor capabilities.

This document provides an overview of GCC 15. It focuses on selecting appropriate optimization options for your application and stresses the benefits of advanced modes of compilation. First, we describe the optimization levels the compiler offers, and other important options developers often use. We explain when and how you can benefit from using **Link Time Optimization (LTO)** and **Profile Guided Optimization (PGO)** builds. We also detail their effects when building a set of well-known CPU-intensive benchmarks. Finally, we look at how these perform on AMD Zen 5 based AMD EPYC 8635P Processors.

# 2 Compilers available in SUSE Linux Enterprise Server 16

Developers of user-space applications and libraries can use the supported GNU Compiler Collection (GCC) C, C++ and Fortran compilers. Compilers for other languages, cross-compilers and accelerator offloading compilers are not available from standard repositories, but developers can install them from Package Hub with community support.

```
sles16:~ # gcc --version
gcc (SUSE Linux) 15.2.0
Copyright (C) 2025 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Unlike the previous version, SUSE Linux Enterprise Server 16 has a different lifecycle for a number of components including compilers. The initial compiler major version in versions 16.0 and 16.1 is GCC 15. Later SUSE Linux Enterprise Server releases will introduce the tick-tock model:

- Each future even-numbered minor release of SUSE Linux Enterprise Server introduces a new major GCC version as the default compiler. This version is supported throughout long-term support (LTS) for that minor release and the subsequent minor release. It will also be supported also for the next SUSE Linux Enterprise Server minor version. For example, GCC 17 is introduced in SUSE Linux Enterprise Server 16.2. It is supported until the end of LTS for both SUSE Linux Enterprise Server 16.2 and 16.3.
- Each future odd-numbered minor release of SUSE Linux Enterprise Server (the tock release) introduces a new non-default major version of GCC. This version will not be the default one. To use this version, you need to explicitly invoke the binaries `gcc-x`, `g++-x` and `gfortran-x`. These non-default versions are supported for 24 months.

### Note: Comparison with SUSE Linux Enterprise Server 15

In SUSE Linux Enterprise Server 15, the one default *system compiler* was GCC 7 across the entire lifecycle and all service packs. Additionally, an *application compiler* was also available through the *Development Tools Module*, but it was never the default and its major version updated annually. The new scheme in SUSE Linux Enterprise Server 16 offers developers on this platform access to a more recent default compiler supported throughout the lifetime of each corresponding minor versions. Compilers added in odd-numbered minor releases enable developers to use the latest features and hardware support.

The basic C and C++ runtime libraries are upgraded to newer GCC major versions across all SUSE Linux Enterprise 16 minor versions. Therefore, applications built with non-default "tock" compilers can run on fully-upgraded systems without requiring the non-default compiler.

In this document, GCC 15 refers to any minor version within major version 15, whereas GCC 15.2 refers specifically to that particular version. In most cases, these terms are interchangeable.

## 2.1 Compiler for kernel modules and package rebuilds

The operating system kernel of SUSE Linux Enterprise Server 16.0 was built using GCC 13. To build a kernel module for it, you should use the corresponding compiler from package `gcc13`. Nevertheless, building kernel modules is outside of the scope of this article. For more information, refer to the Kernel Module Package Manual for SUSE Linux Enterprise Server 16.0 (https://documentation.suse.com/sbp/systems-management/html/KMP-Manual-SLES16/index.html).

Most, but not all, other packages for SUSE Linux Enterprise are built using GCC 13. To perform a build in the same fashion as SUSE did when producing a particular package, use the C compiler referenced above or the corresponding packages for other languages from Package Hub. The correct compiler version is selected automatically when using the Open Build Service to build a package. However, this version might not be GCC 15.

### Important: Support status of GCC 13

Only the C compiler from GCC 13 is supported, and only to build kernel module packages in accordance with SUSE policies. Other compilers from GCC 13 receive *community support basis through Package Hub*. Using the C compiler to build user-space applications or libraries is only covered with *commercially reasonable support*.

## 2.2 New features of GCC 15 compared to previous releases

Each major version of GCC introduces new features, new language versions, support for the most modern processors, and performance and other improvements. GCC 15 is the first compiler to support the C23 standard, published in 2024 as ISO/IEC 9899:2024. This lets you take advantage of all included new language features. This standard, with GNU extensions, is now also the default when no standard is specified on the command line.

The default, and latest fully implemented, C++ standard is ISO/IEC 14882:2017, commonly called C++17. This version, with GNU extensions, is also the default when no standard is specified on the compiler command line. The compiler implements almost all features from C++20 (ISO/IEC 14882:2020), C++23 (ISO/IEC 14882:2024), and most features expected in C++26. However, consider these features experimental. Use them with appropriate caution and avoid linking code produced by different compilers. Note that C++ *modules* are only partially implemented [1] and require the source file to be compiled with the `-fmodules-ts` option. To check the implementation status of C++ features in the compiler or the standard library, consult the following resources:

- C++ Standards Support in GCC (https://gcc.gnu.org/projects/cxx-status.html)
- The GNU C++ Library Manual (https://gcc.gnu.org/onlinedocs/gcc-15.2.0/libstdc++/manual)

Advances in supporting new language specifications are not limited to C++. The Fortran compiler is also continuously improved. If you use `OpenMP` or `OpenACC` extensions for parallel programming, the compiler supports many features from newer versions of these standards.

GCC 15 can generate code for many recent processors not supported by previous major versions. Such a list of processors is too long to include in this document. Nevertheless, *Section 7, "Performance evaluation: SPEC CPU 2017"* describes code optimization for AMD EPYC 8635P Processors that are based on AMD Zen 5 cores. GCC supports this core architecture starting with versions 14.1 and 13.2. Earlier versions do not support this core architecture and cannot optimize for it.

Finally, the general optimization pipeline of the compiler has also significantly improved over time. For more details about improvements in versions of GCC 8 through 15, visit the links provided in the section of this chapter.

## 2.3 Potential issues with the recent versions of the GCC compiler

New versions of a compiler can sometimes behave differently and can introduce issues not present with the system compiler. The upstream project maintains extensive lists of such problems encountered by users which are referenced in the following section. This section highlights the most common pitfalls.

Starting with GCC 14, the C compiler actually issues errors constructs disallowed since the 1999 ISO C standard. GCC 13 and earlier versions generated only warnings for these constructs:

- Implicit int types (`-Werror=implicit-int`)
- Implicit function declarations (`-Werror=implicit-function-declaration`),

**1** Proposals P1766R1 and P1815R2 are only implemented in GCC 16.

- Wrong or misspelled function prototypes (`-Werror=declaration-missing-parameter-type`),
- Incorrect uses of the return statement (`-Werror=return-mismatch`),
- Using pointers as integers and vice versa (`-Werror=int-conversion`), and
- Type mismatches of pointer types (`-Werror=incompatible-pointer-types`)

We highly recommend fixing any of the above issues if you encounter them in your code. They are a frequent source of bugs, portability and even security issues. For more information and common solutions, see the "Porting to GCC 14" document referenced below. For code written in a C standard prior to C99 (1999 ISO standard), use the `-std=gnu89` or `-std=c89` option to allow these constructs. If your code uses features of the C99 standard or a later one, and for some reason cannot be fixed, convert specific errors to warnings using the `-Wno-error=` option. Alternatively, use a new compiler switch `-fpermissive` to apply this behavior to all new errors.

### Note: Impact on build environment probing

Many code snippets (also called *probes*) generated by `autoconf` to discover the availability of various features trigger a compile error when a feature is missing. New compiler errors can cause build failures in code that previously compiled. This can result in available features being silently disabled (despite the fact they are actually available). `autoconf` has supported C99 compilers since version 2.69 in its generic, core probes. However, earlier versions or very specific probes might rely on C features removed in C99 and thus fail with GCC 14 or newer. To verify there are no unexpected differences, compare generated files such as `config.log`, `config.h` and other generated files using **`diff`**.

In addition, the C compiler in GCC 15 changes the default language version for C compilation from `-std=gnu17` to `-std=gnu23` which can lead to the following new compilation errors:

- There are new keywords including `bool`, `true`, `false`, `nullptr`, and `thread_local`. Code that uses these for identifiers is no longer accepted.
- Function declarations without parameters have changed meaning in C23. In earlier standards, an empty parameter list conveyed no parameter information. in C23, it explicitly indicates that the function takes no parameters. This change also affects function pointer type definitions, which can lead to type mismatches.

Again, we recommend porting the code to the new standard. Alternatively, you can use the `-std=` option to select an older version of the standard to restore the old behavior.

Users of the C compiler switching from GCC 9 or earlier should also be aware that newer versions default to `-fno-common` for performance reasons. This means a linker error will now be reported if the same variable is defined in two C compilation units. This can occur if two or more `.c` files include the same header file. If the header declares a variable without the `extern` keyword when doing so, this inadvertently results in the creation of multiple definitions. If you encounter such an error, you need to add the `extern` keyword to the declaration in the header file and define the variable in only a single compilation unit. Alternatively, you can compile your project with an explicit `-fcommon` if you are willing to accept that this behavior is inconsistent with C++ and may incur speed and code size penalties.

Users compiling C++ sources should take note that `g++` version 11 and later default to `-std=gnu++17`, the C++17 standard with GNU extensions. This means that dynamic exception specifications are no longer allowed. Users can annotate functions with `noexcept` as appropriate. Moreover, some C++ Standard Library headers no longer include headers they do not depend on. You may need to explicitly include `<limits>`, `<memory>`, `<utility>` or `<thread>`.

The final issue emphasized here is that the C++ compiler in GCC 8 and later now assumes that no execution path in a non-void function reaches the end of the function without a return statement. This means it is assumed that such code paths will never be executed, and thus they will be eliminated. You should therefore pay special attention to warnings produced by `-Wreturn-type`. This option is enabled by default and indicates which functions are affected.

## 2.4 References to more information on changes in recent GCC versions

This section provides a comprehensible list of links to the description of new features and possible issues with each major version of GCC from 8 to 15 for your reference.

- GCC 8
  - GCC 8 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-8/changes.html),
  - Porting to GCC 8 (https://gcc.gnu.org/gcc-8/porting_to.html),
- GCC 9

    - GCC 9 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-9/changes.html),
    - Porting to GCC 9 (https://gcc.gnu.org/gcc-9/porting_to.html),
- GCC 10
    - GCC 10 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-10/changes.html),
    - Porting to GCC 10 (https://gcc.gnu.org/gcc-10/porting_to.html),
- GCC 11
    - GCC 11 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-11/changes.html),
    - Porting to GCC 11 (https://gcc.gnu.org/gcc-11/porting_to.html),
- GCC 12
    - GCC 12 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-12/changes.html),
    - Porting to GCC 12 (https://gcc.gnu.org/gcc-12/porting_to.html),
- GCC 13
    - GCC 13 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-13/changes.html), and
    - Porting to GCC 13 (https://gcc.gnu.org/gcc-13/porting_to.html), and
- GCC 14
    - GCC 14 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-14/changes.html).
    - Porting to GCC 14 (https://gcc.gnu.org/gcc-14/porting_to.html).
- GCC 15

- GCC 15 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-15/changes.html).
- Porting to GCC 15 (https://gcc.gnu.org/gcc-15/porting_to.html).

# 3 Optimization levels and related options

GCC features an optimization pipeline controlled by approximately a hundred of command line options. Forcing users to decide whether to enable each option during compilation is impractical. Like all other modern compilers, GCC therefore introduces the concept of optimization levels: predefined optimization levels let you select from common configurations. Optionally, you can fine-tune individual options within a selected level if needed.

The default is to not optimize. You can specify this optimization level on the command line as `-O0`. It is often used when developing and debugging a project. This is usually accompanied by the command line switch `-g` so that debug information is emitted. Because no optimizations occur, no information is lost. Variables are not optimized away, and the compiler inlines only functions with attributes that explicitly require it. As a result, the debugger can reliably find everything it searches for in the running program and report the program state. However, the resulting code is large and slow. Thus, do not use this optimization level for release builds.

The most common optimization level for release builds is `-O2`. It attempts to optimize the code aggressively but avoids large compile times and excessive code growth. Optimization level `-O3` instructs GCC to optimize as much as possible. This might increase code size and compilation time. Note that neither `-O2` nor `-O3` affects the precision and semantics of floating-point operations. Even at the optimization level `-O3`, GCC implements math operations and functions according to the respective IEEE and ISO rules [2]. However, it allows floating-point expression contraction, for example when fusing an addition and a multiplication into one operation[3]. This often means that the compiled programs run markedly slower than necessary if such strict adherence is not required. The `-ffast-math` command-line option is a common way to relax rules governing floating-point operations. Listing all options enabled by `-ffast-math` is outside the scope of this document. However, if performance is a priority for your floating-point calculations, review the available options in the GCC manual.

---

**2** When the rounding mode is set to the default round-to-nearest (look up `-frounding-math` in the manual).

**3** See documentation of `-ffp-contract.`

If your project and the techniques you use to debug or instrument it do not depend on *ELF symbol interposition*, consider using `-fno-semantic-interposition` to improve performance. This option allows the compiler to inline calls and propagate information even when it would be illegal if a symbol changed during dynamic linking. Using this option to signal to the compiler that interposition will not occur significantly improves performance of some projects, including the Python interpreter.

The most aggressive optimization level is `-Ofast` which implies `-ffast-math` and `-fno-semantic-interposition` along with a few other options that disregard strict standard compliance. In GCC 15, this level also means optimizers may introduce data races when moving memory stores, which may not be safe for multithreaded applications. Additionally, the Fortran compiler can take advantage of associativity of math operations even across parentheses and convert big memory allocations on the heap to allocations on stack. The last mentioned transformation may cause the code to violate maximum stack size allowed by **`ulimit`** which is then reported to the user as a segmentation fault. To work around this issue, use **`ulimit -S`** with a sufficiently high limit, or **`ulimit -S unlimited`**. We often use level `-Ofast` to build benchmarks. It serves as a shorthand for options beyond `-O3` that often make benchmarks run faster. Most benchmarks are intentionally designed to run correctly even when these strict standard compliance rules are relaxed. While this level may not be safe for general use, you can review the enabled options and evaluate each individually to optimize performance.

If you feed the compiler with large machine-generated input, especially if individual functions are extremely large, compile time can become an issue even when using `-O2`. In such cases, use optimization level `-O1` to avoid running almost all optimizations with quadratic complexity. Finally, the `-Os` level directs the compiler to aggressively optimize for the size of the binary.

### Note: Optimization level recommendation

We usually recommend using `-O2`. This is the optimization level we use to build most SUSE and openSUSE packages. At this level, the compiler makes balanced size and speed trade-offs when building a general-purpose operating system. However, we suggest using `-O3` if your project is compute-intensive and either small or an important part of your actual workload. Moreover, if the compiled code contains performance-critical floating-point operations, investigate whether `-ffast-math` or any of the fine-grained options it implies can be safely used.

Some projects use `-fno-strict-aliasing` to work around type-punning problems in the source code. This is not recommended except for very low-level, hand-optimized code such as the Linux kernel. Type-based alias analysis is a powerful tool. It enables transformations like store-to-load propagation that in turn enable other high-level optimizations, including aggressive inlining and vectorization.

With the `-g` option, GCC generates useful debug information even when optimizing. However, much information is irrecoverably lost in the process. Debuggers also struggle to present a view of the state of a program in which statements are not necessarily executed in the original order. Debugging optimized code can therefore be difficult, but is usually still possible.

The compiler manual provides the complete list of optimization and command-line options. The manual is available in info format in the `gcc-info` package or online at the GCC project Web site (https://gcc.gnu.org/onlinedocs/gcc-15.2.0/gcc/).

Keep in mind that, while nearly all optimizing compilers have optimization levels sharing names for example with GCC, they do not necessarily involve the same trade-offs. For example, GCC's `-Os` optimizes for size much more aggressively than LLVM/Clang's level of the same name. Therefore, it often produces slower code. The more equivalent option in Clang is `-Oz`. Similarly, `-O2` can have different meanings for different compilers. For example, the difference between `-O2` and `-O3` is greater in GCC than in LLVM/Clang.

### Note: Changing the optimization level with **cmake**

If you use **cmake** to configure and set up builds of your application, be aware that its *release* optimization level defaults to `-O3`. This default might not be what you want. To change it, modify the `CMAKE_C_FLAGS_RELEASE`, `CMAKE_CXX_FLAGS_RELEASE` and/or `CMAKE_Fortran_FLAGS_RELEASE` variables. Because these variables are appended at the end of compilation command lines, they overwrite any level set in `CMAKE_C_FLAGS`, `CMAKE_CXX_FLAGS` and similar variables.

# 4 Taking advantage of newer processors

By default, GCC assumes that you want to run the compiled program on a wide variety of CPUs, including fairly old ones, regardless of the selected optimization level. Specifically when using GCC 15 on SUSE Linux Enterprise Server 16.0 and building applications for the `x86_64` architecture, the compiler will by default target CPUs capable of running architecture level `X86-64-v2`[4]. As one implication with a big potential performance impact, the compiler will not use `AVX` instructions, and let alone various `AVX512` instructions for floating-point and vector operations.

4 These are CPUs that support features `CMPXCHG16B (CX16)`, `LAHF-SAHF`, `POPCNT`, `SSE3`, `SSSE3`, `SSE4_1`, and `SSE4_2`.

If you know that the generated binary will only run on machines supporting newer instruction set extensions, specify it on the command line. A complete list is available in the manual, but the most prominent option `-march` lets you select a CPU model to generate code for. For example, if your program will only be executed on AMD EPYC 8635P Processors based on AMD Zen 5 cores or compatible hardware, instruct GCC to take advantage of all the instructions the CPU supports with the option `-march=znver5`.

To run the program on the machine on which you compile it, let the compiler auto-detect the target CPU model with the `-march=native` option. This works reliably only if the compiler is new enough to know the target CPU. Otherwise, that is not the case, the compiler falls back to the vendor's last known generation (that is, for example, `znver3` instead of `znver5`.

Consider using `-march=x86-64-v3` to tell the compiler that the CPUs executing the program are required to support all features defined for architecture level `X86-64-v3`. These include all AMD CPUs based on a Zen core, starting with Zen1. The most important of these features is probably `AVX2`, which allows the compiler to use 256bit vector instructions to significantly boost performance when supported.

# 5 Link Time Optimization (LTO)

*Figure 1* outlines the classic mode of operation of a compiler and a linker. Pieces of a program are compiled and optimized in chunks defined by the user called compilation units to produce so-called object files. These object files already contain binary machine instructions and are combined together by a linker. Because the linker works at a low level, it cannot perform much optimization. Thus, dividing the program into compilation units presents a profound barrier to optimization.

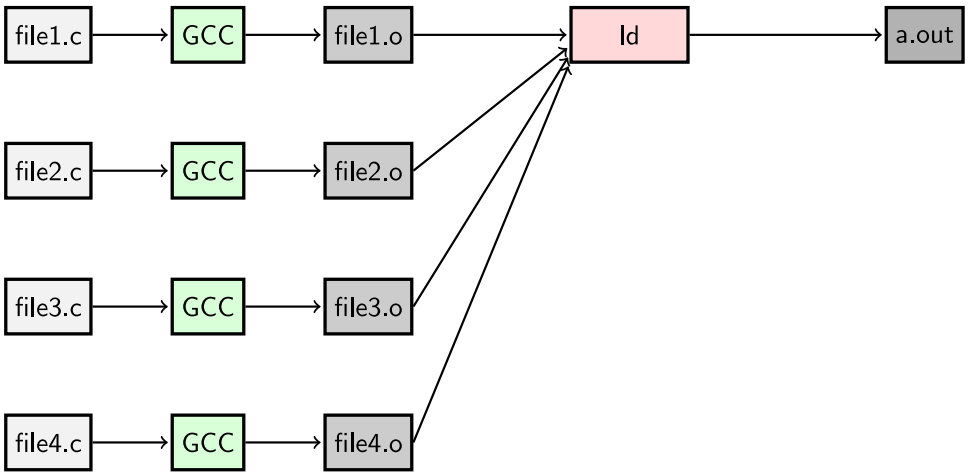


FIGURE 1: TRADITIONAL PROGRAM BUILD

To overcome this limitation, rearrange the process so that the linker does not receive nearly finished object files with machine instructions as input, but rather is invoked on files containing so-called *intermediate language* (IL). This format provides a much richer representation of each original compilation unit (see *figure 2*). The linker identifies the input as incomplete and invokes a linker plugin that runs the compiler again. This time, the compiler has access to the representation of the entire program or library being built. The compiler decides which optimizations to perform across function and compilation unit boundaries, and then divides the program into a set of partitions. Each partition is then optimized independently, machine code is emitted for it, and the result is finally linked the traditional way. Processing of the partitions is performed in parallel.

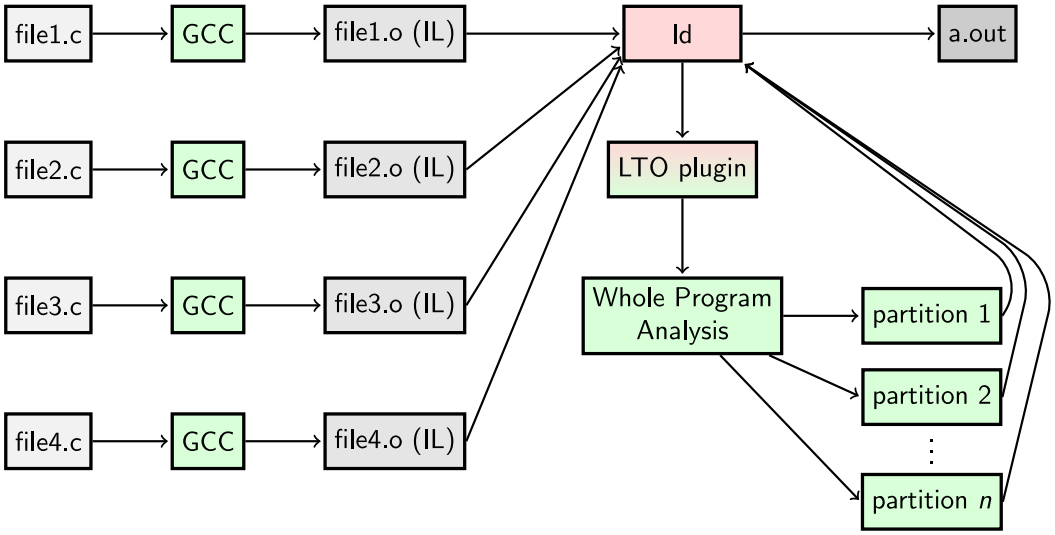


FIGURE 2: BUILDING A PROGRAM WITH GCC USING LINK TIME OPTIMIZATION (LTO)

To use **Link Time Optimization**, add the `-flto` switch to the compilation command line. Most packages in SUSE Linux Enterprise Server 16.0 are built with LTO in this manner. LTO has been the default compilation mode for packages in the openSUSE Tumbleweed distribution for years. Much effort has gone into emitting quality debug information when building with LTO. Thus, the debugging experience is no longer severely limited as it when LTO was first introduced.

LTO in GCC always consists of a *whole program analysis* (WPA) stage followed by the majority of the compilation process performed in parallel. This greatly reduces the build times of most projects. To control the parallelism, explicitly cap the number of parallel compilation processes by *n* if you specify `-flto=`*n* at linker command line. Alternatively, you can use the GNU **`make`** jobserver with `-flto=jobserv` to instruct GNU make to keep the jobserver available to the linker process[5].

You can also use `-flto=auto` which instructs GCC to search for the jobserver. If it is not found, use all available CPU threads.

Note that there is a principal architectural difference in how GCC and LLVM/Clang approach LTO. Clang provides two LTO mechanisms, so-called *thin LTO* and *full LTO*. In *full LTO*, LLVM processes the whole program as if it was a single translation unit that does not allow for any parallelism[6]. LLVM in *thin*

5 When using a **`make`** version earlier than 4.4, it was required to prepend the **makefile** rule invoking link step with character `+`. Fortunately, **`make`** in SUSE Linux Enterprise Server 16 is new enough that this is no longer necessary.

6 GCC can be configured to operate this way with the option `-flto-partition=one`.

*LTO* mode can compile different compilation units in parallel and enables inlining across compilation unit boundaries, but not most other cross-module optimizations. Therefore, this mechanism carries an inherently higher code quality penalty than *full LTO* or the GCC approach.

## 5.1 Most notable benefits of LTO

Applications built with LTO are often faster, mainly because the compiler can *inline* calls to functions in another compilation unit. This possibility also allows programmers to structure their code according to its logical division because they are not forced to put function definitions into header files to enable their inlining. As the compiler cannot inline all calls conveying information known at compilation time, GCC tracks and propagates constants, value ranges, memory reference information and devirtualization contexts from the call sites to the callees, even when passed in an aggregate or by reference. These can in turn save unnecessary computations or enable subsequent optimizations and speed up the built program or library. LTO allows such propagation across compilation unit boundaries, too.

Link Time Optimization with *whole program analysis* also offers many opportunities to shrink the code size of the built project. Functions and their parts that are not necessary in any particular project can be removed with *symbol promotion* and inter-procedural *unreachable code elimination*. This occurs even when they are not declared `static` and are not defined in an anonymous namespace. Automatic *attribute discovery* identifies C++ functions that do not throw exceptions. This allows the compiler to avoid generating a lot of code in exception cleanup regions. *Identical code folding* finds functions with the same semantics and removes all but one of them. The code size savings are often very significant and a compelling reason to use LTO even for applications which are not CPU-bound.

### Note: Building libraries with LTO

The symbol promotion is controlled by resolution information given to the linker and depends on type of the DSO build. When producing a dynamically loaded shared library, all symbols with default visibility can be overwritten by the dynamic linker. This blocks the promotion of all functions not declared inline. Thus, it is necessary to use the hidden visibility wherever possible to achieve best results. To make hidden visibility the default, use the compiler flag `-fvisibility=hidden`. Similar problems occur even when building static libraries with `-rdynamic`.

## 5.2 Potential issues with LTO

As mentioned earlier, the most packages in SUSE Linux Enterprise Server 16 and in the openSUSE Tumbleweed community distribution are built with LTO by default and work well without tweaks. Nevertheless, some low-level constructs pose a problem for LTO. One typical issue involves symbols defined in *inline assembly*, which can end up in a different partition from their uses and subsequently fail final linking. To build such projects with LTO, place assembler snippets defining symbols into a separate assembler source file so they participate only in final linking. Global `register` variables are not supported by LTO, so programs must either avoid this feature or build the traditional way. You can also exclude specific compilation units from LTO by compiling them without `-flto` or appending `-fno-lto` to the compilation command line). The rest of the program still benefits from using LTO.

Another notable limitation of LTO is that it does not support *symbol versioning* implemented with special inline assembly snippets (as opposed to a linker map file). To define symbol versions in the source files, do so with the `symver` function attribute. As an example, the following snippet will make the function `foo_v1` implement `foo` in *node* `VERS_1` (which must be specified in the version script supplied to the linker). See the manual (https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-symver-function-attribute) for more details.

```
__attribute__ ((__symver__ ("foo@VERS_1")))
int foo_v1 (void)
{
}
```

Sometimes the extra power of LTO reveals pre-existing problems that otherwise remain hidden. Violations of (strict) *aliasing* rules and C++ *one definition rule* tend to cause misbehavior significantly more often. The `-Wodr` warning reports the latter by default and should not be ignored. in addition, there are also cases where the use of the `flatten` function attribute can lead to excessive inlining with LTO. Furthermore, LTO is not suitable for code snippets compiled by `configure` scripts generated by `autoconf` to discover feature availability. This applies especially when the script searches for a string in the generated assembly.

# 6 Profile-Guided Optimization (PGO)

Optimizing compilers frequently make decisions based on which code path is most likely to execute, expected loop iterations, and similar estimates. They also often face trade-offs between potential runtime benefits and code size growth. Ideally, they optimize only frequently executed (so-called *hot*) parts of a program for speed and everything else for size. This reduces strain on caches and makes distribution of the built software cheaper. However, guessing which parts of a program are *hot* is difficult. Even sophisticated estimation algorithms implemented in GCC are no match for actual measurement.

If you do not mind adding an extra level of complexity to the build system of your project, you can make such measurement part of the process. The **makefile** (or any other) build script needs to compile the project twice. First, compile the code with the `-fprofile-generate` option and execute the resulting binary in one or multiple *train runs*. During these runs, the program saves its behavior information to special files. Afterward, the project needs to be rebuilt again, this time with the `-fprofile-use` option. This instructs the compiler to look for the files with the measurements and use them when making optimization decisions. This process is called *Profile-Guided Optimization (PGO)*.

It is important that the train run exhibits the same characteristics as the real workload. Unless you use the option `-fprofile-partial-training` in the second build, the train run must exercise the code most frequently executed in real use. Otherwise, it will be optimized for size and PGO will do more harm than good. With this option, GCC estimates properties of project portions not exercised in the train run, as if compiled without profile feedback. However, this also means that this code does not perform better or shrink as much as expected from a PGO build.

On the other hand, train runs do not need to be a perfect simulation of the real workload. For example, even though a test suite should not be a very good train run in theory because it disproportionately often tests various corner cases, in practice many projects use it as a train run and achieve significant runtime improvements with real workloads, too.

Profiles collected using an instrumented binary for multithreaded programs may be inconsistent because of missed counter updates. You can use `-fprofile-correction` in addition to `-fprofile-use` so that GCC uses heuristics to correct or smooth out such inconsistencies instead of emitting an error.

Profile-Guided Optimization can be combined with and is complimentary to Link Time Optimization. While LTO expands what the compiler can do, PGO informs it which parts of the program are important and require focus. The case study in the following section shows how the two techniques work together on a well-known set of benchmarks.

# 7 Performance evaluation: SPEC CPU 2017

*Standard Performance Evaluation Corporation* (SPEC) is a non-profit corporation that publishes a variety of industry standard benchmarks to evaluate performance and other characteristics of computer systems. Its suite of CPU intensive workloads, called SPEC CPU 2017, is often used to compare compilers and how well they optimize code with different settings. This is because the included benchmarks are well known and represent a wide variety of computation-heavy programs. The following section highlights selected results of a GCC 15 evaluation using the suite.

Note that when we use SPEC to perform compiler comparisons, we are lenient toward some official SPEC rules that system manufacturers must observe for official scores. We disregard the concepts of *base* and *peak* metrics and focus on compilation results using a particular set of options. We also patched several benchmarks:

- Benchmarks `505.mcf_r`, `511.povray_r`, and `527.cam4_r` contain an implementation of quicksort which violates (strict) C/C++ aliasing rules which can lead to erroneous behavior when optimizing at link time. SPEC decided not to change the released benchmarks and suggests that these benchmarks are built with the `-fno-strict-aliasing` option when they are built with GCC. That makes evaluation of compilers using SPEC problematic, examining their ability to use aliasing rules to facilitate optimizations is important. We have therefore disabled it only for the problematic `qsort` functions with the following function attribute:

  ```
  __attribute__((optimize("-fno-strict-aliasing")))
  ```

  As a result, the only benchmark which we compile with `-fno-strict-aliasing` is `500.perlbench_r` and `502.gcc_r`[7].

- Benchmark `511.povray_r` cannot be built with option `-ffinite-math-only` which is a part of options enabled by `-ffast-math` for reasons described in GCC bug 107021 (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=107021). The `-Ofast` measurements using GCC 15 or LLVM 21 in this section therefore append `-fno-finite-math-only` to the compilation command lines, but again only for this specific benchmark.

- We have increased the tolerance of `549.fotonik3d_r` to rounding errors after it became clear the intention was that the compiler can use relaxed semantics of floating-point operations in the benchmark (see GCC bug 84201 (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84201)).

Moreover, SPEC 2017 CPU offers so-called *speed* and *rate* metrics. For our purposes, we mostly ignore the differences and run the benchmarks configured for rate metrics (mainly because the runtimes are smaller) but we always run all benchmarks single-threaded. For these and other reasons, all the results in this document are *non-reportable*. Nevertheless, all presented results of individual benchmarks are medians from three runs.

---

7 The version of the GCC compiler from which `502.gcc_r` derived contains bug 48981 (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=48981) which was a strict aliasing rules violation. The upstream GCC project has fixed this issue so that it can be built with strict-aliasing enabled.

Finally, SPEC specifies a base runtime for each benchmark and defines a *rate* as the ratio of the base runtime and the median measured runtime (this rate is a separate concept from the rate metrics). The overall suite score is then calculated as geometric mean of these ratios. The bigger the rate or score, the better it is. In the remainder of this section, we report runtimes using relative rates and their geometric means as they were measured on an AMD EPYC 8635P Processor running SUSE Linux Enterprise Server 16.0.

## 7.1 Benefits of LTO and PGO

In *Section 3, "Optimization levels and related options"*, we recommend compiling HPC workloads with `-O3` and benchmarks with `-Ofast`. However, examining integer-crunching benchmarks built with only `-O2` is still useful, as Linux distributions often build the underlying programs this way. As mentioned earlier, both SUSE and openSUSE build almost all recent distributions with LTO and selected packages with PGO. The following paragraphs demonstrate why.

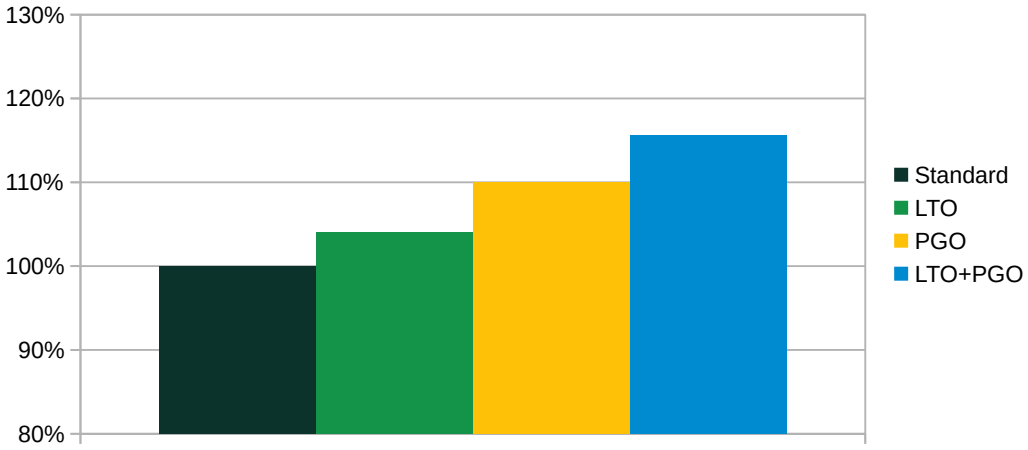


FIGURE 3: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC INTRATE 2017 BUILT WITH GCC 15.2 AND -O2

*Figure 3* shows the overall performance effect on the whole integer benchmark suite as captured by the geometric mean of all individual benchmark rates. Using both PGO and LTO results in remarkable relative uplift of 15.6%. This occurs even though starting with GCC 12, the compiler can conservatively auto-vectorize code in `525.x264_r` at `-O2`, which previously required PGO at this level. Nevertheless, this benchmark still benefits a lot when the profile information is available, together with several others derived from programs typically compiled with `-O2`. This is illustrated in *figure 4*.

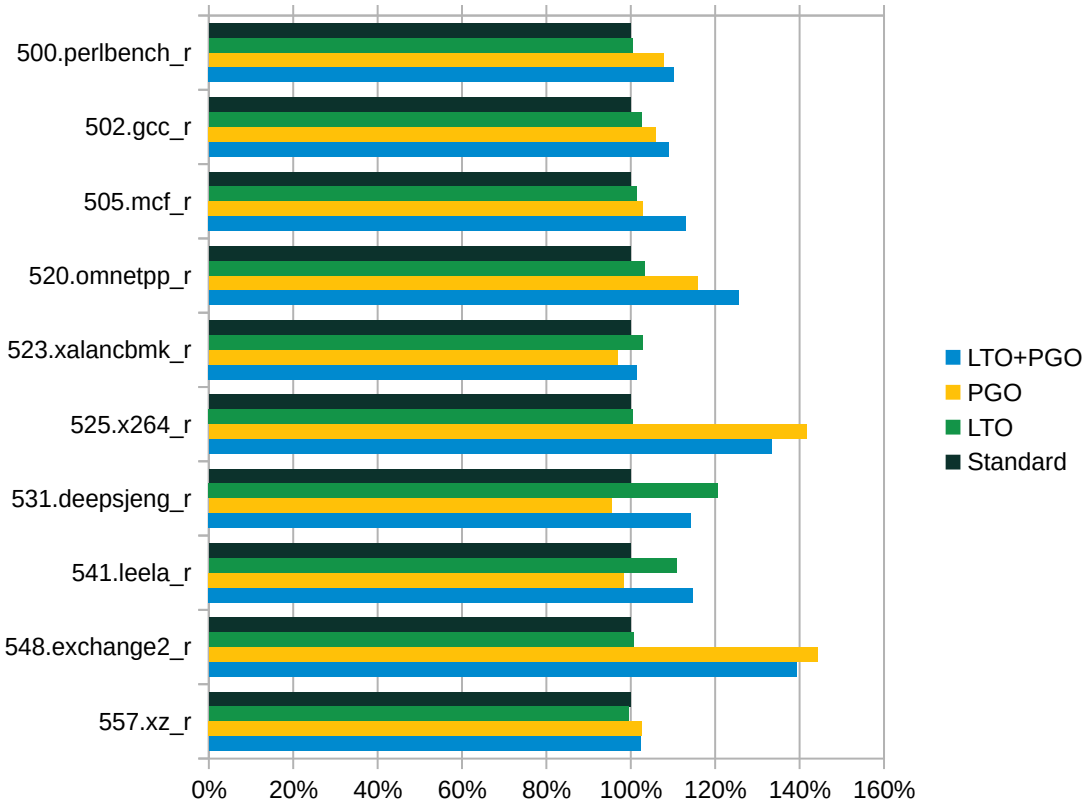


FIGURE 4: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF INDIVIDUAL INTEGER BENCHMARKS BUILT WITH GCC 15.2 AND -O2

*Figure 5* shows another important advantage of LTO and PGO: a significant reduction of the size of the binaries (measured without debug info). Note that the figure does not show that the size of benchmark `548.exchange2_r` grew to 266% of the original size with PGO and to 177% with both PGO and LTO. Although this growth appears large, it stems from a small base. It is the only Fortran benchmark in the integer suite and, its size penalty is offset by significant speed-up, making the trade-off reasonable. For completeness, we show this result in *figure 6*. When we sum sizes of all ten benchmarks, LTO shrinks size by 5%, PGO by 7.5% and using both by 14.5%.

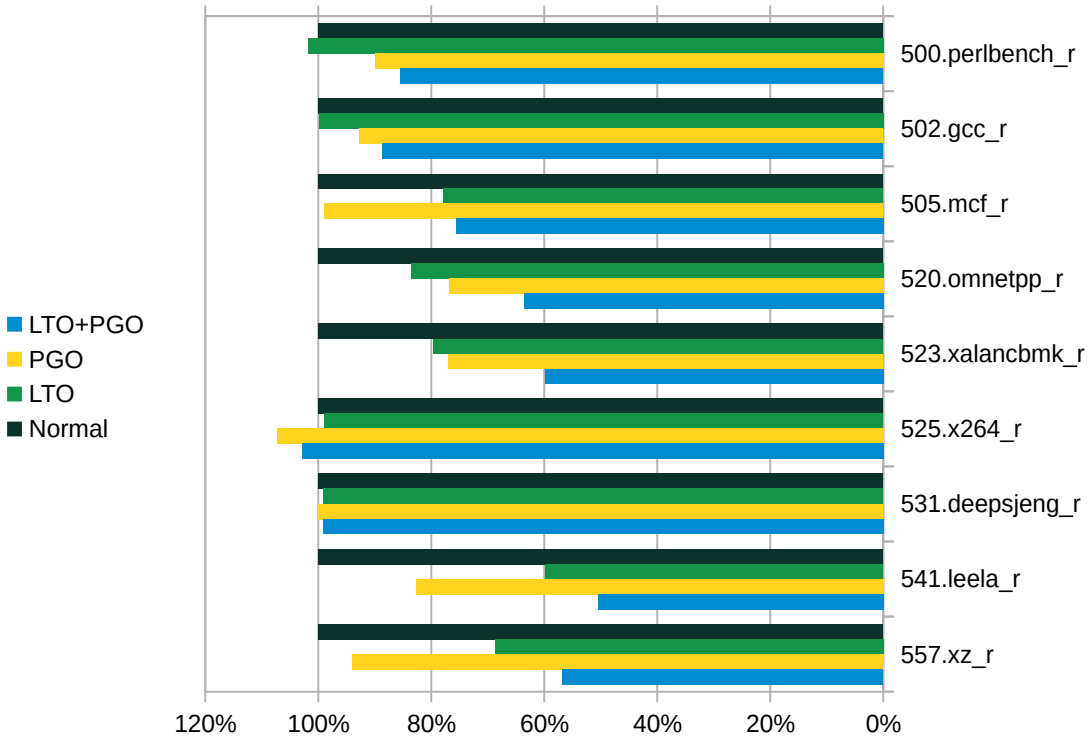


FIGURE 5: BINARY SIZE (SMALLER IS BETTER) OF INDIVIDUAL INTEGER BENCHMARKS BUILT WITH GCC 15.2 AND -O2

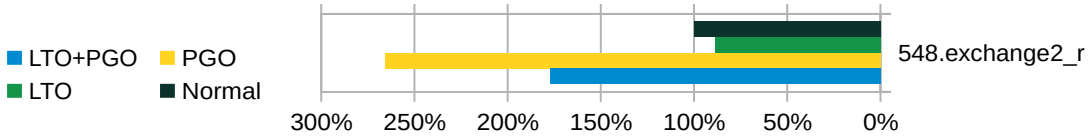


FIGURE 6: BINARY SIZE (SMALLER IS BETTER) OF 548.EXCHANGE2_R BUILT WITH GCC 15.2 AND -O2

Runtime benefits and binary size savings are also visible when using the optimization level `-Ofast` and option `-march=native`. This enables the compiler to take advantage of all instructions that the AMD EPYC 8635P Processor supports. *Figure 7* shows the respective geometric means, and *figure 8* shows how rates change for individual benchmarks. Even at the aggressive optimization level, PGO brings clear benefits for benchmarks derived from interpreters and compilers like `500.perlbench_r` and `502.gcc_r`. However, the compiler can struggle to update measured profile information during complex inter-procedural optimizations, such as in `548.exchange2_r`. This can cause the technique to decrease performance. Finally, although optimization levels `-O3` and `-Ofast` permit to be relaxed about the final binary size, PGO and especially LTO can reduce binary size at these levels, too. *Figure 9* shows the relative binary sizes of all integer benchmarks.

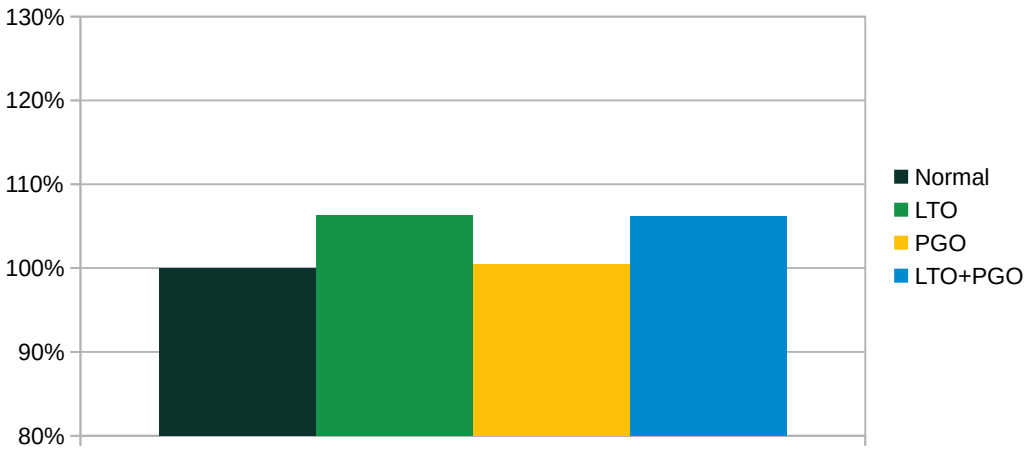


FIGURE 7: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC INTRATE 2017 BUILT WITH GCC 15.2 USING -OFAST AND -MARCH=NATIVE

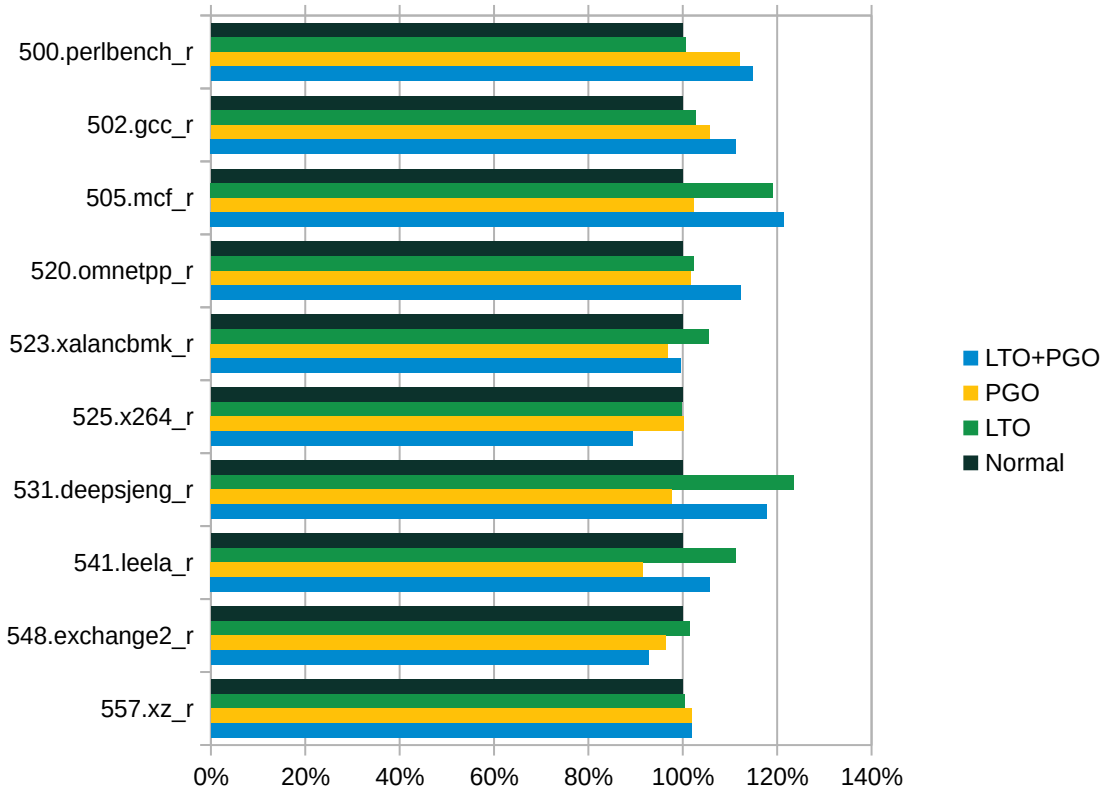


FIGURE 8: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF INDIVIDUAL INTEGER BENCHMARKS BUILT WITH GCC 15.2 USING -OFAST AND -MARCH=NATIVE

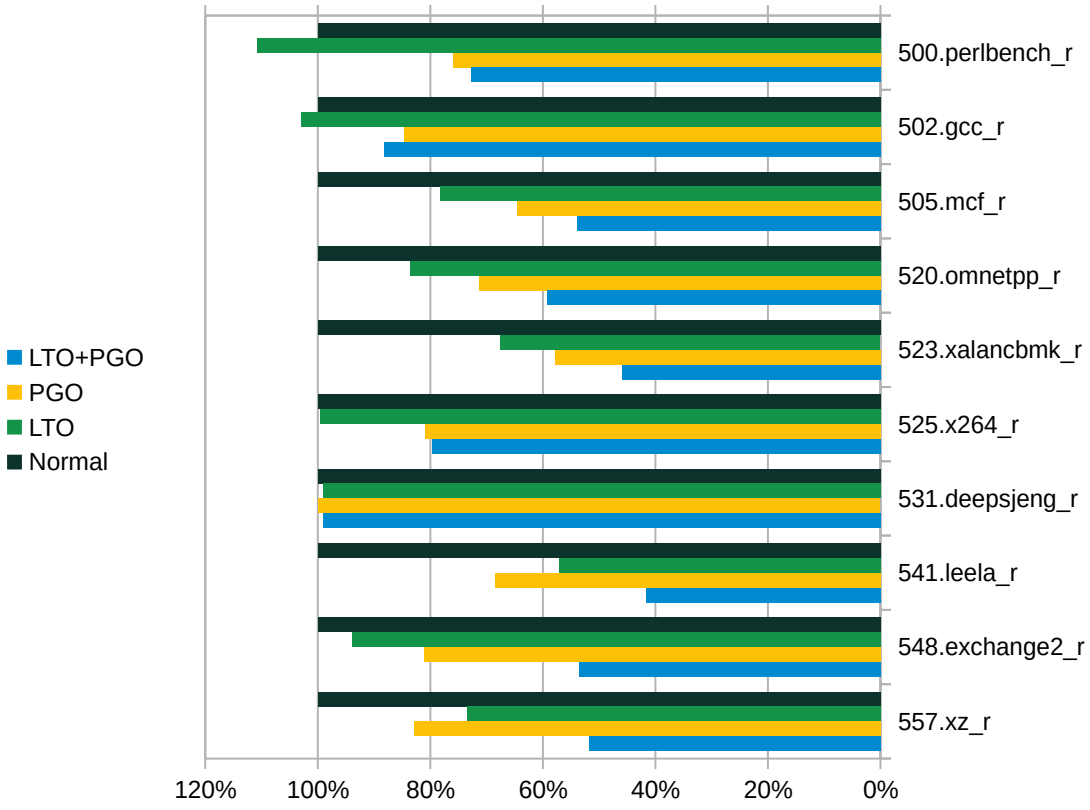


FIGURE 9: BINARY SIZE (SMALLER IS BETTER) OF SPEC INTRATE 2017 BUILT WITH GCC 15.2 USING -OFAST AND -MARCH=NATIVE

Many of the SPEC 2017 floating-point benchmarks measure how well a system can optimize and execute a handful of number crunching loops. They often come from performance sensitive programs written with traditional compilation method in mind. As a result, there are fewer cross-module dependencies, making the identification of hot paths less critical. Consequently, the overall impact of LTO and PGO on the suite is often minimal. Nevertheless, there are important cases when these modes of compilation also bring significant performance increases. *Figure 10* shows the effect of these methods on individual benchmarks when compiled at `-Ofast` and targeting the full ISA of the AMD EPYC 8635P Processor. Furthermore, binary size savings of PGO and LTO are sometimes even bigger than those achieved on integer benchmarks, as can be seen in *figure 11*

Two cases require explanation here. First, `519.lbm_r` compiled with GCC 15.2 and LTO suffers a code positioning problem. When the main hottest loop of the benchmark is placed at an unfortunate address, the CPU executes it considerably slower. This is rare and difficult to reproduce. Changing the start address of the encapsulating function is sufficient to mitigate the issue.

Second, in the `538.imagick_r` benchmark, a mismatch exists between the code paths exercised in the train run and the reference run. The train run determines which program parts to optimize for speed, while the reference run provides the benchmark score. This is the problem we warn against in *Section 6, "Pro-*

*file-Guided Optimization (PGO)"*, and it shows the predictable detrimental effect on performance.[8] Moreover, because the main loop, which is not appropriately optimized as not executed in the train run, is in a function in which another loop is heavily executed in the train run, even using the `-fprofile-partial-training` does not help to mitigate the problem. This is a bug in the SPEC CPU suite that causes the overall performance score to decrease by 0.5% when using both LTO and PGO.

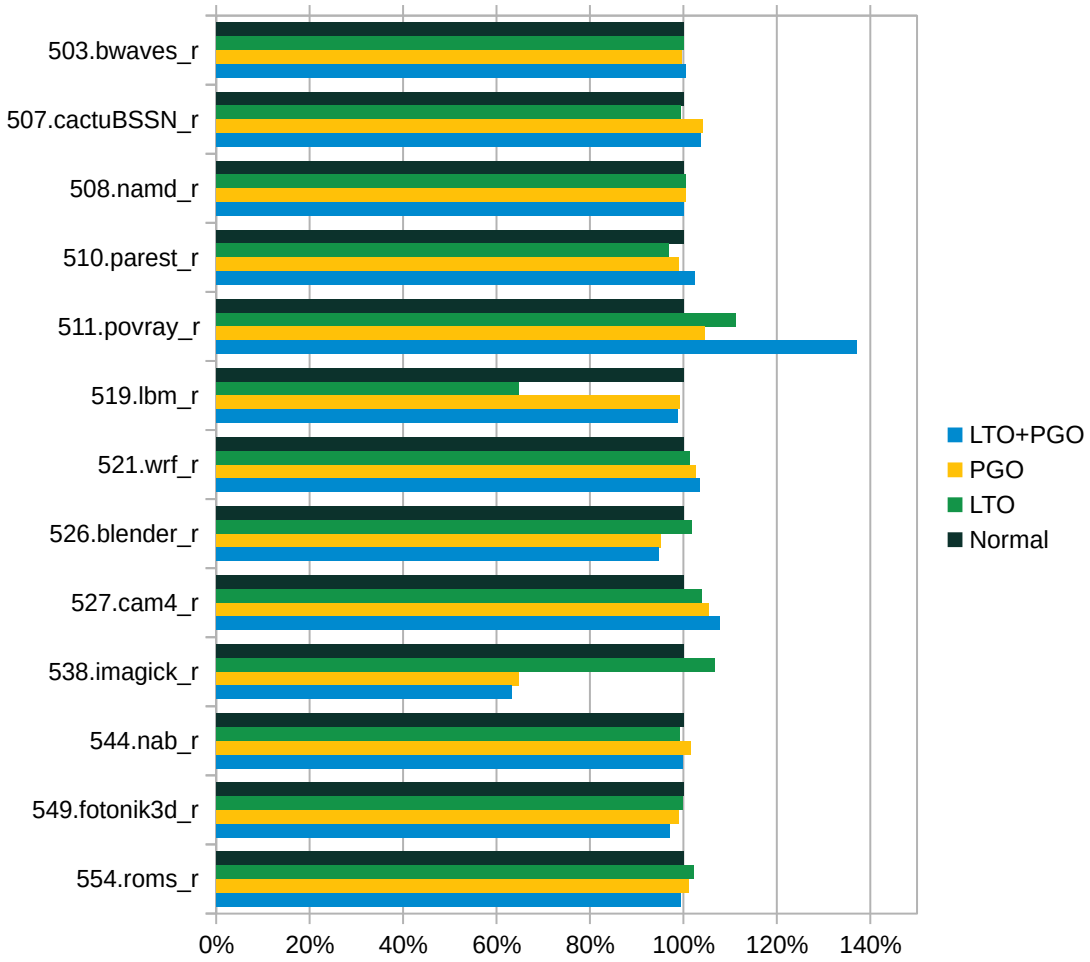


FIGURE 10: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF INDIVIDUAL FLOATING-POINT BENCHMARKS BUILT WITH GCC 15.2 USING -OFAST AND -MARCH=NATIVE

**8** See GCC bug 111551 (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=111551) for more details.

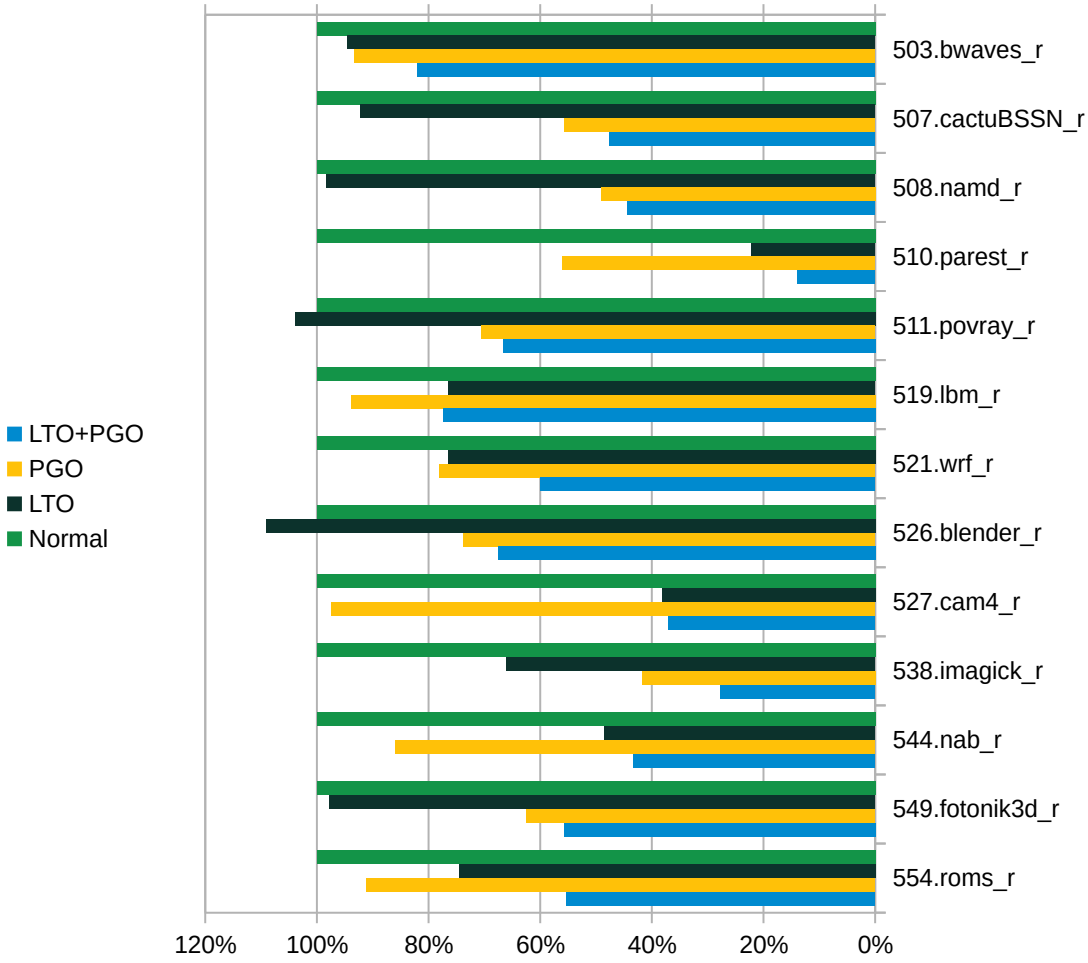


FIGURE 11: BINARY SIZE (SMALLER IS BETTER) OF SPEC FPRATE 2017 BUILT WITH GCC 15.2 USING -OFAST AND -MARCH=NATIVE

## 7.2 GCC 15.2 compared to GCC 13

Upgrading the compiler sometimes can lead to issues, as described in section *Section 2.3, "Potential issues with the recent versions of the GCC compiler"*. Therefore, users sometimes hesitate to upgrade when they do not need new features. Reasons to upgrade usually include better performance of code generated by new compilers. This chapter uses the SPEC CPU 2017 suite to examine how GCC 15 compares to GCC13 in this regard. GCC 13 is only two years older and was the first major version targeting Zen5-based processors, such as the AMD EPYC 8635P Processor we used.

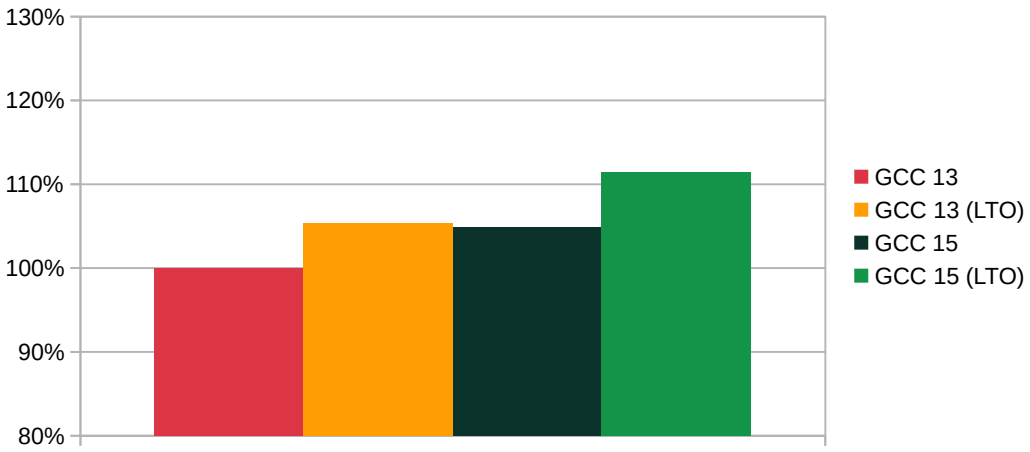


FIGURE 12: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC INTRATE 2017 BUILT WITH GCC 13.4 AND 15.2 (-OFAST -MARCH=NATIVE)

*Figure 12* captures the benefits of using the newer compiler with integer workloads in the form of relative improvements of the geometric mean across the SPEC INTrate 2017 suite. If we compare the geometric means obtained with LTO, GCC 15 achieves 5.9% better score. *Figure 13* dives deeper and shows which particular benchmarks gained most in terms of performance. The benchmark `525.x264_r` illustrates how newer compilers sometimes manage to use modern hardware better. Both GCC 13 and GCC 15 can correctly identify a Zen5-based CPU and can auto-vectorize code using 512 bit vectors. In this particular scenario, doing this alone, like GCC 13 does, reduces performance because the main loops do not iterate enough times. GCC 15 produces cascaded vectorized epilogues in this scenario, ensuring the compiler uses the optimal vector width. Because big parts of compiler optimizations rely on heuristics, small regressions can still occur. In this scenario, without LTO, `531.deepsjeng_r` regressed by 3%. However, compilation with LTO compilation showed no regression.

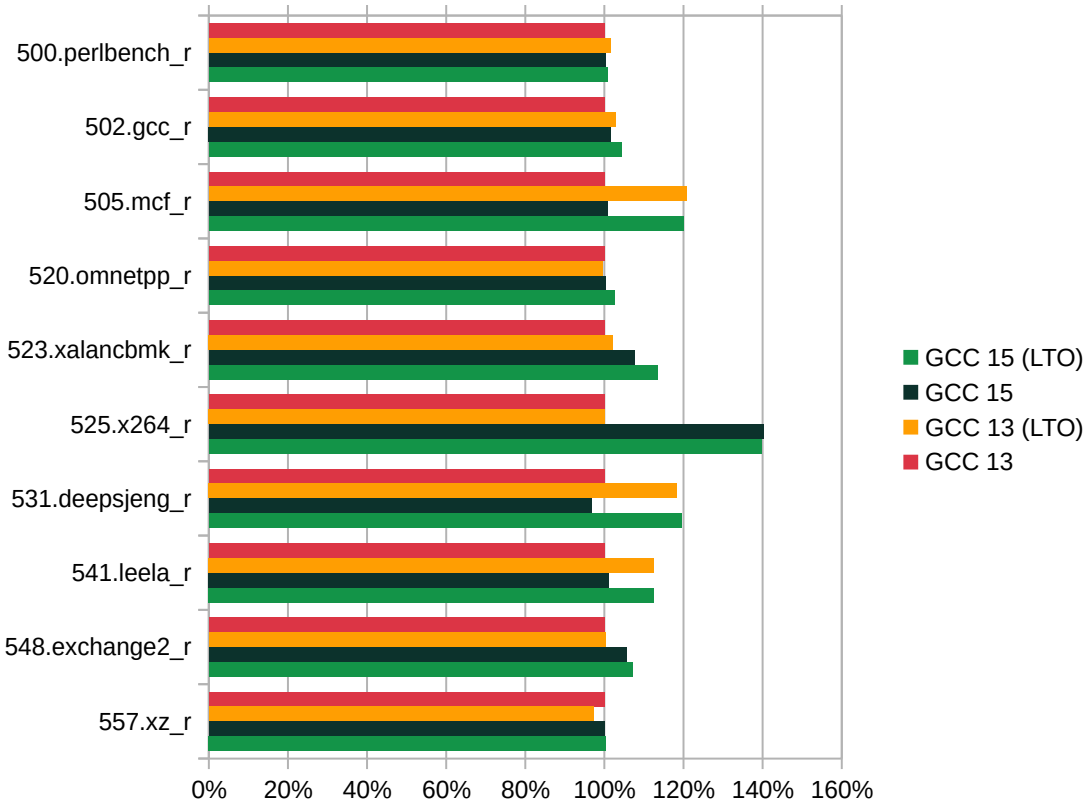


FIGURE 13: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF SELECTED INTEGER BENCHMARKS BUILT WITH GCC 13.4 AND 15.2 (-OFAST -MARCH=NATIVE)

On the other hand, floating-point results show that performance rarely degrades severely. As previously discussed, `519.lbm_r` compiled with GCC 15.2 and LTO has a code positioning issue that slows execution significantly. Outside a benchmark context, this particular problem is easy to identify and fix by modifying the source code. For a standard benchmark, however, we can only report a 45% performance reduction of that particular benchmark. In contrast, *figure 14* shows that the non-LTO variant of the benchmark improved slightly. This suggests the LTO variant would also improve if it avoided this issue. The same figure shows clear improvements in other benchmarks, particularly`538.imagick_r`. The compiler optimizes to avoid *store-to-load forwarding stalls*, yielding a 39% improvement in this benchmark.

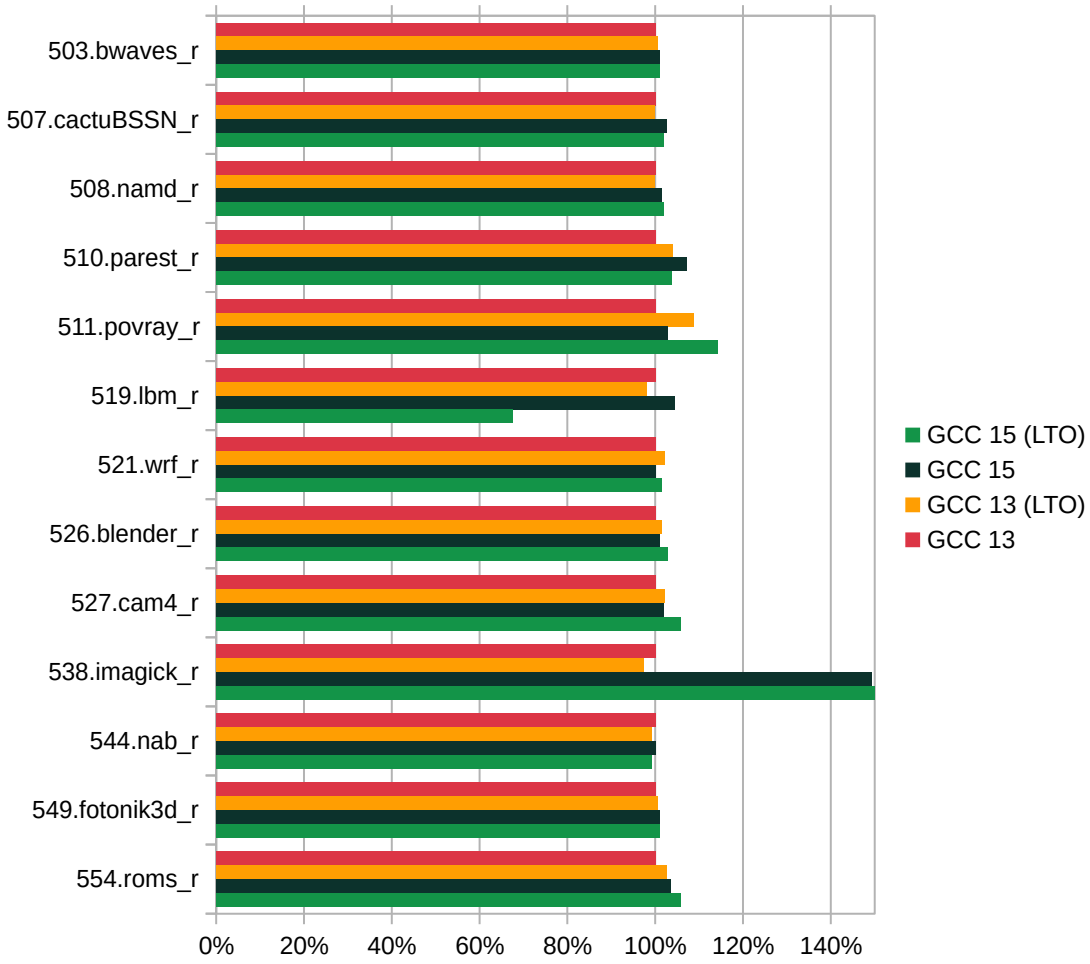


FIGURE 14: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF SELECTED FLOATING-POINT BENCHMARKS BUILT WITH GCC 13.4 AND 15.2 (-OFAST -MARCH=NATIVE)

The overall score for the floating point benchmarks is available in *figure 15*. However, we also provide *figure 16* which compares the geometric mean of scores of all benchmarks except `519.lbm_r`. This provides a better comparison of the compilers because there is little that a compiler can do to avoid the code placement problem. in general, GCC 13 is as susceptible to this issue as the newer version.

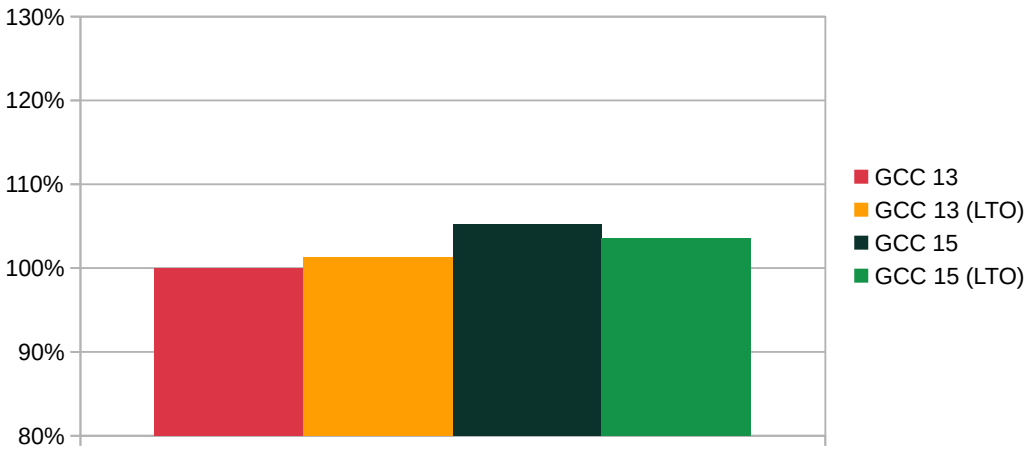


FIGURE 15: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC FPRATE 2017 BUILT WITH GCC 13.4 AND 15.2 (-OFAST -MARCH=NATIVE)

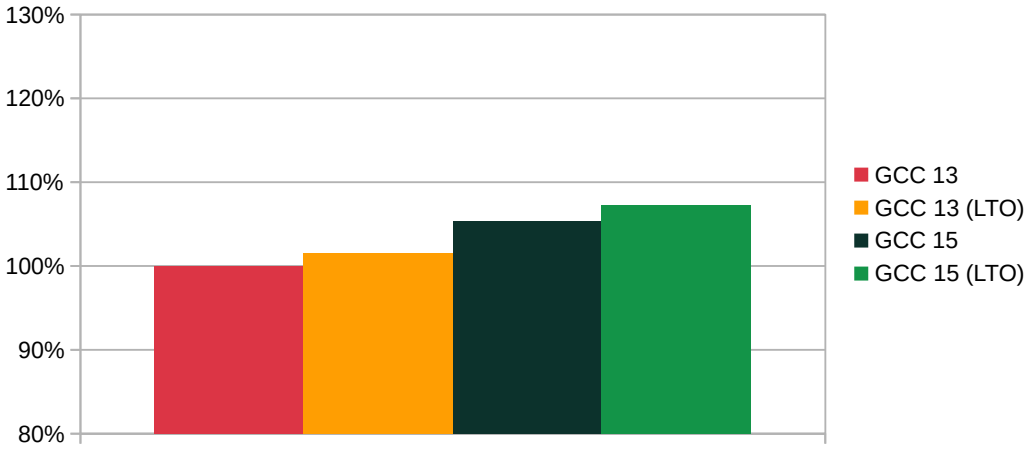


FIGURE 16: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC FPRATE 2017 EXCLUDING `519.lbm_r` BUILT WITH GCC 13.4 AND 15.2 (-OFAST -MARCH=NATIVE)

## 7.3 Effects of `-ffast-math` on floating-point performance

In *Section 3, "Optimization levels and related options"*, we noted failing to relax floating-point math functions semantics can sacrifice performance. This applies even when strict adherence to all IEEE and/or ISO rules is unnecessary. This section uses the SPEC FPrate 2017 test suite to demonstrate this performance impact.

We have built the benchmarking suite using optimization level `-O3`, LTO (though without PGO) and `-march=native` to target the native ISA of the AMD EPYC 8635P Processor. Then we compared its runtime score against the suite built with these options and `-ffast-math`. Overall, the build using `-ffast-`

`math` performed 7% better across all benchmarks. When excluding `519.lbm_r` from the comparison due to the code placement issue described earlier, performance improved by 11%. Nevertheless, if you look at *figure 17* it shows that the scores of five benchmarks improved by more than 10%, with `538.imagick_r` growing by 57%.

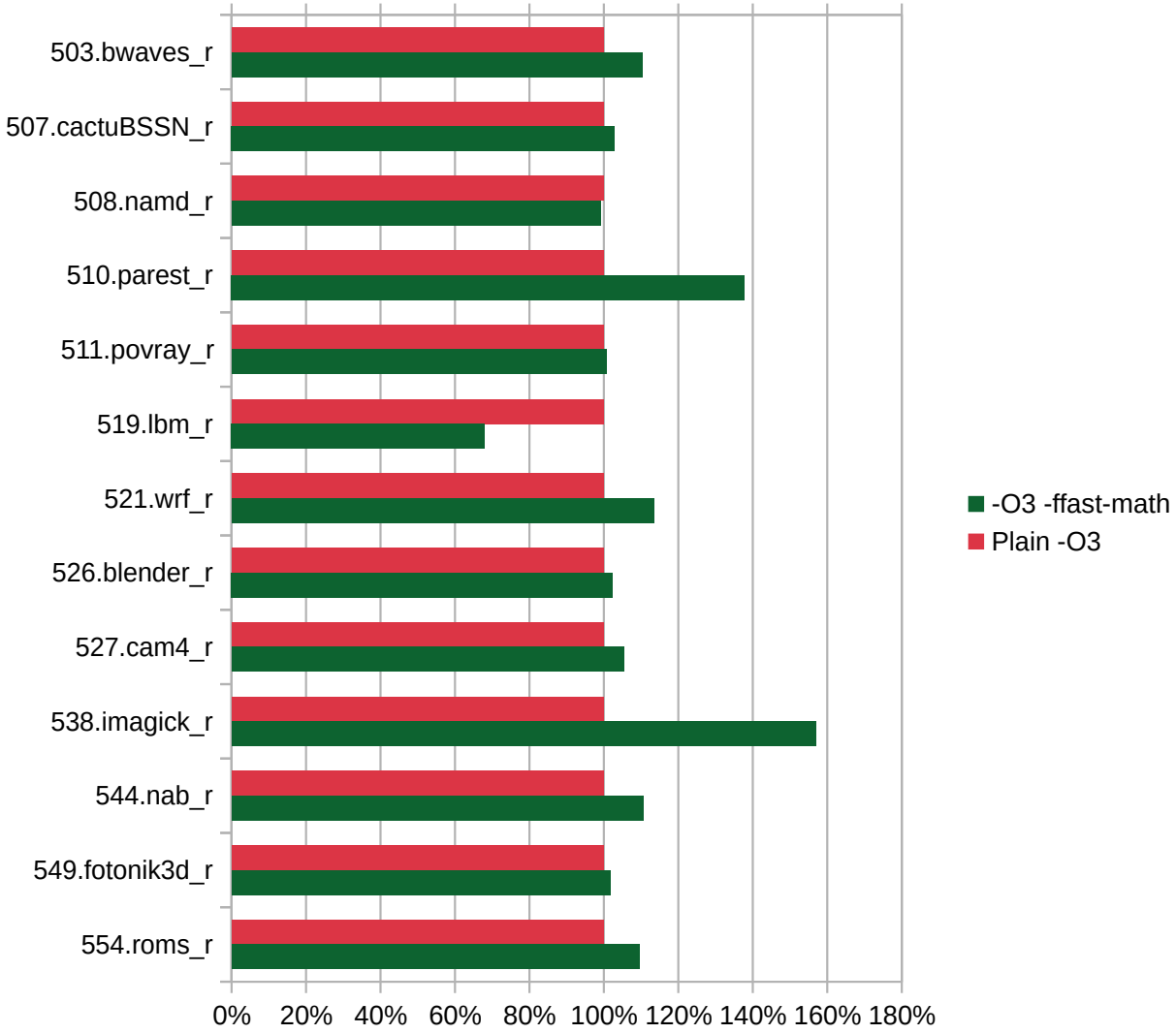


FIGURE 17: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF INDIVIDUAL FLOATING-POINT BENCHMARKS BUILT WITH GCC 15.2 AND -O3 -FLTO -MARCH=NATIVE, WITHOUT AND WITH -FFAST-MATH

## 7.4 Comparison with other compilers

The SUSE toolchain team regularly uses the SPEC CPU 2017 suite to compare the optimization capabilities of GCC with other compilers, primarily LLVM/Clang and ICX from Intel. In the final section of this case study, we discuss how the default compiler on SUSE Linux Enterprise Server 16.0 compares to these

competitors. Before starting, note that individuals with significantly more expertise in GCC than in the other compilers conducted the comparison and are not completely “unbiased”. In addition, keep in mind that all previous explanations about how we carry out measurements and patch benchmarks also apply to this section. However, because the results inform our own work, rest assured that we strive for accuracy.

We built the `clang`, `clang++` and `flang-new` compilers from sources obtained from the official git repository (tag `llvmorg-22.1.7`). We used them to compile the SPEC CPU 2017 suite with `-Ofast` and `-march=native`. We then compared the performance against the suites built with GCC 15.2 using the same options. When using LLVM/Clang's LTO to compile SPEC, we selected the *full* variant. This variant provides stronger optimization capabilities, although it is not suitable for building large projects.

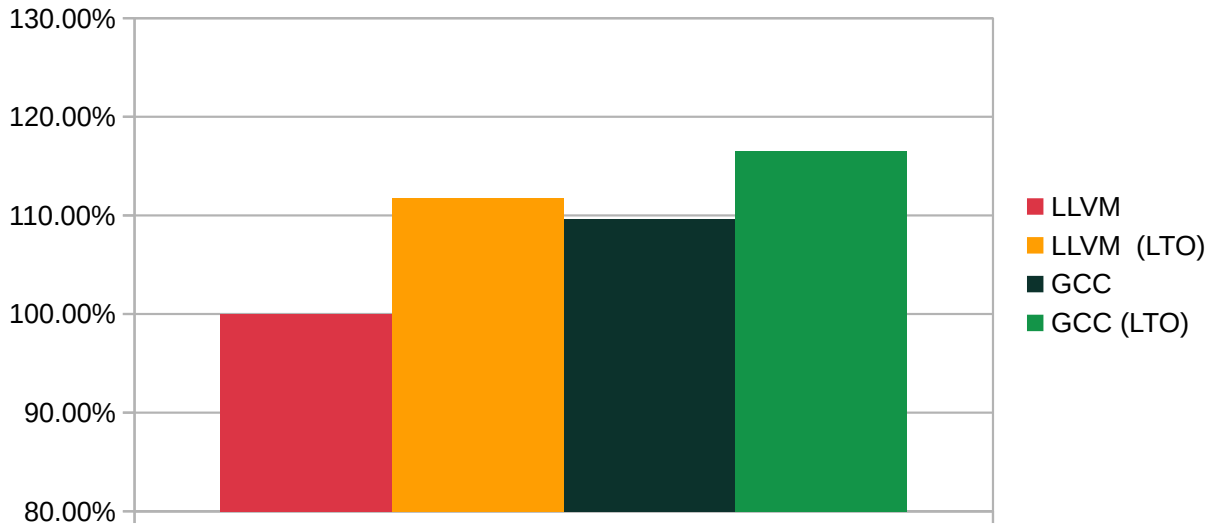


FIGURE 18: OVERALL PERFORMANCE (BIGGER IS BETTER) OF C/C++ INTEGER BENCHMARKS BUILT WITH LLVM/CLANG 22 AND GCC 15.2

*Figure 18* shows that the geometric mean of the whole SPEC INTrate 2017 suite runs faster when benchmarks are compiled with GCC. *Figure 19* which shows results for individual benchmarks confirms this. LLVM/Clang produces faster code for two benchmarks: `525.x264_r` by 4% and `541.leela_r` by 6% with LTO. However, code generated by GCC performs better in more cases and by larger margins. For example, `505.mcf_r` is 11% faster and `523.xalancbmk_r` is 17% faster.

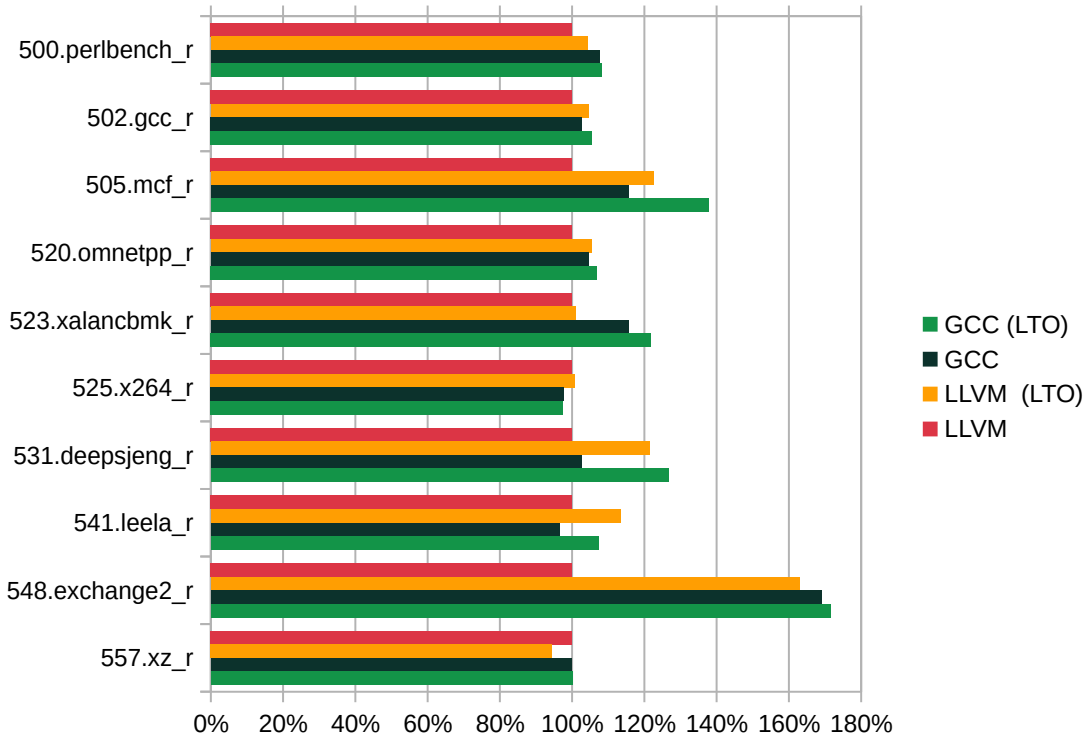


FIGURE 19: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF C/C++ INTEGER BENCHMARKS BUILT WITH LLVM/CLANG 22 AND GCC 15.2

The comparison of the geometric mean scores for the SPEC FPrate 2017 suite built with both compiler suites is shown in *figure 20*. Individual results are compared in *figure 21*. Again, LLVM/Clang outperforms GCC in two benchmarks: `508.namd_r` by 11% and `544.nab_r` by 20% when comparing LTO scores. However, GCC generates notably faster code in nine cases, sometimes also by a wide margins. The two largest differences are `519.lbm_r` at 25% and `538.imagick_r` at 45%. Despite the code placement issue we described earlier, `519.lbm_r` compiled with GCC 15 and LTO is still a lot faster than when compiled with LLVM/Clang.

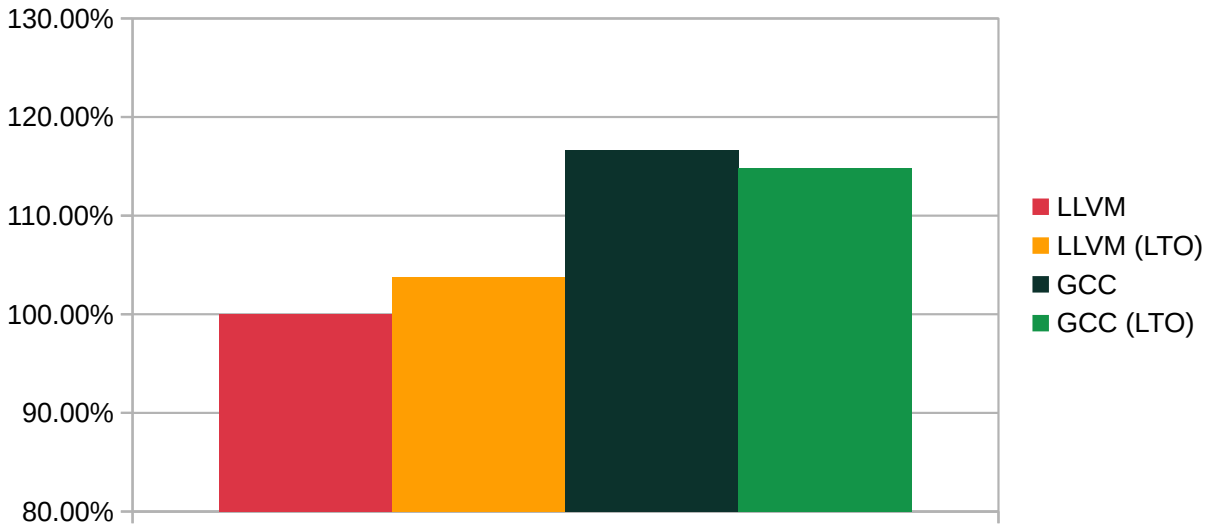


FIGURE 20: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC FPRATE 2017 BUILT WITH LLVM/CLANG 22 AND GCC 15.2

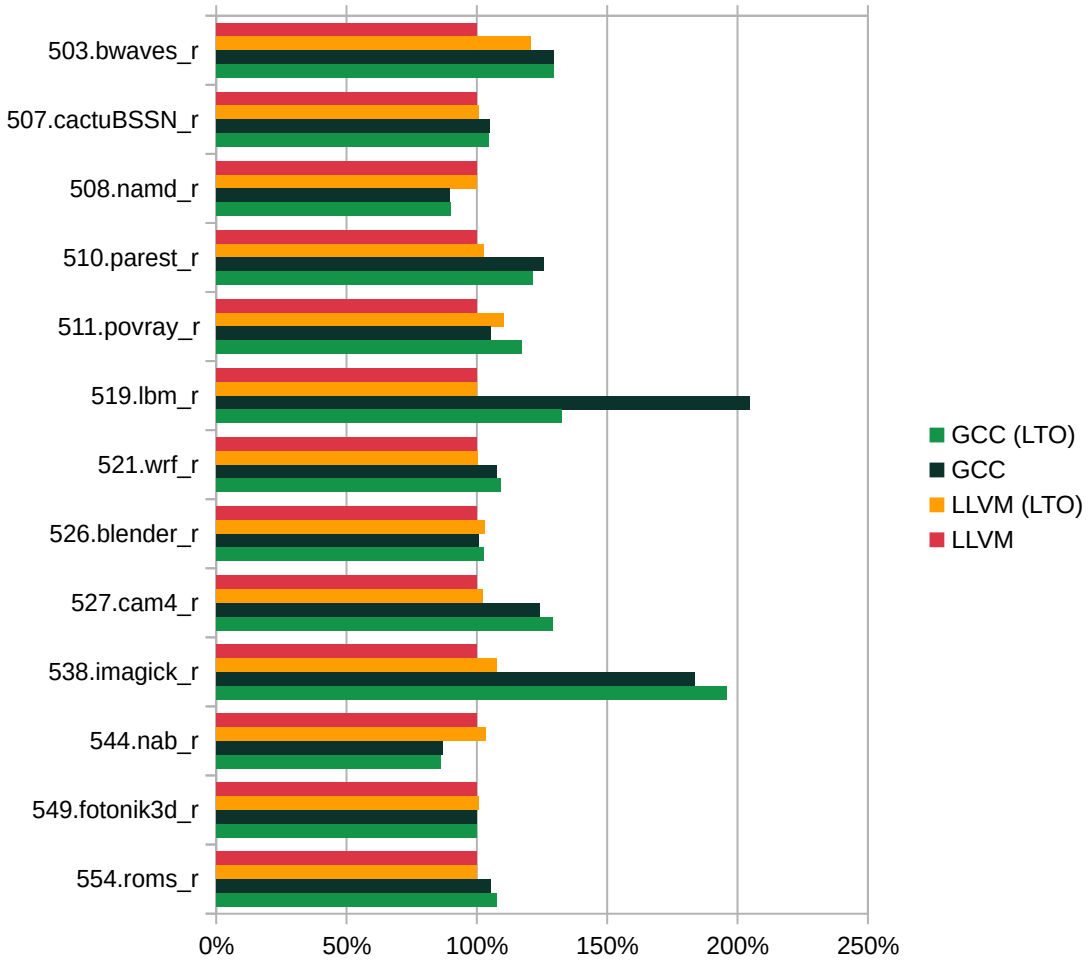


FIGURE 21: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF FLOATING POINT BENCHMARKS BUILT WITH LLVM/CLANG 22 AND GCC 15.2

Although Intel compilers are not designed for AMD processors, they are well-known for their high-level optimization capabilities, particularly in vectorization. Therefore, we traditionally included ICC in our comparisons of compilers. When Intel discontinued this compiler and transitioned to ICX, a new compiler built on top of LLVM, we began comparing GCC with ICX instead.

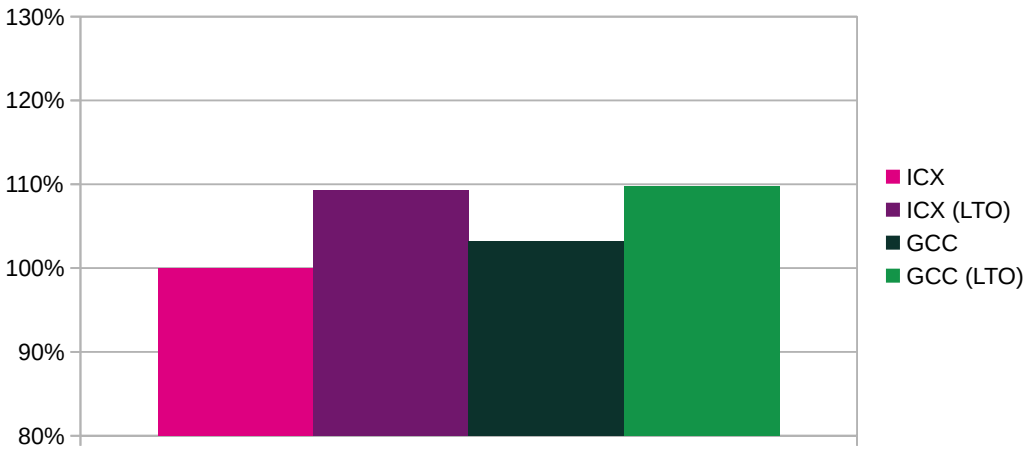


FIGURE 22: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC INTRATE 2017 BUILT WITH ICX 2026.0.0 AND GCC 15.2

*Figure 22* shows the overall SPEC INTrate scores achieved by both compilers using `-Ofast` and `-march=native`. The results of individual benchmarks are in *figure 23*. Both figures show comparable performance between the compilers. The two notable differences are that, with LTO, GCC produces `525.x264_r` that is 15% slower and `548.exchange2_r` that is 15% faster.

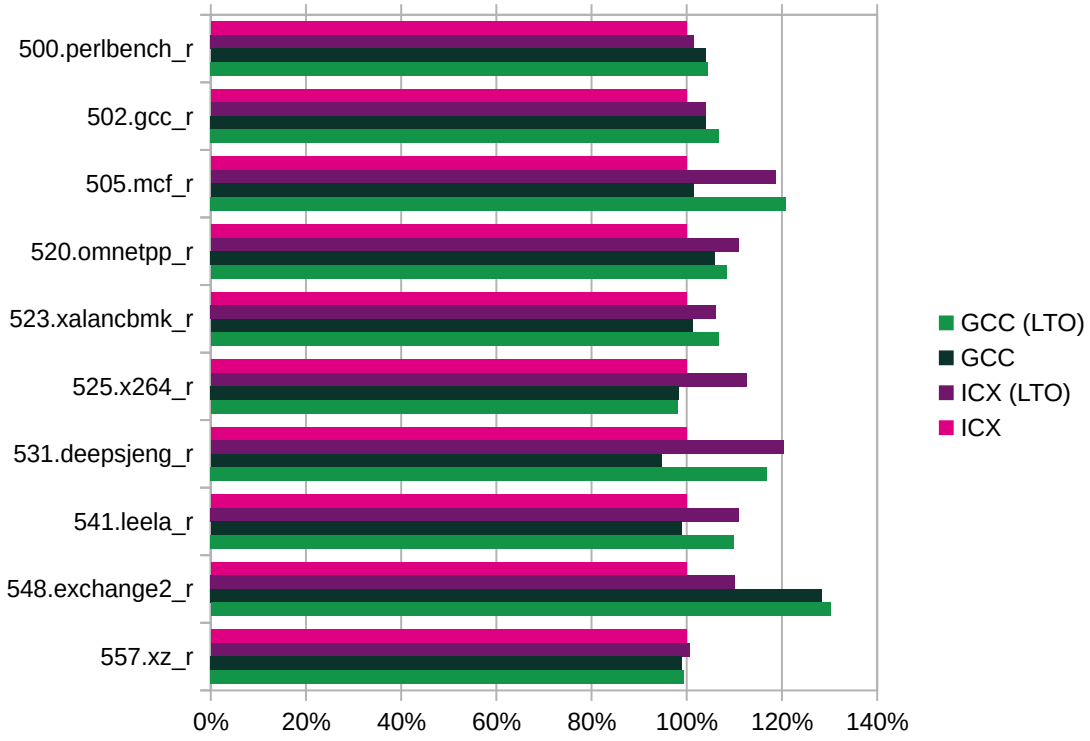


FIGURE 23: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF INDIVIDUAL INTEGER BENCHMARKS BUILT WITH ICX 2026.0.0 AND GCC 15.2

When building the SPEC FPrate suite, GCC achieves a 12% higher geometric mean without LTO and 5% with LTO (see *figure 24*). However, individual benchmark results reveal a more nuanced overall picture (see *figure 25*). There are benchmarks where GCC generates much faster code (most prominently for example in `538.imagick_r` and `554.roms_r`). However, there are also benchmarks where ICX generates considerably faster code (especially in `519.lbm_r` and `544.nab_r`, and non-LTO results suggest ICX would remain faster even without the code placement issue in GCC described earlier).

Nevertheless, in conclusion GCC performs consistently and competitively against these high-performance compilers.

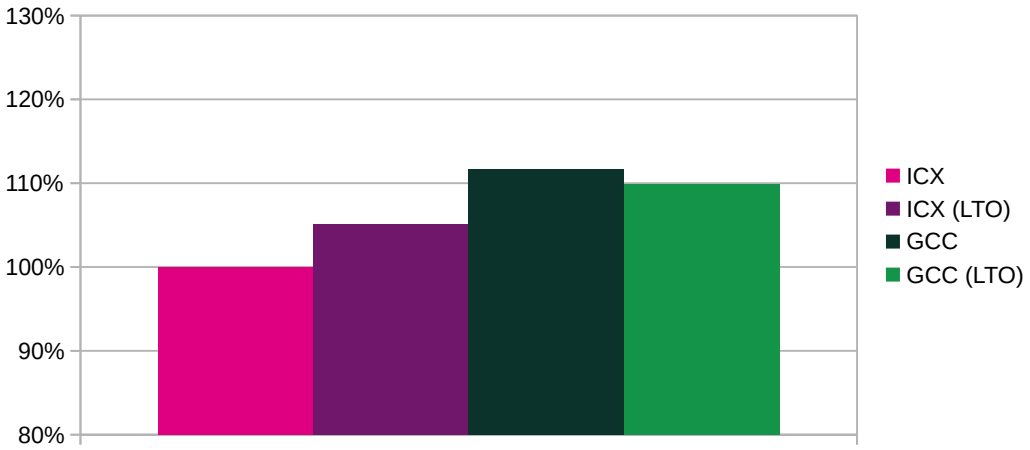


FIGURE 24: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC FPRATE 2017 BUILT WITH ICX 2026.0.0 AND GCC 15.2

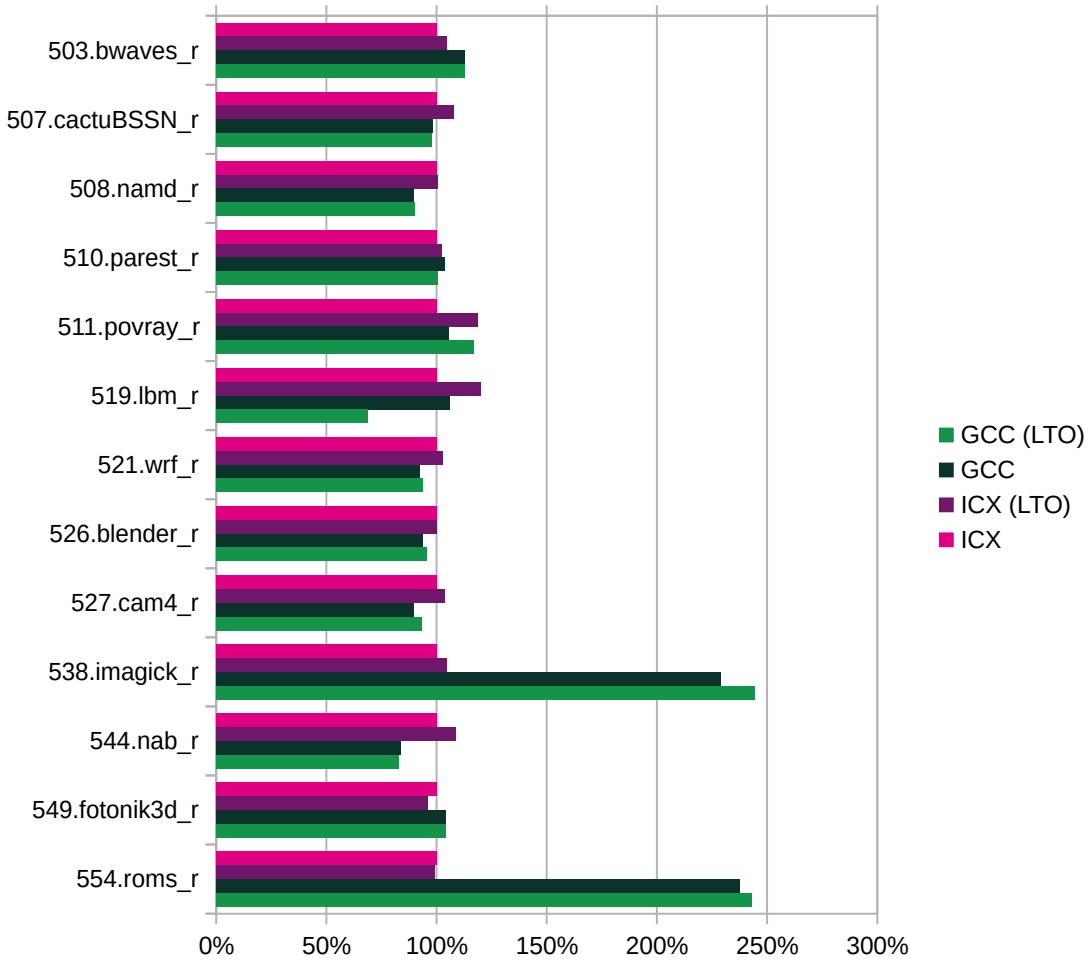


FIGURE 25: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF INDIVIDUAL FLOATING POINT BENCHMARKS BUILT WITH ICX 2026.0.1 AND GCC 15.2

# 8 Legal notice

Copyright ©2006-2026 SUSE LLC and contributors. All rights reserved.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or (at your option) version 1.3; with the Invariant Section being this copyright notice and license. A copy of the license version 1.2 is included in the section entitled “GNU Free Documentation License”.

SUSE, the SUSE logo and YaST are registered trademarks of SUSE LLC in the United States and other countries. For SUSE trademarks, see http://www.suse.com/company/legal/. Linux is a registered trademark of Linus Torvalds. All other names or trademarks mentioned in this document may be trademarks or registered trademarks of their respective owners.

Documents published as part of the **SUSE Best Practices** series have been contributed voluntarily by SUSE employees and third parties. They are meant to serve as examples of how particular actions can be performed. They have been compiled with utmost attention to detail. However, this does not guarantee complete accuracy. SUSE cannot verify that actions described in these documents do what is claimed or whether actions described have unintended consequences. SUSE LLC, its affiliates, the authors, and the translators may not be held liable for possible errors or the consequences thereof.

Below we draw your attention to the license under which the articles are published.

## GNU Free Documentation License

Copyright (C) 2000, 2001, 2002 Free Software Foundation, Inc. 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA. Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

### 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

### 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

### 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

### 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

## 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

## ADDENDUM: How to use this License for your documents

```
Copyright (c) YEAR YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
 Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts". line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.