Advanced Optimization and New Capabilities of GCC 14

SUSE Linux Enterprise Server 15 SP6 and later
Development Tools Module

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)

The document at hand provides an overview of GCC 14.2 as the current Development Tools Module compiler in SUSE Linux Enterprise 15 SP6. 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 9005 Series Processors.

Contents

1 Overview

The first release of the GNU Compiler Collection (GCC) with the major version 14, GCC 14.1, took place in May 2024. GCC 14.2, with fixes to over 100 bugs, was released in August of the same year. Soon after, the openSUSE Tumbleweed Linux distribution began using this compiler to build its packages. Subsequently, it has replaced the compiler in the SUSE Linux Enterprise (SLE) Development Tools Module. GCC 14 is the first major version to support the new capabilities of a wide range of computer architectures, including AMD CPUs based on the Zen 5 core. It also introduces many new features. These include the 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). Additionally, there are numerous generic improvements in optimization.

This document gives an overview of GCC 14. 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 9005 Series Processors.

2 System compiler versus Development Tools Module compiler

The major version of the system compiler in SUSE Linux Enterprise 15 remains to be GCC 7, regardless of the service pack level. This is to minimize the danger of any unintended changes over the entire life time of the product.

```
sles15: # gcc --version
gcc (SUSE Linux) 7.5.0
Copyright (C) 2017 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.
```

That does not mean that, as a user of SUSE Linux Enterprise 15, you are forced to use a compiler with features frozen in 2016. You can install an add-on module called **Development Tools Module** which is included in the SUSE Linux Enterprise Server 15 subscription and contains a much newer compiler.

At the time of writing this document, the compiler included in the Development Tools Module is GCC 14.2. It is important to note, however, that unlike the system compiler, the major version of the latest GCC from the module will change a few months after the upstream release of GCC 15.2 (scheduled for summer 2025), followed by GCC 16.2 (summer 2026), and so on. Note that only the most recent compiler in the Development Tools Module is supported at any time, with the exception of a six-month overlap period following an upgrade. Developers on a SUSE Linux Enterprise Server 15 system therefore have always access to two supported GCC versions: the almost unchanging system compiler and the most recent compiler from the Development Tools Module.

Programs and libraries built with the compiler from the Development Tools Module can run on computers running SUSE Linux Enterprise Server 15 which do not have the module installed. All necessary runtime libraries are available from the main repositories of the operating system itself, and new ones are added through the standard update mechanism. In this document, we use the term GCC 14 to refer to any minor version within the major version 14, while GCC 14.2 specifically refers to that particular version. In practice they should be interchangeable.

2.1 When to use compilers from the Development Tools Module

Often you will find that the system compiler perfectly satisfies your needs. After all, it is the compiler used to build the vast majority of packages and their updates in the system itself. On the other hand, there are situations where a newer compiler is necessary, or where you want to consider using a newer compiler to get some benefits of its ongoing development.

If the program or library you are building uses language features which are not supported by GCC 7, you cannot use the system compiler. However, the compiler from the Development Tools Module will usually be sufficiently new. The most obvious case is `C++`. GCC 14 has a mature implementation of `C++17` features, whereas the one in GCC 7 is only experimental and incomplete. The `GNU C++ Library` which accompanies GCC 14 is also `C++17` feature-complete.

Important: Code using C++17 features

Code using `C++17` features should always be compiled with the compiler from the Development Tools Module. Linking two objects, such as an application and a shared library, both using `C++17`—where one is built with `g++ 8` or earlier and the other with `g++ 9 or later`—is especially risky. This is because `C++` STL objects instantiated by the experimental code may provide implementation and even ABI that is different from what the mature implementation expects and vice versa. Issues caused by such a mismatch are difficult to predict and may include silent data corruption.

Most of `C++20` features are implemented in GCC 14 as experimental features. Try them out with appropriate caution and avoid linking together code that uses them and is produced by different compilers. *Modules* are only partially implemented [1] and require that the source file is compiled with `-fmodules-ts` option. Similarly, *coroutines* [2] are also implemented but require that the source file is compiled with the `-fcoroutines` switch. GCC 14 also experimentally implements many `C++23` and some `C++26` features. If you are interested in the implementation status of any particular `C++` feature in the compiler or the standard library, consult the following pages:

- `C++` Standards Support in GCC (https://gcc.gnu.org/projects/cxx-status.html) ↗, and
- The GNU `C++` Library Manual (https://gcc.gnu.org/onlinedocs/gcc-14.2.0/libstdc++/manual) ↗.

1 Proposals P1766R1 and P1815R2

2 Proposal P0912R5

Advances in supporting new language specifications are not limited to `C++`. GCC 14 experimentally supports most of the new features from the ISO `C23` standard, and the Fortran compiler is also continuously improved. And if you use `OpenMP` or `OpenACC` extensions for parallel programming, you will find that the compiler supports a lot of features of new versions of these standards. For more details, visit the links at the end of this section.

In addition to new supported language constructs, GCC 14 offers improved diagnostics when it reports errors and warnings to the user so that they are easier to understand and to be acted upon. This is particularly useful when dealing with issues in templated `C++ code`. Furthermore, there are several new warnings which help to avoid common programming mistakes.

Because GCC 14 is newer, it can generate code for many recent processors not supported by GCC 7. Such a list of processors would be too large to be enumerated here. Nevertheless, in *Section 7, "Performance evaluation: SPEC CPU 2017"* we specifically look at optimizing code for AMD EPYC 9005 Series Processors which are based on AMD Zen 5 cores. The *system compiler* does not know this kind of core and therefore cannot optimize for it. On the other hand, GCC 14.2 can both detect and optimize for Zen 5.

Finally, the general optimization pipeline of the compiler has also significantly improved over the years. To find out more about improvements in versions of GCC 8 through 14, visit their respective "changes" pages:

- GCC 8 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-8/changes.html) ↗,
- GCC 9 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-9/changes.html) ↗,
- GCC 10 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-10/changes.html) ↗,
- GCC 11 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-11/changes.html) ↗,
- GCC 12 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-12/changes.html) ↗,
- GCC 13 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-13/changes.html) ↗, and
- GCC 14 Release Series Changes, New Features, and Fixes (https://gcc.gnu.org/gcc-14/changes.html) ↗.

2.2 Potential issues with the Development Tools Module Compiler

GCC 14 from the Development Tools Module can sometimes behave differently in a way that can cause issues which were not present with the system compiler. Such problems encountered by other users are listed in the following documents:

- Porting to GCC 8 (https://gcc.gnu.org/gcc-8/porting_to.html),
- Porting to GCC 9 (https://gcc.gnu.org/gcc-9/porting_to.html),
- Porting to GCC 10 (https://gcc.gnu.org/gcc-10/porting_to.html),
- Porting to GCC 11 (https://gcc.gnu.org/gcc-11/porting_to.html),
- Porting to GCC 12 (https://gcc.gnu.org/gcc-12/porting_to.html),
- Porting to GCC 13 (https://gcc.gnu.org/gcc-13/porting_to.html), and
- Porting to GCC 14 (https://gcc.gnu.org/gcc-14/porting_to.html).

To get an understanding of the problems, read through these pages, all but the last one are fairly short. The document at hand briefly mentions a few most common potential pitfalls.

Starting with GCC 14, the `C` compiler treats some situations which were never allowed since the 1999 revision ISO `C` as errors. In GCC 13 and before, the compiler only generated warnings for them:

- Implicit int types (`-Werror=implicit-int`)
- Implicit function declarations (`-Werror=implicit-function-declaration`),
- 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 strongly recommend that you take the time to fix any of the above problems if you encounter them in your code. They have been a frequent source of bugs, portability and even security issues. More information about all of these cases together with the most common ways of addressing them are given in the “Porting to GCC 14” document referenced above. If the code is written in a version of `C` before the 1999 ISO standard, you can tell the compiler by using the `-std=gnu89` or `-std=c89` option, which will again allow those constructs. If your code uses features of this standard or a later one and for some reason it is not possible to fix it, you can either turn a specific class of the new errors back to warnings with a corresponding `-Wno-error=` option or use a new compiler switch `-fpermissive` to do so for all of the above.

Note: Impact on build environment probing

Many code snippets (also called *probes*) generated by `autoconf` to discover the availability of various features work in the way that they trigger a compile error when a feature is missing. The new errors may cause compilation to fail when it worked before and thus lead to features being silently disabled even when 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 that were removed in C99 and thus fail with GCC 14. In cases where this is a concern, you can compare the generated `config.log`, `config.h` and other generated files using **`diff`** to ensure there are no unexpected differences.

The second common pitfall is that GCC 10 and later 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 happen if two or more `.c` files include the same header file which intends to declare a variable but omits the `extern` keyword when doing so, inadvertently resulting in 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 also be aware that `g++` version 11 and later default to `-std=gnu++17`, the `C++17` standard with GNU extensions, instead of `-std=gnu++14`. Moreover, some `C++` Standard Library headers have been changed to no longer include other headers that 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.

2.3 Installing GCC 14 from the Development Tools Module

Similar to other modules and extensions for SUSE Linux Enterprise Server 15, you can activate the Development Tools Module using either the command line tool **SUSEConnect** or the **YaST** setup and configuration tool. To use the former, carry out the following steps:

1. As root, start by listing the available and activated modules and extensions:

```
sles15: # SUSEConnect --list-extensions
```

2. In the computer output, look for “Development Tools Module”:

```
            Development Tools Module 15 SP6 x86_64
            Activate with: suseconnect -p sle-module-development-tools/15.6/x86_64
```

   If you see the text `(Activated)` next to the module name, the module is already ready to be used. You can safely proceed to the installation of the compiler packages.

3. Otherwise, issue the activation command that is shown in the command output above:

```
sles15: # suseconnect -p sle-module-development-tools/15.6/x86_64
Registering system to SUSE Customer Center

Updating system details on https://scc.suse.com ...

Activating sle-module-development-tools 15.6 x86_64 ...
-> Adding service to system ...
-> Installing release package ...

Successfully registered system
```

If you prefer to use **YaST**, the procedure is also straightforward. Run YaST as root and go to the **Add-On Products** menu in the **Software** section. If “Development Tools Module” is among the listed installed modules, you already have the module activated and can proceed with installing individual compiler packages. If not, click the **Add** button, select **Select Extensions and Modules from Registration Server**, and **YaST** will guide you through a simple procedure to add the module.

When you have the Development Tools Module installed, you can verify that the GCC 14 packages are available to be installed on your system:.

```
sles15: # zypper search gcc14
Refreshing service 'Basesystem_Module_15_SP6_x86_64'.
Refreshing service 'Certifications_Module_15_SP6_x86_64'.
```

```
Refreshing service 'Containers_Module_15_SP6_x86_64'.
Refreshing service 'Desktop_Applications_Module_15_SP6_x86_64'.
Refreshing service 'Development_Tools_Module_15_SP6_x86_64'.
Refreshing service 'Python_3_Module_15_SP6_x86_64'.
Refreshing service 'SUSE_Linux_Enterprise_Server_15_SP6_x86_64'.
Refreshing service 'SUSE_Package_Hub_15_SP6_x86_64'.
Refreshing service 'Web_and_Scripting_Module_15_SP6_x86_64'.
Loading repository data...
Reading installed packages...

S  | Name                      | Summary
---+---------------------------+----------------------------------------------------------
   | gcc14                     | The GNU C Compiler and Support Files
   | gcc14                     | The GNU C Compiler and Support Files
   | gcc14-32bit               | The GNU C Compiler 32bit support
   | gcc14-ada                 | GNU Ada Compiler Based on GCC (GNAT)
   | gcc14-ada-32bit           | GNU Ada Compiler Based on GCC (GNAT)
   | gcc14-c++                 | The GNU C++ Compiler
   | gcc14-c++-32bit           | The GNU C++ Compiler
   | gcc14-d                   | GNU D Compiler
   | gcc14-d-32bit             | GNU D Compiler
   | gcc14-fortran             | The GNU Fortran Compiler and Support Files
   | gcc14-fortran-32bit       | The GNU Fortran Compiler and Support Files
   | gcc14-go                  | GNU Go Compiler
   | gcc14-go-32bit            | GNU Go Compiler
   | gcc14-info                | Documentation for the GNU compiler collection
   | gcc14-locale              | Locale Data for the GNU Compiler Collection
   | gcc14-m2                  | GNU Modula-2 Compiler
   | gcc14-m2-32bit            | GNU Modula-2 Compiler
   | gcc14-obj-c++             | GNU Objective C++ Compiler
   | gcc14-obj-c++-32bit       | GNU Objective C++ Compiler
   | gcc14-objc                | GNU Objective C Compiler
   | gcc14-objc-32bit          | GNU Objective C Compiler
   | gcc14-PIE                 | A default configuration to build all binaries in PIE mode
   | libquadmath0-devel-gcc14  | The GNU Fortran Compiler Quadmath Runtime Library Develop
   | libstdc++6-devel-gcc14    | Include Files and Libraries mandatory for Development
```

Now you can install the compilers for the programming languages you use with **`zypper`**:

```
sles15: # zypper install gcc14 gcc14-c++ gcc14-fortran
```

The compilers are installed on your system, the executables are called **`gcc-14`**, **`g++-14`**, **`gfortran-14`** and so forth. It is also possible to install the packages in **`YaST`**. To do so, enter the “Software Management” menu in the **Software** section and search for “gcc14”. Then select the packages you want to install. Finally, click the **Accept** button.

Note: Newer compilers on openSUSE Leap 15.6

The community distribution openSUSE Leap 15.6 shares the base packages with SUSE Linux Enterprise Server 15 SP6. The system compiler on systems running openSUSE Leap 15.6 is also GCC 7.5. There is no Development Tools Module for the community distribution available, but a newer compiler is provided. Install the packages `gcc14`, `gcc14-c++`, `gcc14-fortran`, and the like.

3 Optimization levels and related options

GCC has a rich optimization pipeline that is controlled by approximately a hundred of command line options. It would be impractical to force users to decide about each one of them whether they want to have it enabled when compiling their code. Like all other modern compilers, GCC therefore introduces the concept of optimization levels which allow the user to pick a configuration from a few common ones. Optionally, the user can tweak the selected level, but that does not happen frequently.

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 means it is usually accompanied with the command line switch `-g` so that debug information is emitted. As no optimizations take place, no information is lost because of it. No variables are optimized away, the compiler only inlines functions with special attributes that require it, and so forth. As a consequence, the debugger can almost always find everything it searches for in the running program and report on its state very well. On the other hand, the resulting code is big and slow. Thus this optimization level should not be used for release builds.

The most common optimization level for release builds is `-O2` which 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, even if the resulting code might be considerably bigger and the compilation can take longer. Note that neither `-O2` nor `-O3` imply anything about the precision and semantics of floating-point operations. Even at the optimization level `-O3` GCC implements math operations and functions so that they follow the respective IEEE and/or ISO rules [3] with the exception of allowing floating-point expression contraction, for example when fusing an addition and a multiplication into one operation[4]. This often means that the

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

4 See documentation of `-ffp-contract.`

compiled programs run markedly slower than necessary if such strict adherence is not required. The command line switch `-ffast-math` is a common way to relax rules governing floating-point operations. It is out of scope of this document to provide a list of the fine-grained options it enables and their meaning. However, if your software crunches floating-point numbers and its runtime is a priority, you can look them up in the GCC manual and review what semantics of floating-point operations you need.

The most aggressive optimization level is `-Ofast` which does imply `-ffast-math` along with a few options that disregard strict standard compliance. In GCC 14, this level also means the optimizers may introduce data races when moving memory stores which may not be safe for multithreaded applications, and disregards the possibility of ELF symbol interposition happening at runtime. 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, you can use **`ulimit -S`** with a sufficiently high limit, or **`ulimit -S unlimited`**. We often use level `-Ofast` to build benchmarks. It is a shorthand for the options on top of `-O3` which often make them run faster. Most benchmarks are intentionally written in a way that they run correctly even when these rules are relaxed.

If you feed the compiler with huge machine-generated input, especially if individual functions happen to be extremely large, the compile time can become an issue even when using `-O2`. In such cases, use the most lightweight optimization level `-O1` that avoids 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

Usually we recommend using `-O2`. This is the optimization level we use to build most SUSE and openSUSE packages, because 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 you know that your project is compute-intensive and is either small or an important part of your actual workload. Moreover, if the compiled code contains performance-critical floating-point operations, we strongly advise that you investigate whether `-ffast-math` or any of the fine-grained options it implies can be safely used.

If your project and the techniques you use to debug or instrument it do not depend on *ELF symbol interposition*, you may consider trying to speed it up by using `-fno-semantic-interposition`. This 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 is not going to happen is known to significantly boost performance of some projects, most notably the Python interpreter.

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 very powerful tool. It is used to enable other transformations, such as store-to-load propagation that in turn enables other high level optimizations, such as aggressive inlining, vectorization and others.

With the `-g` switch GCC tries hard to generate useful debug information even when optimizing. However, a lot of information is irrecoverably lost in the process. Debuggers also often struggle to present the user with 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 a challenging task but usually is still somewhat possible.

The complete list of optimization and other command line switches is available in the compiler manual. The manual is provided in the info format in the package `gcc14-info` or online at the GCC project Web site (https://gcc.gnu.org/onlinedocs/gcc-14.2.0/gcc/).

Keep in mind that while nearly all optimizing compilers have optimization levels, and these levels often share the same names as those in GCC, they don't necessarily involve the same trade-offs. Famously, GCC's `-Os` optimizes for size much more aggressively than LLVM/Clang's level with 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 much bigger 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` which might not be what you want. To change it, you must modify the `CMAKE_C_FLAGS_RELEASE`, `CMAKE_CXX_FLAGS_RELEASE` and/or `CMAKE_Fortran_FLAGS_RELEASE` variables. Since they are appended at the end of the compilation command lines, they are overwriting any level set in the variables `CMAKE_C_FLAGS`, `CMAKE_CXX_FLAGS`, and the like.

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. On architectures like `x86_64` and `aarch64` the generated code will only contain instructions available on every CPU model of the architecture, including the earliest ones. On `x86_64` in particular this means that the programs will use the `SSE` and `SSE2` instruction sets for floating-point and vector operations but not any more recent ones.

If you know that the generated binary will run only on machines supporting newer instruction set extensions, you can specify it on the command line. Their complete list is available in the manual, but the most prominent one is `-march` which lets you select a CPU model to generate code for. For example, if you know that your program will only be executed on AMD EPYC 9005 Series Processors based on AMD Zen 5 cores or processors that are compatible with it, you can instruct GCC to take advantage of all the instructions the CPU supports with option `-march=znver5`. Note that, on SUSE Linux Enterprise Server 15, the system compiler does not know this particular value of the switch. You need to use GCC 14 from the Development Tools Module to optimize code for these processors.

To run the program on the machine on which you are compiling it, you can have the compiler auto-detect the target CPU model for you with the option `-march=native`. This only works if the compiler is new enough. The system compiler of SUSE Linux Enterprise Server, for example, misidentifies AMD EPYC 9005 Series Processors as being based on the AMD Zen 1 core. Among other things, this means that it only emits 128 bit vector instructions, even though the CPU has data-paths wide enough to efficiently execute 512 bit ones. Again, the easy solution is to use the compiler from the Development Tools Module when targeting recent processors.

Note: Running 32-bit code

SUSE Linux Enterprise Server does not support compilation of 32-bit applications, it only offers runtime support for 32-bit binaries. To do so, you will need 32-bit libraries your binary depends on which likely include at least glibc which can be found in package `glibc-32bit`. See chapter 20 (32-bit and 64-bit applications in a 64-bit system environment) of the Administration Guide (https://documentation.suse.com/sles/15-SP6/html/SLES-all/cha-64bit.html) ↗ for more information.

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 such low level, it cannot perform much optimization and the division of the program into compilation units thus presents a profound barrier to optimization.

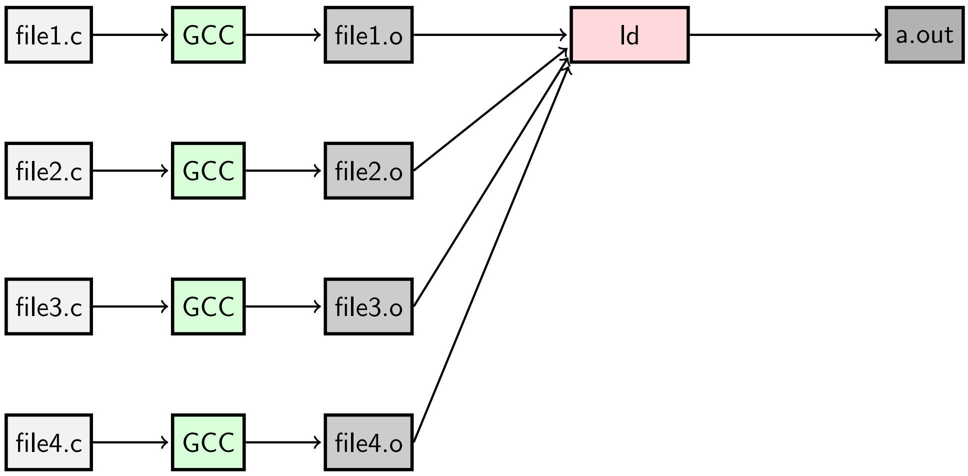

FIGURE 1: TRADITIONAL PROGRAM BUILD

This limitation can be overcome by rearranging the process so that the linker does not receive as its input the almost finished object files containing machine instructions, but is invoked on files containing so called *intermediate language* (IL). This is a much richer representation of each original compilation unit (see figure *figure 2*). The linker identifies the input as not yet entirely compiled and invokes a linker plugin which in turn runs the compiler again. But this time it has at its disposal the representation of the entire program or library that is being built. The compiler makes decisions about what optimizations across function and compilation unit boundaries will be carried out and then divides the program into a set of partitions. Each of the partitions is further optimized independently, and machine code is emitted for it, which is finally linked the traditional way. Processing of the partitions is performed in parallel.

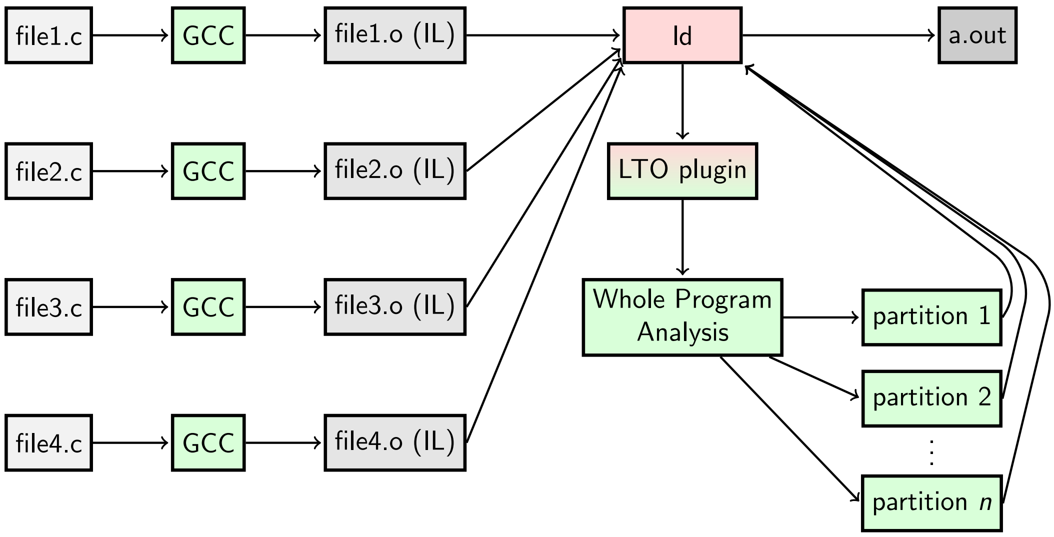

FIGURE 2: BUILDING A PROGRAM WITH GCC USING LINK TIME OPTIMIZATION (LTO)

To use **Link Time Optimization**, all you need do is to add the `-flto` switch to the compilation command line. The vast majority of packages in the Linux distribution openSUSE Tumbleweed has been built with LTO for over five years without any major problems. A lot of work has been put into emitting good debug information when building with LTO too. Thus the debugging experience is not severely limited anymore as it was seven years ago.

LTO in GCC always consists of a *whole program analysis* (WPA) stage followed by the majority of the compilation process performed in parallel, which greatly reduces the build times of most projects. To control the parallelism, you can explicitly cap the number of parallel compilation processes by *n* if you specify `-flto=n` at linker command line. Alternatively, it is possible to use the GNU **`make`** jobserver with `-flto=jobserv` while also prepending the **makefile** rule invoking link step with character `+` to instruct GNU make to keep the jobserver available to the linker process. However, this modification of `makefiles` is not necessary with make version 4.4 or newer.

You can also use `-flto=auto` which instructs GCC to search for the jobserver and 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 which does not allow for any parallelism. GCC can be configured to operate in this way with the option `-flto-partition=one`. LLVM in thin LTO mode can compile different compilation units in parallel and makes possible inlining across compilation unit boundaries, but not most other types of cross-module optimizations. This mechanism therefore has inherently higher code quality penalty than full LTO or the approach of GCC.

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. Since 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 then subsequently 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. Thanks to *symbol promotion* and inter-procedural *unreachable code elimination*, functions and their parts which are not necessary in any particular project can be removed even when they are not declared `static` and are not defined in an anonymous namespace. Automatic *attribute discovery* can identify `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* can find functions with the same semantics and remove 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. Similar problems happen even when building static libraries with `-rdynamic`.

5.2 Potential issues with LTO

As mentioned earlier, the vast majority of packages in the openSUSE Tumbleweed distribution are built with LTO by default and work fine without any tweaks. Nevertheless, some low-level constructs pose a problem for LTO. One typical issue are symbols defined in *inline assembly* which can happen to be placed in a different partition from their uses and subsequently fail the final linking step. To build such projects with LTO, the assembler snippets defining symbols must be placed into a separate assembler source file so that they only participate in the final linking step. Global `register` variables are not supported by LTO, and programs either must not use this feature or be built the traditional way. You can also exclude some compilation units from LTO (by compiling them without `-flto` or appending `-fno-lto` to the compilation command line), while the rest of the program can still benefit from using this feature.

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, you can 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). Consult 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 which do not manifest themselves otherwise. Violations of (strict) *aliasing* rules and `C++` *one definition rule* tend to cause misbehavior significantly more often. The latter is fortunately reported by the `-Wodr` warning which is on by default and should not be ignored. We have also seen cases where the use of the `flatten` function attribute led to unsustainable amount of inlining with LTO. Furthermore, LTO is not a good fit for code snippets compiled by `configure` scripts (generated by `autoconf`) to discover the availability of various features, especially when the script then searches for a string in the generated assembly.

Finally, we needed to configure the virtual machines building the biggest openSUSE packages to have more memory than when not using LTO. Whereas in the traditional mode of compilation 1 GB of RAM per core was enough to build Mozilla Firefox, the serial step of LTO means the build-bots need 16 GB even when they have fewer than 16 cores.

6 Profile-Guided Optimization (PGO)

Optimizing compilers frequently make decisions that depend on which path through the code they consider most likely to be executed, how many times a loop is expected to iterate, and similar estimates. They also often face trade-offs between potential runtime benefits and code size growth. Ideally, they would optimize only frequently executed (also called *hot*) bits of a program for speed and everything else for size to reduce strain on caches and make the distribution of the built software cheaper. Unfortunately, guessing which parts of a program are the *hot* ones is difficult, and even sophisticated estimation algorithms implemented in GCC are no match for a 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. The first time it needs to compile with the `-fprofile-generate` option and then execute the resulting binary in one or multiple *train runs* during which it will save information about the behavior of the program 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, a process 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, it needs to exercise the code that is also the most frequently executed in real use, otherwise it will be optimized for size and PGO would make more harm than good. With the option, GCC reverts to guessing properties of portions of the projects not exercised in the train run, as if they were compiled without profile feedback. This however also means that this code will not perform better or shrink as much as one would expect 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 disproportionally 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 and is complimentary to Link Time Optimization. While LTO expands what the compiler can do, PGO informs it about which parts of the program are the important ones and should be focused on. The case study in the following section shows how the two techniques work with each other 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 latest suite of CPU intensive workloads, 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 14 evaluation using the suite.

Note that when we use SPEC to perform compiler comparisons, we are lenient toward some official SPEC rules which system manufacturers need to observe to claim an official score for their system. We disregard the concepts of *base* and *peak* metrics and focus on results of compilations using a particular set of options. We even patched several benchmarks:

- Benchmarks `502.gcc_r`, `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`.

- Benchmark `511.povray_r` cannot be built with option `-fno-finite-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 14 or Clang 19 in this section therefore append `-fno-finite-math-only` to the compilation command lines, but again only for this one 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*.

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 9755 Processor running SUSE Linux Enterprise Server 15 SP6.

7.1 Benefits of LTO and PGO

In *Section 3, "Optimization levels and related options"* we recommend that HPC workloads are compiled with `-O3` and benchmarks with `-Ofast`. But it is still interesting to look at integer crunching benchmarks built with only `-O2` because that is how Linux distributions often build the programs from which they were extracted. We have already mentioned that almost the whole openSUSE Tumbleweed distribution is now built with LTO, and selected packages with PGO, and the following paragraphs demonstrate why.

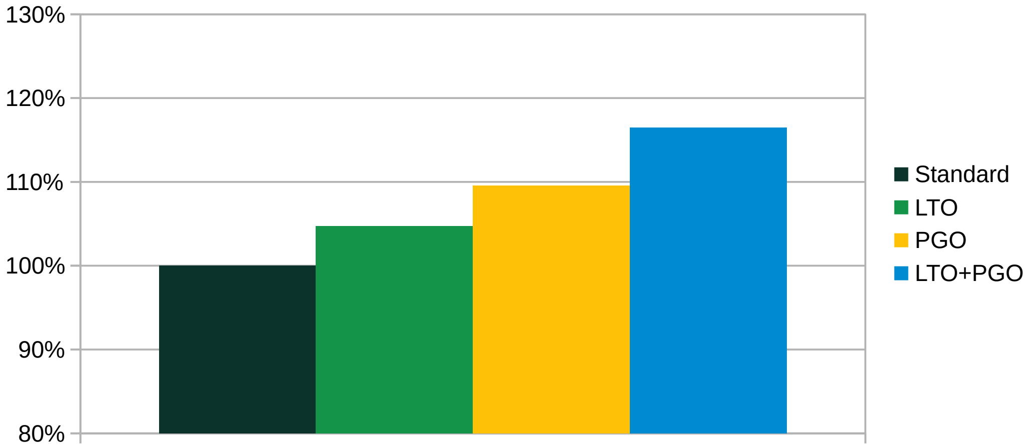

FIGURE 3: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC INTRATE 2017 BUILT WITH GCC 14.2 AND -O2

Figure 3 shows the overall performance effect on the whole integer benchmark suite as captured by the geometric mean of all individual benchmark rates. Employing both PGO and LTO results in remarkable relative uplift of 16.5%. That is despite the fact that starting with GCC 12, the compiler can conservatively auto-vectorize code in `525.x264_r` also at plain `-O2`, whereas previously it was only automatically performed with PGO at this level. Nevertheless, this benchmark still benefits a lot from the more advanced modes of compilation, together with several others which are derived from programs that are typically compiled with `-O2`. This is illustrated in *figure 4*.

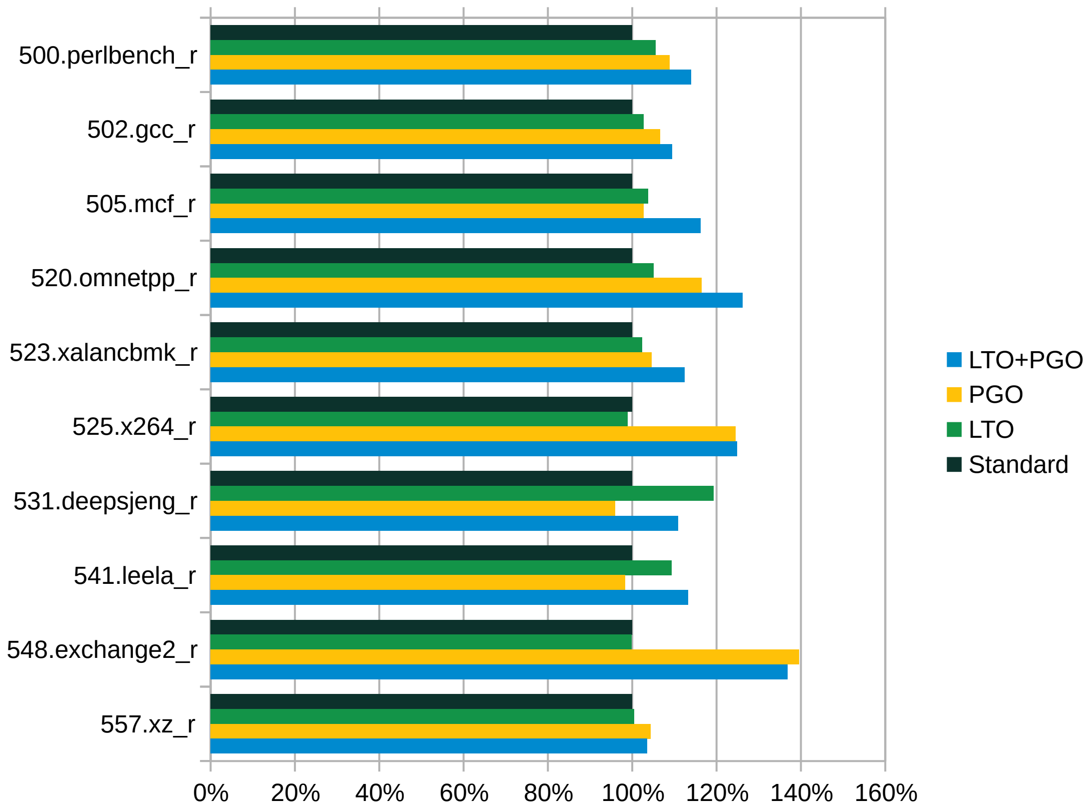

FIGURE 4: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF INDIVIDUAL INTEGER BENCHMARKS BUILT WITH GCC 14.2 AND -O2

Figure 5 shows another important advantage of LTO and PGO which is significant reduction of the size of the binaries (measured without debug info). Note that it does not depict that the size of benchmark `548.exchange2_r` grew to 260% and 174% of the original size when built with PGO or both PGO and LTO respectively, which looks huge but the growth is from a particularly small base. It is the only Fortran benchmark in the integer suite and, most importantly, the size penalty is offset by significant speed-up, making the trade-off reasonable. For completeness, we show this result in *figure 6*

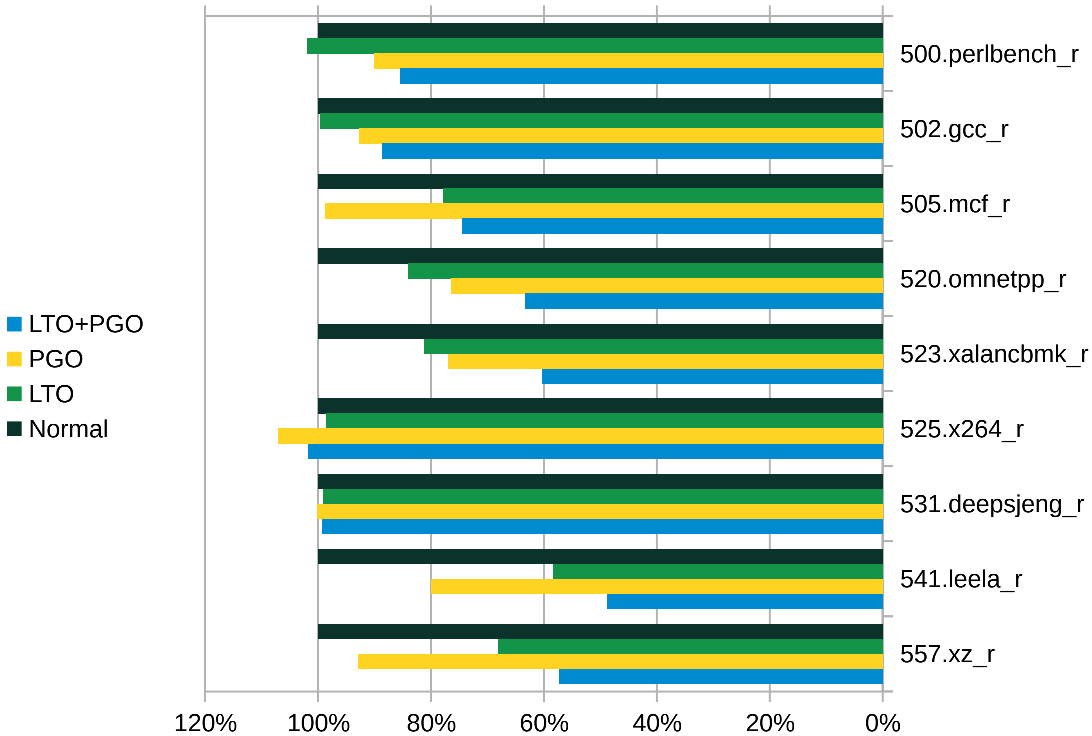

FIGURE 5: BINARY SIZE (SMALLER IS BETTER) OF INDIVIDUAL INTEGER BENCHMARKS BUILT WITH GCC 14.2 AND -O2

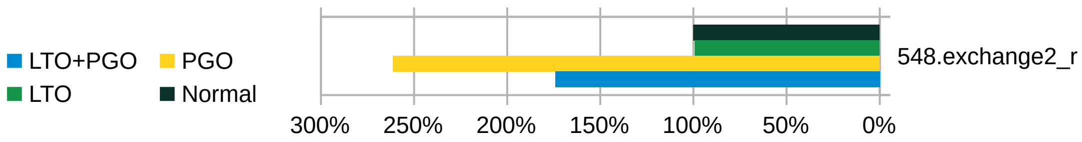

FIGURE 6: BINARY SIZE (SMALLER IS BETTER) OF 548.EXCHANGE2_R BUILT WITH GCC 14.2 AND -O2

The runtime benefits and binary size savings are also easily visible when using the optimization level `-Ofast` and option `-march=native` to allow the compiler to take full advantage of all instructions that the AMD EPYC 9755 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 about clear benefits for benchmarks derived from interpreters and compilers like `500.perlbench_r` and `502.gcc_r` but the compiler can struggle to correctly update the measured profile information when performing complex inter-procedural optimizations like in the case of `548.exchange2_r` leading to the technique actually decreasing performance. Lastly, even though optimization levels `-O3` and `-Ofast` are permitted to be relaxed about the final binary size, PGO and especially LTO can bring it nicely down at these levels, too. *Figure 9* depicts the relative binary sizes of all integer benchmarks.

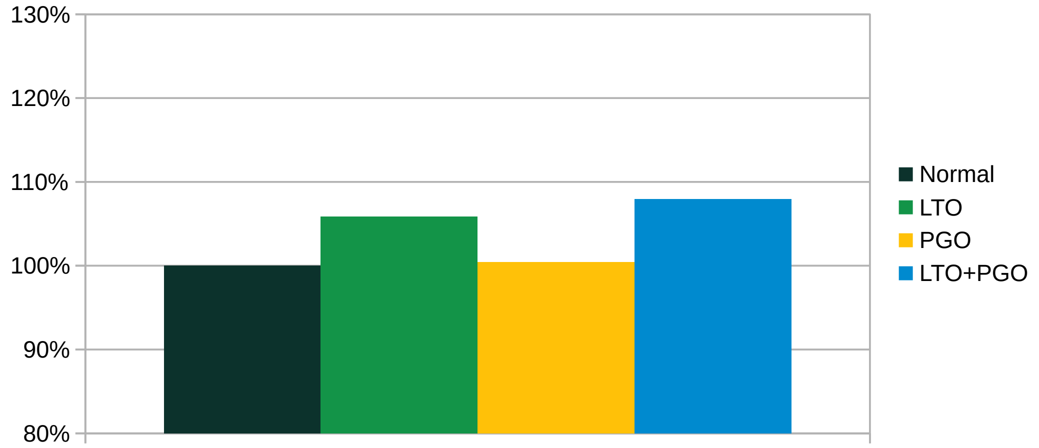

FIGURE 7: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC INTRATE 2017 BUILT WITH GCC 14.2 USING -OFAST AND -MARCH=NATIVE

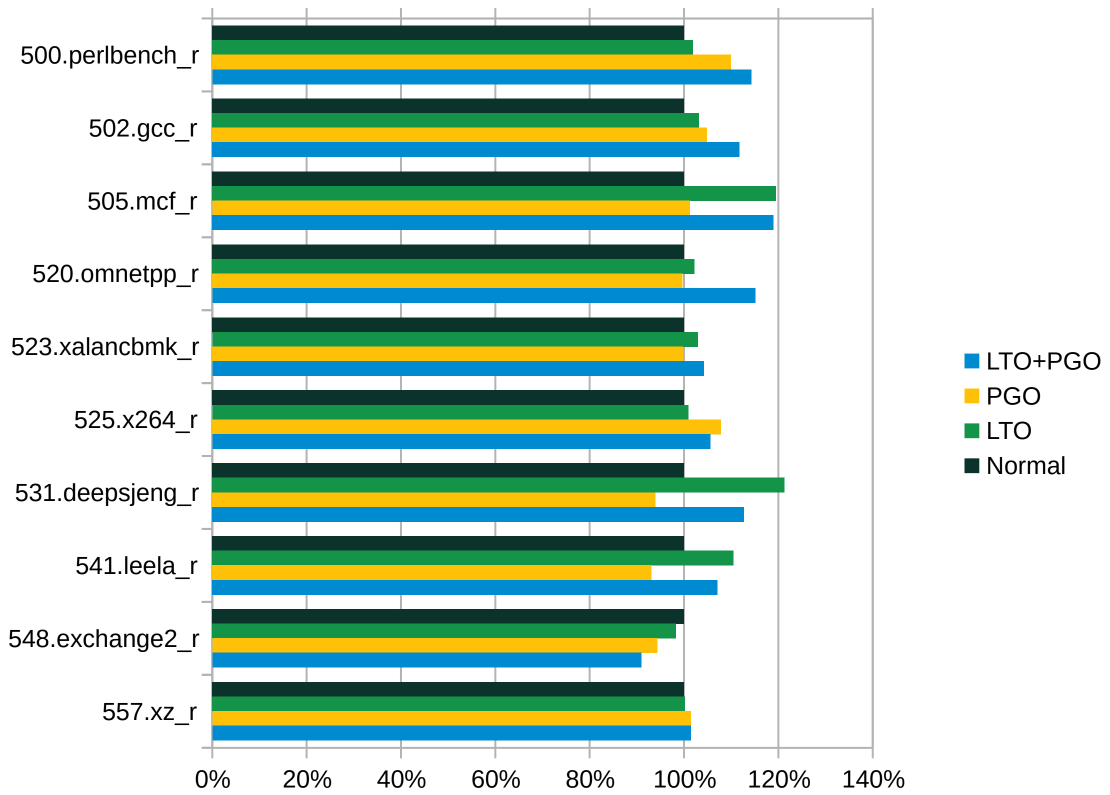

FIGURE 8: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF INDIVIDUAL INTEGER BENCHMARKS BUILT WITH GCC 14.2 USING -OFAST AND -MARCH=NATIVE

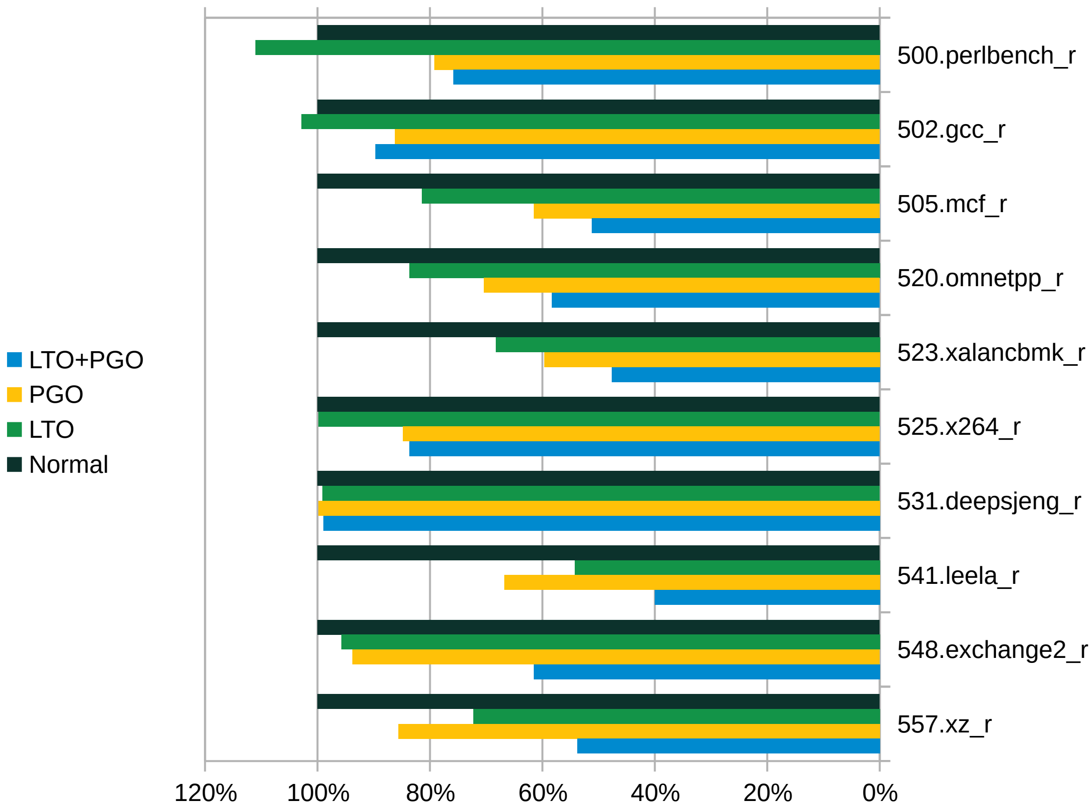

FIGURE 9: BINARY SIZE (SMALLER IS BETTER) OF SPEC INTRATE 2017 BUILT WITH GCC 14.2 USING -OFAST AND -MARCH=NATIVE

Many of the SPEC 2017 floating-point benchmarks measure how well a given 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 about 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 9755 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*

Unfortunately, in the case of `538.imagick_r` benchmark there is a big mismatch in between the code paths exercised in the train run which is used to measure which parts of the program need to be optimized for speed and the actual reference run which is then used to obtain the benchmark score. This is exactly the problem we warn against in *Section 6, "Profile-Guided Optimization (PGO)"* and it has the predictable detrimental effect on performance.[5] Moreover, because the important loop, which is not appropriately optimized because it is not executed in the train run, is in a function in which there is another loop which 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 and it means that the overall performance score even decreases by 1% when using both LTO and PGO.

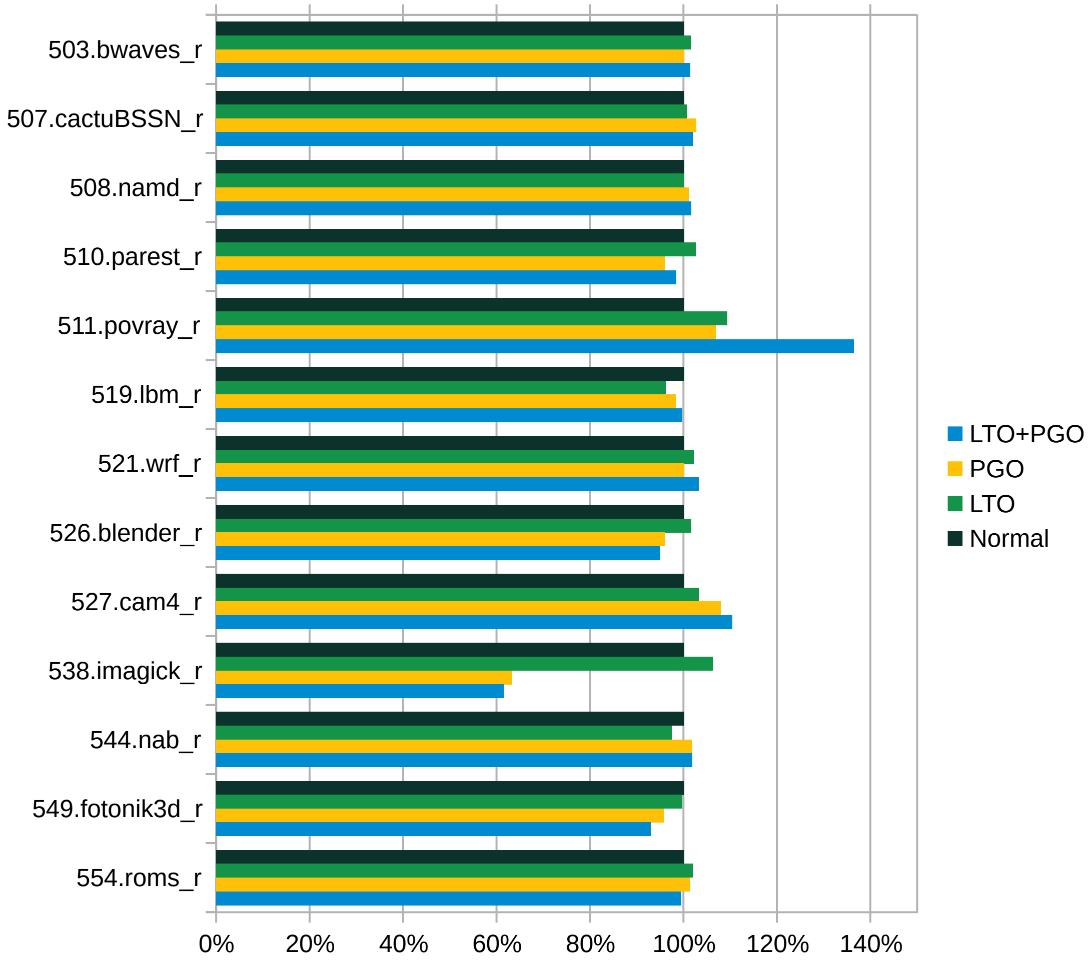

FIGURE 10: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF INDIVIDUAL FLOATING-POINT BENCHMARKS BUILT WITH GCC 14.2 USING -OFAST AND -MARCH=NATIVE

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

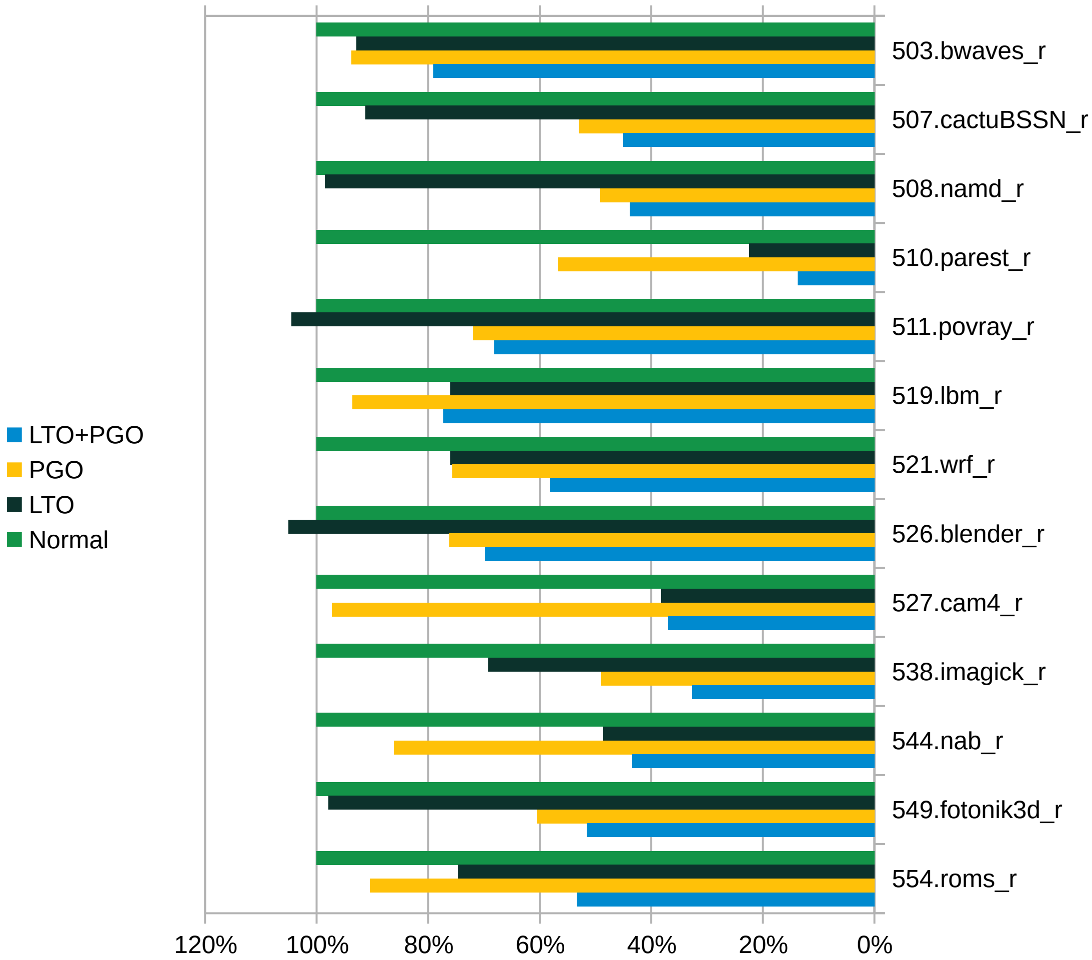

FIGURE 11: BINARY SIZE (SMALLER IS BETTER) OF SPEC FPRATE 2017 BUILT WITH GCC 14.2 USING -OFAST AND -MARCH=NATIVE

7.2 GCC 14.2 compared to GCC 7.5

In previous sections we have recommended the use of GCC 14.2 from the Development Tools Module over the system compiler. Among other reasons, we did so because of its more powerful optimization pipeline and its support for newer CPUs. This section compares SPEC CPU 2017 built with GCC 7.5, the system compiler in SUSE Linux Enterprise Server 15, and GCC 14.2 on an AMD EPYC 9755 Processor, when all benchmarks are compiled with `-Ofast` and `-march=native`. Note that the latter option means that both compilers differ in their CPU targets because GCC 7.5 does not know the Zen 5 core. This in turn means that in large part the optimization benefits presented here exist because the old compiler only issues 128bit (AVX2) vector operations whereas the newer one can take full advantage of AVX512. Nevertheless, be aware that using wider vectors everywhere often backfires. GCC has made substantial advancements over the recent years to avoid such issues, both in its vectorizer and other optimizers. It is therefore much better placed to use the extra vector width appropriately and produce code which utilizes the processor better in general.

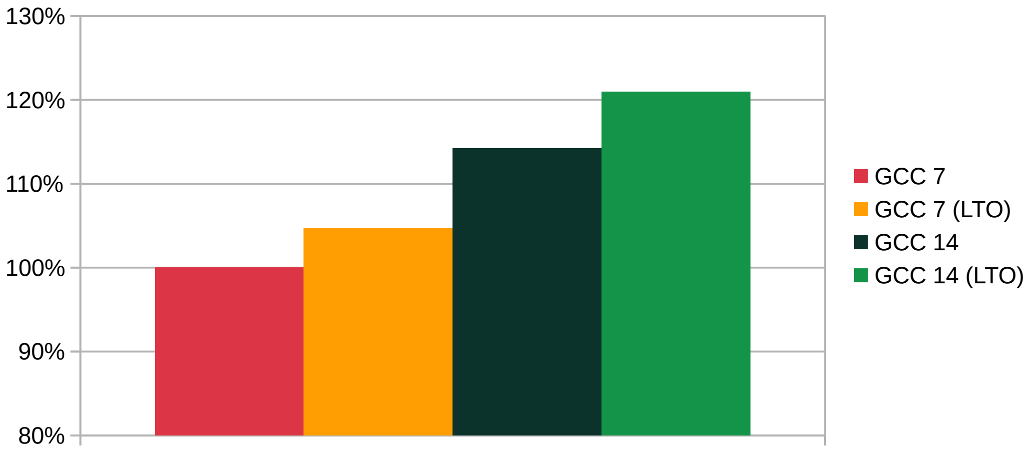

FIGURE 12: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC INTRATE 2017 BUILT WITH GCC 7.5 AND 14.2 (-OFAST -MARCH=NATIVE)

Figure 12 captures the benefits of using the modern compiler with integer workloads in the form of relative improvements of the geometric mean of the whole SPEC INTrate 2017 suite. *Figure 13* dives deeper and shows which particular benchmarks gained most in terms of performance. It was already mentioned that `525.x264_r` especially benefits from vectorization and therefore it is not surprising it has improved a lot. `531.deepsjeng_r` is faster chiefly because it can emit better code for *count trailing zeros* (CTZ) operation which it performs frequently. Finally, modern GCC can optimize `548.exchange2_r` particularly well by specializing different invocations of the hottest recursive function and it also clearly shows in the picture.

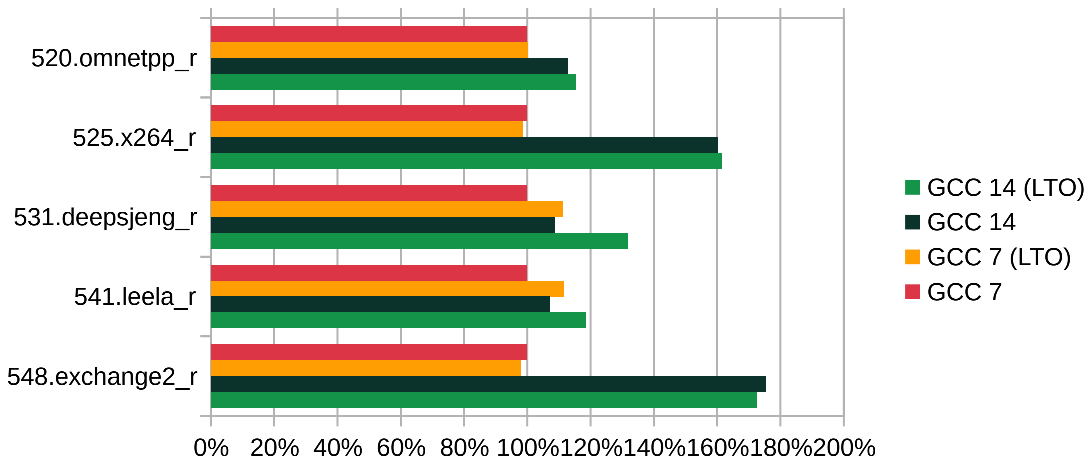

FIGURE 13: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF SELECTED INTEGER BENCHMARKS BUILT WITH GCC 7.5 AND 14.2 (-OFAST -MARCH=NATIVE)

Floating-point computations tend to particularly benefit from vectorization advancements. Thus it should be no surprise that the FPrate benchmarks also improve substantially when compiled with GCC 14.2, which also emits AVX512 instructions for a Zen 5 based CPU. The overall boost is shown in *figure 14* whereas *figure 15* provides a detailed look at which benchmarks contributed most to the overall score difference.

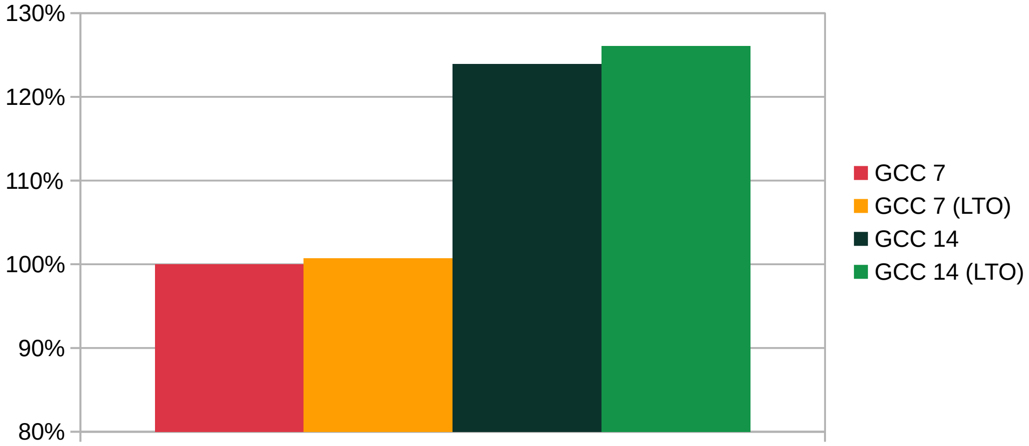

FIGURE 14: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC FPRATE 2017 BUILT WITH GCC 7.5 AND 14.2 (-OFAST -MARCH=NATIVE)

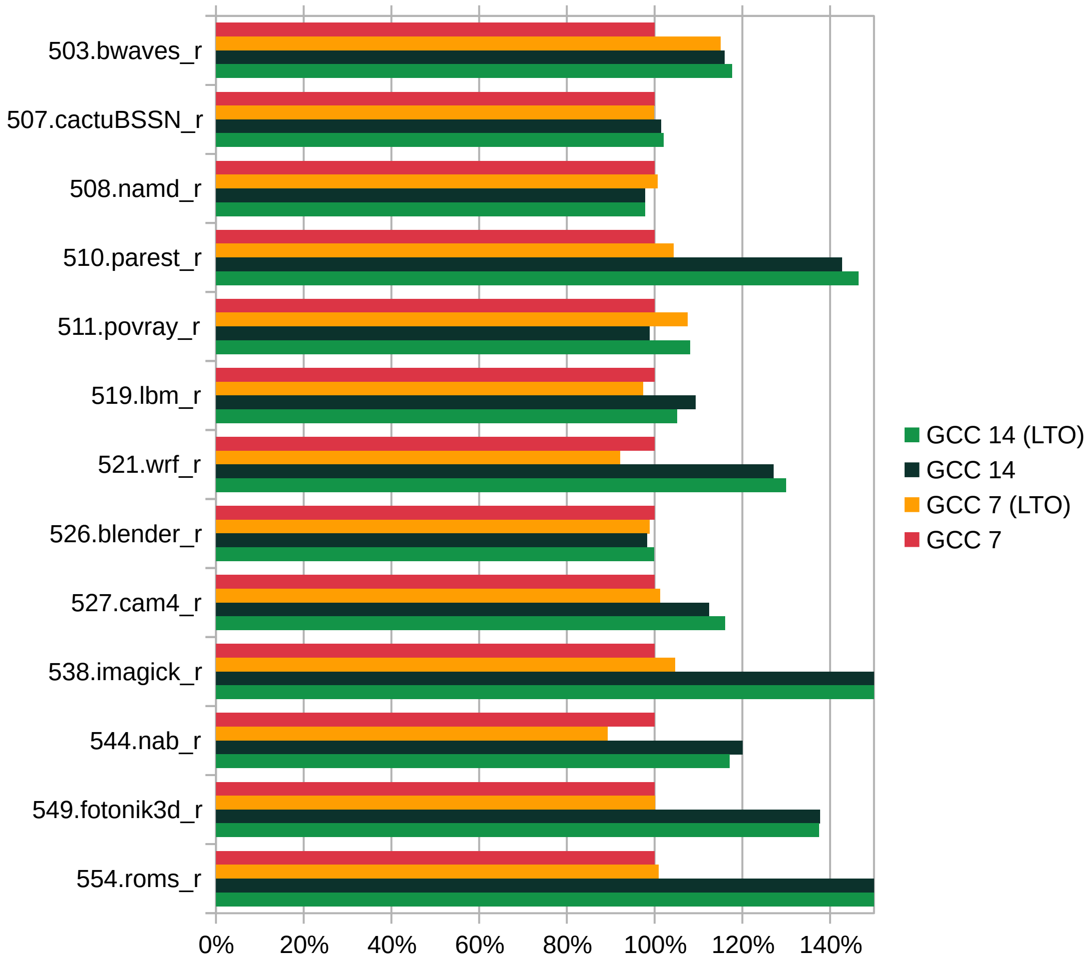

FIGURE 15: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF SELECTED FLOATING-POINT BENCHMARKS BUILT WITH GCC 7.5 AND 14.2 (-OFAST -MARCH=NATIVE)

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

In *Section 3, "Optimization levels and related options"*, we highlighted that if you do not relax the semantics of floating-point math functions, despite not needing strict adherence to all IEEE and/or ISO rules, you are likely to sacrifice some performance. This section uses the SPEC FPrate 2017 test suite to illustrate how much performance that might be.

We have built the benchmarking suite using optimization level `-O3`, LTO (though without PGO) and `-march=native` to target the native ISA of our AMD EPYC 9755 Processor. Then we compared its runtime score against the suite built with these options and `-ffast-math`. As you can see in *figure 16*, the geometric mean grew by over 13%. But a quick look at *figure 17* will tell you that there are four benchmarks with scores which improved by more than 15% and that of `538.imagick_r` grew by over 60%.

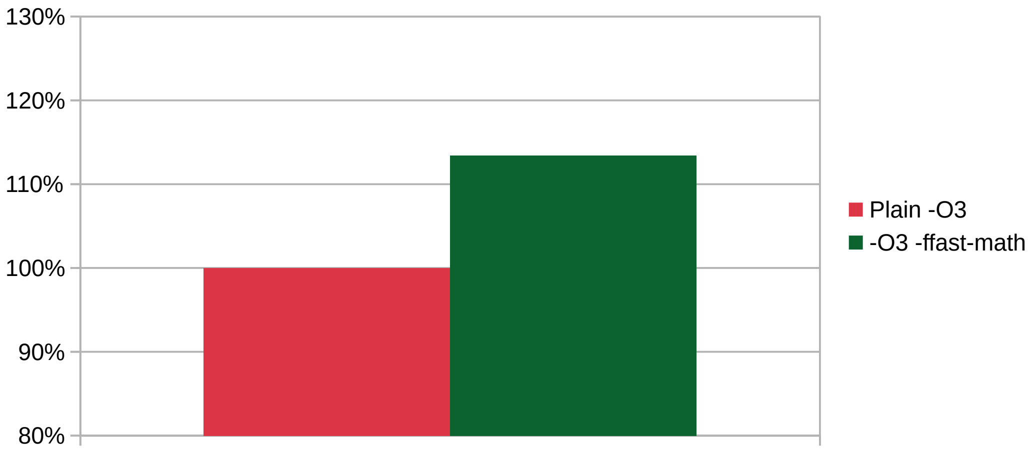

FIGURE 16: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC FPRATE 2017 BUILT WITH GCC 14.2 AND -O3 -FLTO -MARCH=NATIVE, WITHOUT AND WITH -FFAST-MATH

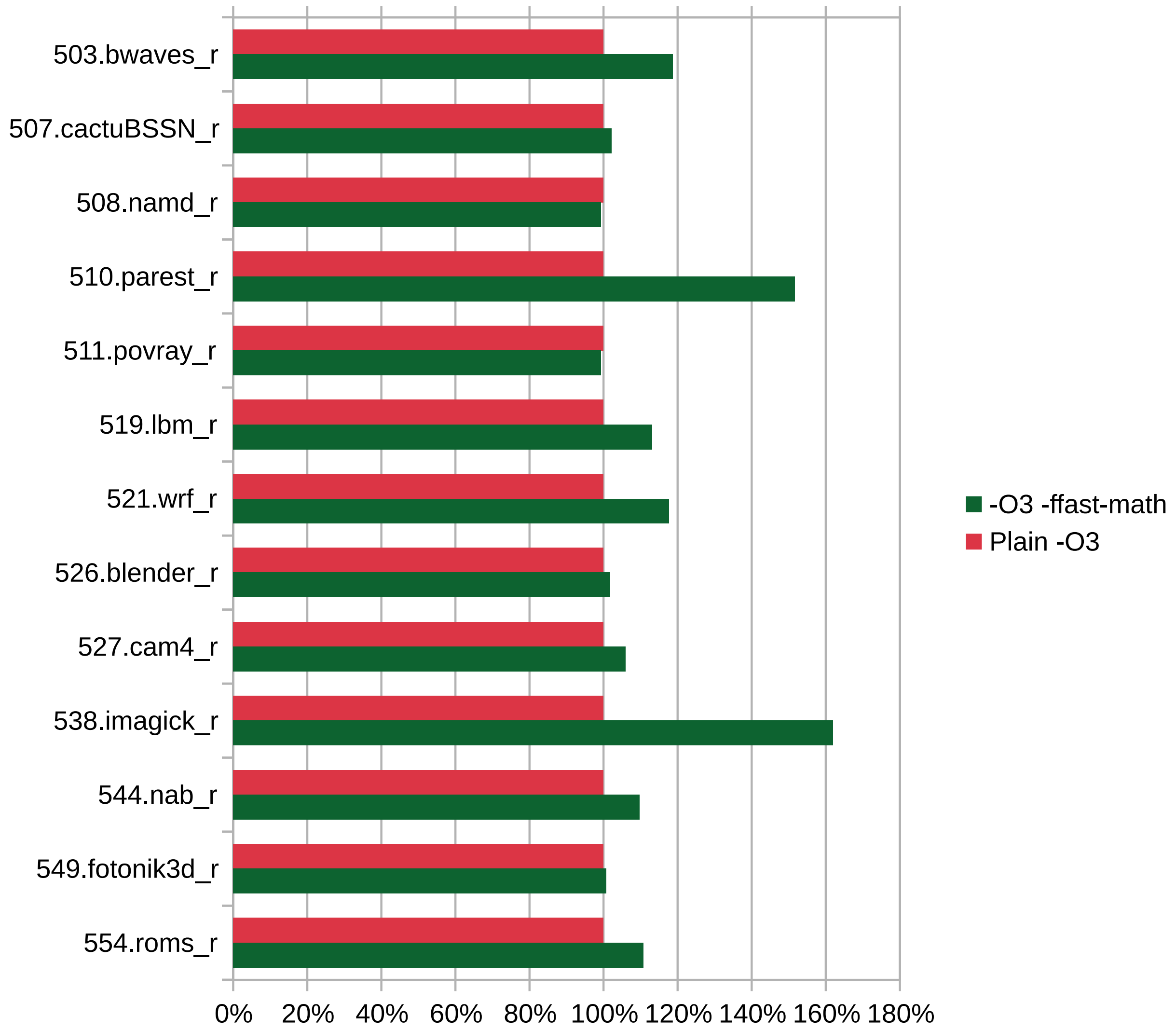

FIGURE 17: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF INDIVIDUAL FLOATING-POINT BENCHMARKS BUILT WITH GCC 14.2 AND -O3 -FLTO -MARCH=NATIVE, WITHOUT AND WITH -FFAST-MATH

7.4 Comparison with other compilers

The toolchain team at SUSE regularly uses the SPEC CPU 2017 suite to compare the optimization capabilities of GCC with other compilers, mainly LLVM/Clang and ICX from Intel. In the final section of this case study, we will discuss how the Development Module compiler compares to its competitors on SUSE Linux Enterprise Server 15 SP6. Before we begin, it is important to note that the comparison was conducted by individuals with significantly more expertise in GCC than in the other compilers, and they are not completely “unbiased”. Also, keep in mind that everything we explained previously about how we carry out the measurements and patch the benchmarks also applies to this section. However, since the results inform our own work, rest assured that we strive for accuracy.

We have built the `clang`, `clang++` and `flang-new` compilers from sources obtained from the official git repository (tag `llvmorg-19.1.4`), used it to compile the SPEC CPU 2017 suite with `-Ofast` and `-march=native` and compared the performance against the suites built with GCC 14.2 with the same options. When using Clang's LTO to compile SPEC, we selected the *full* variant because it is more powerful in terms of optimization capabilities even though it is not suitable for building large projects.

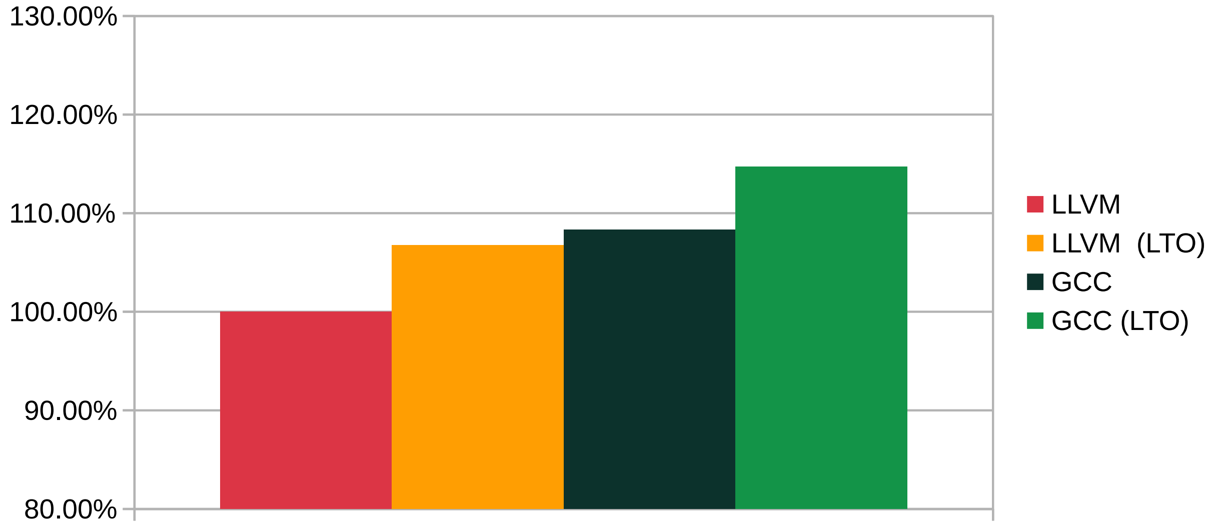

FIGURE 18: OVERALL PERFORMANCE (BIGGER IS BETTER) OF C/C++ INTEGER BENCHMARKS BUILT WITH CLANG 19 AND GCC 14.2

Figure 18 shows that the geometric mean of the whole SPEC INTrate 2017 suite is quite substantially better when the benchmarks are compiled with GCC. To be fair, a disproportionate amount of the difference is because GNU Fortran can optimize `548.exchange2_r` much better than LLVM (see *figure 18*). Given that the LLVM Fortran front-end is relatively new and the optimization opportunities in this particular benchmark are quite specific, the result may not be relevant for many users.

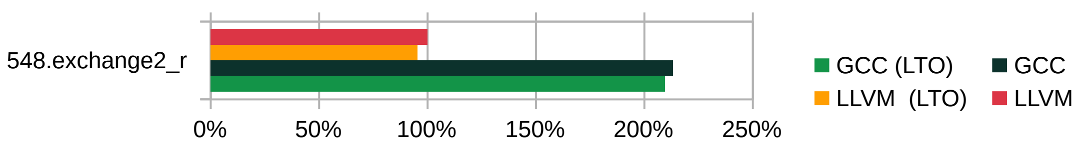

FIGURE 19: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF 548.EXCHANGE2_R BENCHMARKS BUILT WITH CLANG 19 AND GCC 14.2

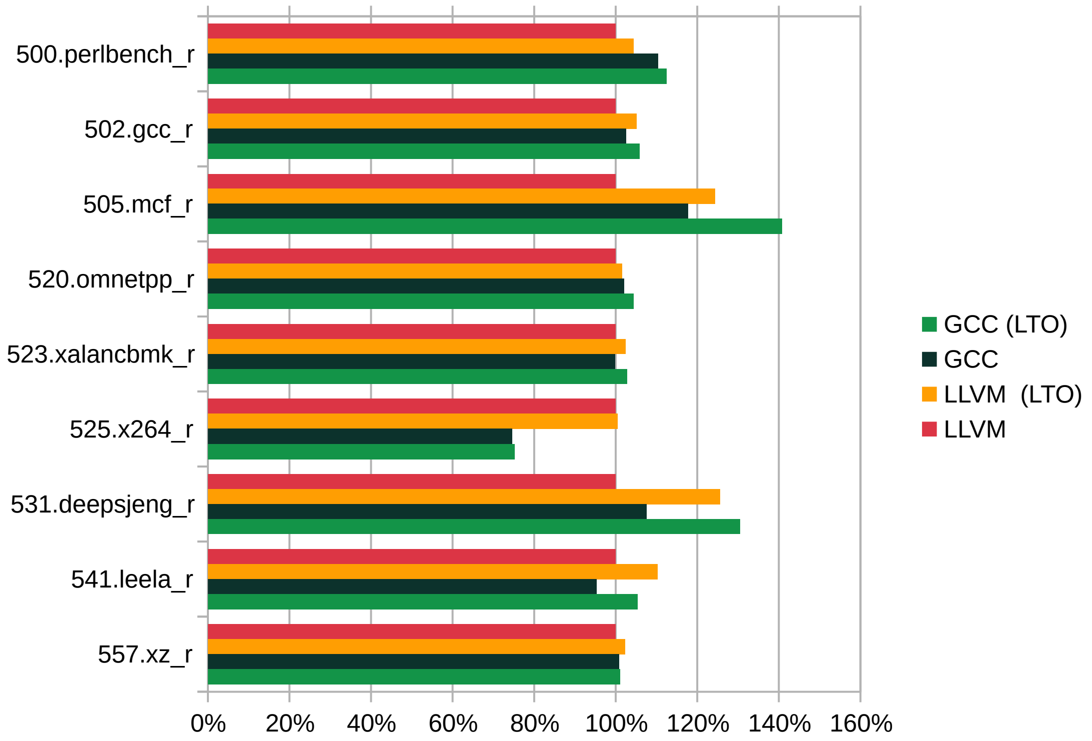

FIGURE 20: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF C/C++ INTEGER BENCHMARKS BUILT WITH CLANG 19 AND GCC 14.2

Figure 20 shows relative rates of integer benchmarks written in C/C++ and the compilers perform fairly similarly there. GCC wins by a significant margin on `505.mcf_r`, `531.deepsjeng_r` and `500.perlbench_r` but clearly loses when compiling `525.x264_r`. This is because the compiler chooses a vectorizing factor that is too large for the important loops in this video encoder. It is possible to mitigate the problem using compiler option `-mprefer-vector-width=128`, with which it is only 9% slower than Clang/LLVM, as you can see in *figure 21*. Another option yielding similar runtime of the benchmark is to use masked vectorized epilogues by passing option `–param vect-partial-vector-usage=1` to the compiler. Note that PGO can substantially help in this case too. The upcoming version, GCC 15, aims to solve the problem without a need for extra options by producing multiple cascading vector epilogues.

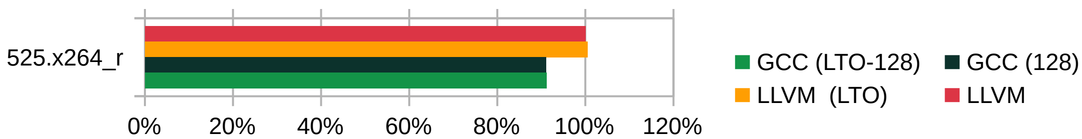

FIGURE 21: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF 525.X264_R BENCHMARK BUILT WITH CLANG 19 AND WITH GCC 14.2 USING -MPREFER-VECTOR-WIDTH=128

The comparison of geometric mean of scores of SPEC FPrate 2017 suite when built with the two compiler suites is depicted in *figure 22*. The floating point benchmark suite includes many more Fortran benchmarks, and it is clear that GCC has an advantage in having a mature optimization pipeline for this language. This is particularly evident when compiling `503.bwaves_r`, `519.lbm_r` and `527.cam4_r` (see *figure 23*). The comparison of performance of individual benchmarks also shows that the performance of `538.imagick_r` is substantially bigger when compiled with GCC 14.2 while Clang/LLVM has an edge when optimizing `508.namd_r` and `544.nab_r`.

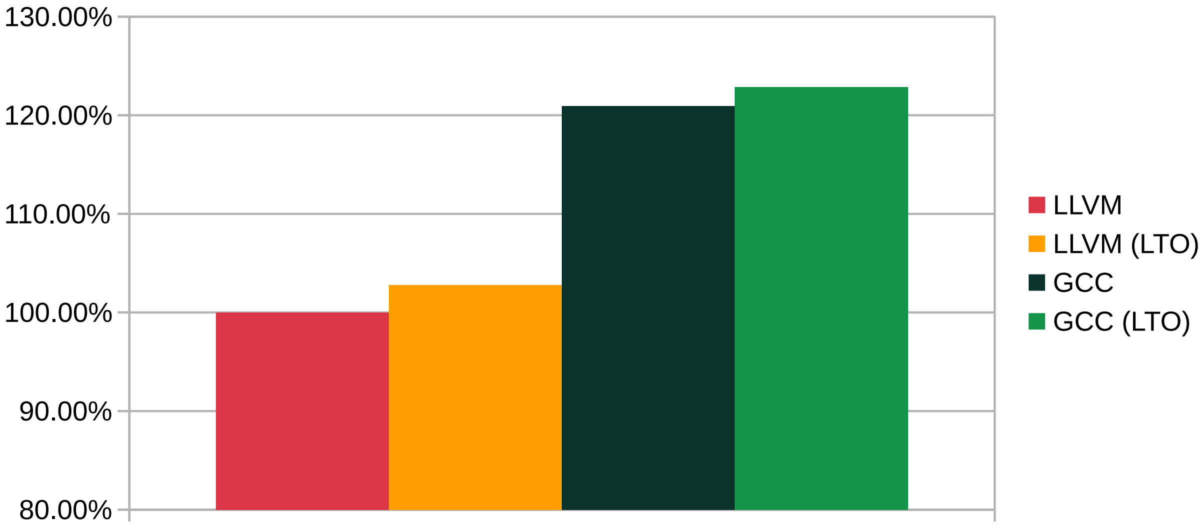

FIGURE 22: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC FPRATE 2017 BUILT WITH CLANG 19 AND GCC 14.2

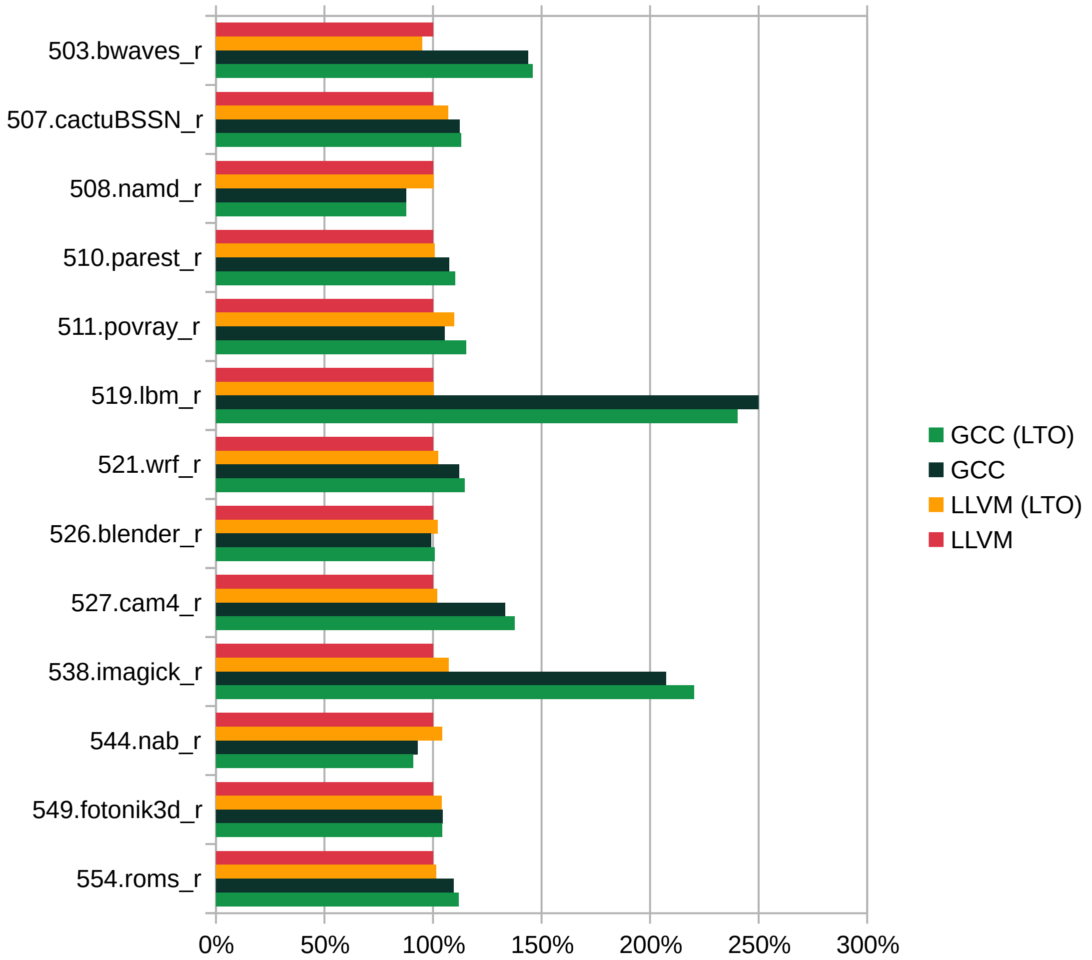

FIGURE 23: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF FLOATING POINT BENCHMARKS BUILT WITH CLANG 19 AND GCC 14.2

Although Intel compilers are not designed for AMD processors, they are well-known for their high-level optimization capabilities, particularly in vectorization. Therefore, we have traditionally included ICC in our comparisons of compilers. Recently, however, Intel decided to discontinue this compiler and redirect its users toward ICX, a new compiler built on top of LLVM. In consequence, we have also shifted our focus to ICX. To keep the amount of data presented in this section manageable, we will focus on comparing only binaries built with LTO `-Ofast` and `-march=native`.

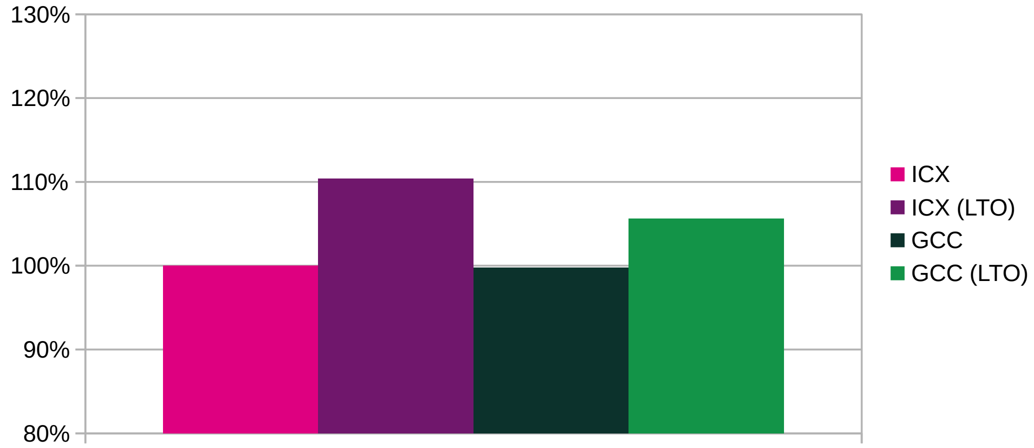

FIGURE 24: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC INTRATE 2017 BUILT WITH ICX 2025.0.1 AND GCC 14.2

Figure 24 shows that the new ICX compiler takes the lead in overall SPEC INTrate assessment. The results of individual benchmarks (see *figure 25*), however, illustrate that the majority of the lead is due to one benchmark, `525.x264_r`, and for the same reasons we outlined when discussing LLVM/Clang results. GCC picks too large vectorizing factor and the mitigation is again using `-mprefer-vector-width=128` which leads to a much narrower gap (see *figure 26*). When looking at the other benchmarks (see *figure 25*), GCC achieves comparable results. In fact, if we excluded benchmark `525.x264_r` from the computations of the geometric means, GCC would achieve a slightly better score than ICX in the LTO case. At this point we want to re-iterate that the next version of GCC aims to solve this problem without a need for extra compiler options.

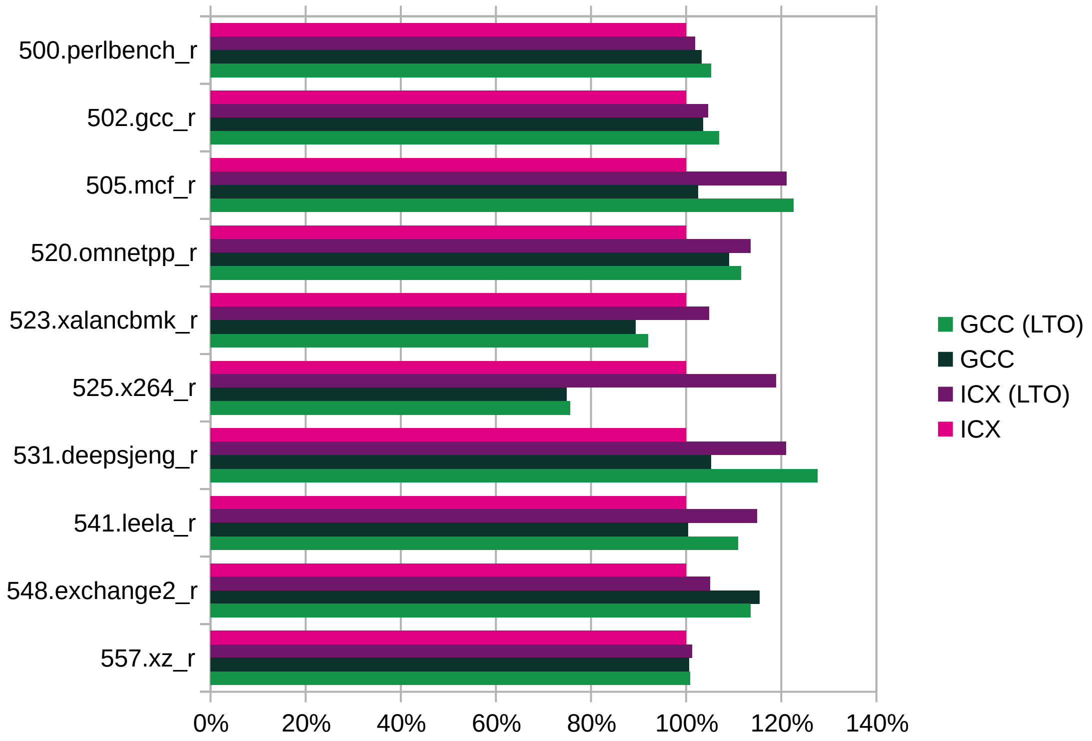

FIGURE 25: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF INDIVIDUAL INTEGER BENCHMARKS BUILT WITH ICX 2025.0.1 AND GCC 14.2

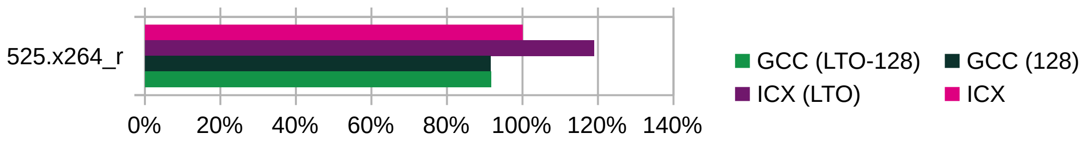

FIGURE 26: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF 525.X264_R BENCHMARK BUILT WITH ICX 2025.0.1 AND WITH GCC 14.2 USING -MPREFER-VECTOR-WIDTH=128

If we look at the geometric means that the two compilers can achieve when they are used to build SPEC FPrate suite, GCC wins by 17% or 19% without and with LTO respectively (see *figure 27*). Even in this case it is important to look at individual results though as the overall picture is more nuanced (see *figure 28*). There are benchmarks where GCC is much better (most prominently `538.imagick_r` and `554.roms_r`) but there are also those where the competition produces considerably faster code (especially `519.lbm_r` and `544.nab_r`). Nevertheless, the conclusion is that GCC manages to perform consistently and competitively against these high-performance compilers.

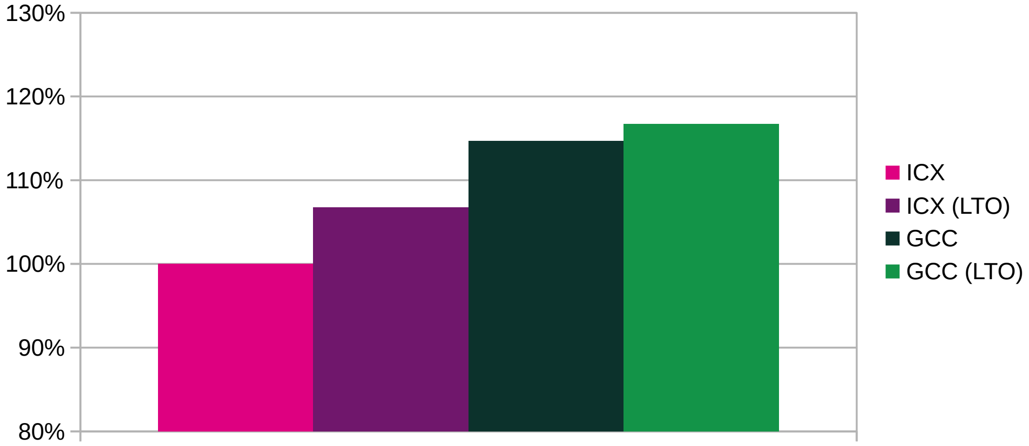

FIGURE 27: OVERALL PERFORMANCE (BIGGER IS BETTER) OF SPEC FPRATE 2017 BUILT WITH ICX 2025.0.1 AND GCC 14.2

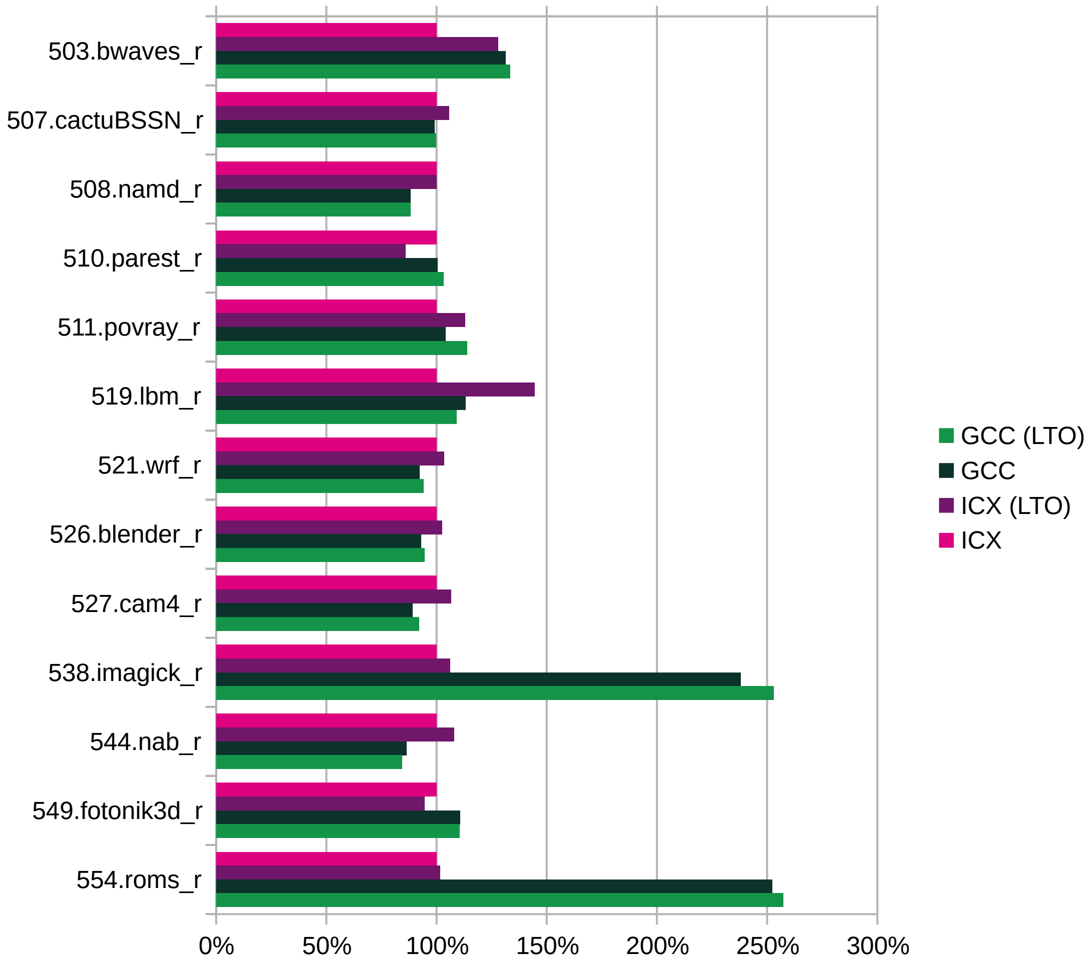

FIGURE 28: RUNTIME PERFORMANCE (BIGGER IS BETTER) OF INDIVIDUAL FLOATING POINT BENCHMARKS BUILT WITH ICX 2025.0.1 AND GCC 14.2

8 Legal notice

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

GNU Free Documentation License

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.