Modern C++ Programming

29. Build Time

Federico Busato

2026-01-06

Table of Contents

Table of Contents

Table of Contents

10 Other Aspects

Compile Time Introduction

Compile Time

Long **build time**, also called *build latency*, is strongly associated with **poor productivity**, discourage refactoring, and experimentation. Studies suggest that even moderate improvements to build latency lead to productivity gain.

> *"While waiting for builds to complete, developers will often work on another project, check email, get a coffee, or go get lunch.*
>
> *The developer's flow was broken: they not only delayed the progress of their task, but they'll likely pay a small penalty associated with the cognitive overhead of task resumption."*

`Developer Productivity for Humans, Part 4: Build Latency, Predictability, and Developer Productivity`, Google 2023

A discontinuous development process also leads to **low engineer satisfaction**.

Delayed development due to slow builds increases the risk of bugs and could result in **customer dissatisfaction**, undermining the company's reputation.

In addition, long build times drive **high infrastructure costs**, especially in large organizations with many developers and frequent CI builds. Such costs could directly affect operational budgets:

- *Energy consumption*: Large projects can consume hours of CPU time.
- *Storage costs*: Large build artifacts and intermediate files generated during prolonged builds could requires terabytes of data.
- *Memory capacity*: Compiling complex code requires gigabytes of memory which can limit the parallelization of the process or even cause compilers to crash if the requirements are not met.

Although C++ is one of the *fastest languages to compile*, **even small changes in large code bases can significantly increase the build time**.

→ `Robot Operating System (ROS) software stack`: 5 commits lead to 59% increase in overall build time. The `Compilation Bloat Issue` resulted in higher CI/CD cost and a significant slow down in development cadence.

Codebase build times tend to increase non-linearly as the project evolves over time. While the effect is negligible for new or small projects, later development phases could be disproportionately affected.

→ `Figma Case`: 10% growth in code size lead to 50% increase in the build time.

Large Real-World Codebases

Chromium Browser	A full build can take from one hour up to 6 hours.
Unreal Engine 4	A full build can take up to 4 hours, while modifying a single file could take 20 minutes.
Microsoft Office	20 minutes or more for a full build.
Windows 10	Up to 16 hours to build.
LLVM	From 30 minutes to 2 hours.
NVIDIA CUTLASS	From 45 minutes to hours.
Linux Kernel	A few minutes (15M lines of code, mostly C), Ubuntu or Fedora configurations could take up to 2 hours.
Large Code Bases	1M+ lines of code, 200+ contributors, 3 yrs+ projects could requires >1 hour to compile

- **Header dependencies**: Including a single header implicitly includes all of its dependencies. As the project grows, the number of dependencies will increase at a faster rate than the number of files. The concept is analogous to *densification* in graph theory*.

- **Template**: Similar to code dependencies, *template instantiations* grow faster than the number of files. Redundant instantiations are difficult to track, especially in large projects.

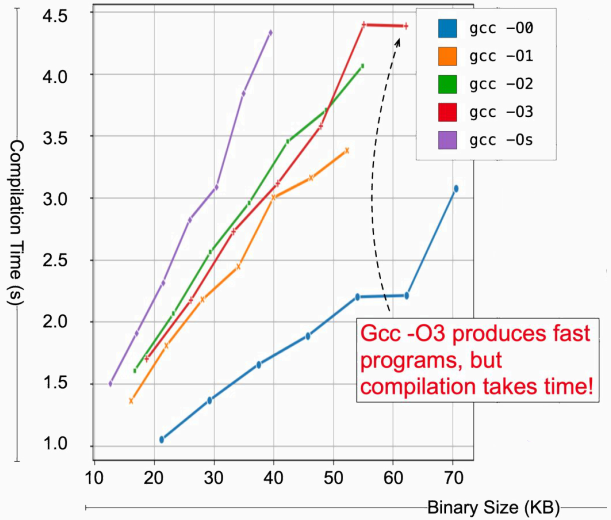
* Graph Evolution: Densification and Shrinking Diameters

- **Monolithic Organization**. Each translation unit is compiled sequentially. As a result, "heavy" non-modular translation units, which carry multiple dependencies and template instantiations, cause a bottleneck in the whole process.

- **Linking**: Linking and Link-Time-Optimization are often the slowest phases due to their sequential nature. In addition, unlike object compilation, the linking phase is not incremental because it is performed in a single step.

Compiler Aspects

Compile Time, Binary Size, Optimization Level



Recent versions of compilers not only focus on implementing new language features, performance, or new warnings, but also on improving compile time.

- A notable example is MSVC, where switching to a recent version (17.14) improved the build time by ∼30%[1].
- In another case, updating the compiler to MSVC to 17.16 leads to 20% improvement [2].
- Clang 22 improves AST representation, making it 8% faster [3].

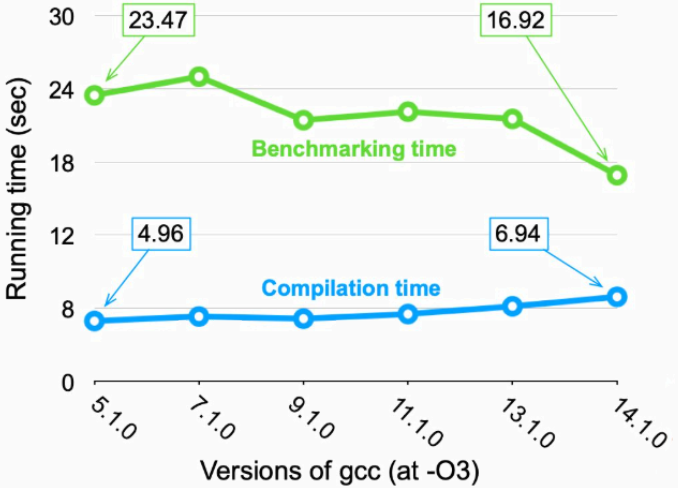
1 Impressive build speedup with new MSVC Visual Studio 2022 version 17.14

2 GDC 2025: How Build Insights Reduced Call of Duty: Modern Warfare II's Build Times by 50%

3 Making the Clang AST Leaner and Faster

Compiler differences: Historically, `clang` showed better compile performance compared to `GCC`. However, the difference has narrowed over time as both compilers have improved. A recent `PostgreSQL` benchmark showed a significant faster build time with `clang`, up to 2x compared to `GCC`. Also, it is important to note that flags selection could lead to different results.

GCC Evolution - Compile-Time vs. Performance



`-O0`, `/O0` *No optimizations* → slowest code, shortest compile time.

`-O<N>`, `/O<N>` *Higher optimization level* → better run-time performance but longer compile time.

`-g`, `-g<N>`, `/DEBUG` *Debug mode* → longer compile time. High debug level worsens compile time.

`-gsplit-dwarf` Split *debugging* information (DWARF) between the standard object file and a separate DWARF object file. This results in smaller object files and, consequently, faster linking times.

`-Wunused`, `-Wunreachable-code`

Raise a warning for unused variables, functions, parameters, return values, type definitions, and unreachable code. Removing unused or unreachable code reduces the code to compile. Included in `-Wall`.

`-DNDEBUG` `/DNDEBUG`

Remove assertions $\rightarrow$ slightly improve compile time by skipping compilation of assertion code .

`-pipe` Use communication pipelines (Linux pipes) rather than temporary files when communicating between different stages of compilation, such as preprocessing and compilation → avoid slow accesses to secondary memory.

`-flto`, `/GL` *Link Time Optimization (LTO), Whole Program Optimization* → The compile time can be increased by up to 10 times.

The compiler generates intermediate representations (IRs) of object files, which the linker uses to optimize the program as a single entity.

As an alternative to using the operating system compiler or downloading a pre-compiled, general binary of the compiler, it is possible to **build the compiler executable directly from the source code**.

This allows users to build a compiler optimized for their CPU architecture and tailored to their workload. An optimized compiler can speed up the compilation process from 15% to 50%.

Build **clang** optimized binary for the current architecture + Link-Time Optimization (LTO). Additionally, enable **lld** linker to speed up the linking process:

```
git clone https://github.com/llvm/llvm-project.git
cmake -G Ninja -S llvm -B build -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_LTO=Thin \
                                -DLLVM_ENABLE_LLD=ON CMAKE_CXX_FLAGS="-O3 -march=native"
cmake --build build
```

Add Profiled Guided Optimization (PGO) and Post-link Binary Layout Optimizer (BOLT). This kind of build is called *multi-stage* or *bootstrap*.

```
cmake -G Ninja -S llvm -B build -C clang/cmake/caches/BOLT-PGO.cmake \
-DPGO_INSTRUMENT_LTO=Thin \
-DBOOTSTRAP_LLVM_ENABLE_LLD=ON \
-DBOOTSTRAP_BOOTSTRAP_LLVM_ENABLE_LLD=ON \
-DBOOTSTRAP_CMAKE_C_FLAGS="-O3 -march=native" \
-DBOOTSTRAP_CMAKE_CXX_FLAGS="-O3 -march=native" \
-DCLANG_PGO_TRAINING_DATA=<lit-style test tree directory> \
-CLANG_PERF_TRAINING_DATA_SOURCE_DIR=<CMake project to build during training>

ninja stage2-clang-bolt
# (1) Builds a stage1 compiler + tools to build an instrumented stage2 compiler.
# (2) Use the instrumented compiler to generate profdata based on the training files.
# (3) Use the stage1 compiler with the stage2 profdata to build a PGO-optimized compiler.
```

Build gcc optimized binary for the current architecture + Link-Time Optimization (LTO):

```
git clone https://github.com/gcc-mirror/gcc.git
./configure --prefix=/opt/gcc-<X.Y> --enable-languages=c,c++ --disable-multilib
make -j20 BOOT_CFLAGS='-flto -O3 -march=native -pipe'
```

Use profile feedback to optimize the compiler itself:

```
./configure --prefix=/opt/gcc-<X.Y> --enable-languages=c,c++ --disable-multilib \
                                    --with-build-config=bootstrap-lto
make -j20 BOOT_CFLAGS='-O3 -march=native -pipe' BUILD_CONFIG=bootstrap-lto \
     profiledbootstrap
```

C++ Standard Version

Newer versions of the C++ standard tend to increase compile time due to the introduction of advanced features, including `constexpr`, lambda expressions, structure binding, etc. The complexity of the language has a direct impact on the performance of the compiler front-end, leading to increased parsing time. The compiler optimizer, operating on the intermediate representation, remains unaffected.

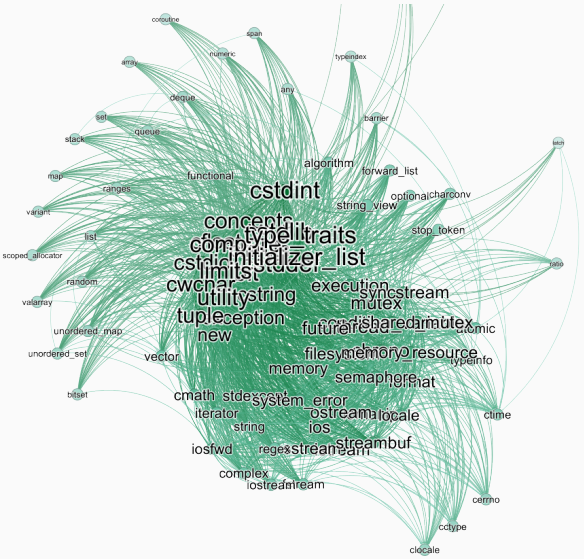
In isolation, C++ standard versions have generally a negligible impact on compile time. On the other hand, *higher C++ versions enable many new functionalities in standard library headers*, which largely dominate the handling of language complexity.

C++ Standard Version and Compile Time

Header	C++03		C++17		C++23	
	Micro sec	LOC	Micro sec	LOC	Micro sec	LOC
`<algorithm>`	48	10,364	67	17,647	130 (2.7x)	32,273 (3.1x)
`<cmath>`	49	6,315	97	21,243	130 (2.7x)	28,298 (4.5x)
`<vector>`	49	7,856	83	20,193	124 (2.5x)	27,972 (3.5x)
`<functional>`	31	20,658	118	36,914	167 (5.3x)	68,984 (3.3x)
`<thread>`	N/A	10,387	86	25,587	513 (5.9x)	33,601 (3.2x)
`<iostream>`	96	5,376	176	25,032	458 (4.7x)	78,717 (14.8x)

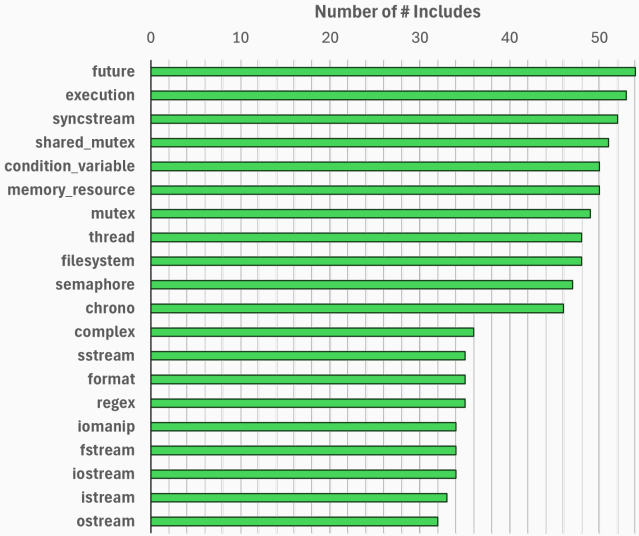
```
g++ -std=c++<VER> -E -x c++ /usr/include/c++/<VER>/<HEADER> | wc -l
```

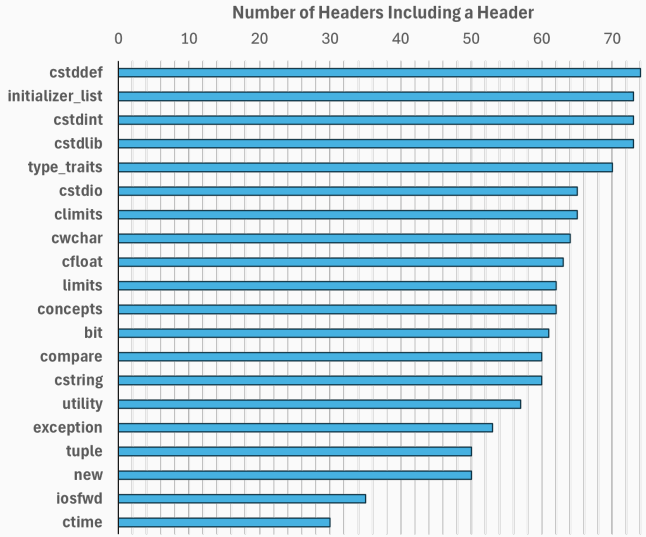
- What happened with compilation times in c++20?
- C++ Compile Health Watchdog



Graph extracted from `STL explorer` ↗, **VS 2022, C++20**:

- 96 headers in total with 1,721 include relationships
- Density: 0.18


Number of # Includes
0
10
20
30
40
50
future
execution
syncstream
shared_mutex
condition_variable
memory_resource
mutex
thread
filesystem
semaphore
chrono
complex
sstream
format
regex
iomanip
fstream
iostream
istream
ostream


Number of Headers Including a Header
0
10
20
30
40
50
60
70
cstddef
initializer_list
cstdint
cstdlib
type_traits
cstdio
climits
cwchar
cfloat
limits
concepts
bit
compare
cstring
utility
exception
tuple
new
iosfwd
ctime

Precompiled Header (PCH)

Compiling a source file involves combining the code of all the header files included directly and, recursively, all their dependencies, until no further inclusions are possible. The complete set of the source file and its headers is referred to as a *translation unit* or *compilation unit*.

Compilation process inefficiency:

- Modifying any code within a translation unit, even a single space, requires recompiling the entire unit from scratch.
- Headers are often included in multiple translation units. Even a small change can cause a large part of the program to be recompiled.
- The prevalent development process involves modifying a single file or a small set of files, even though *most of the code changes infrequently*. One example is the standard library.

Precompiled headers (PCHs) address the problem of lengthy build times in large projects by partially compiling code related to a header into a binary file. Subsequent compilations can then load this file instead of *re-parsing* the headers, which significantly reduces build times.

In more detail, the compiler saves serialized *Abstract Syntax Trees* (ASTs) and supporting data structures in a compressed bitstream to minimize both creation time and initial load time.

Precompiled headers are particularly effective for large header libraries that make extensive use of templates, like the Standard Library, `Eigen`, and `Boost`.

- `MSVC - Precompiled header files`
- `LLVM - Precompiled Header and Modules Internals`
- `GCC - Using Precompiled Headers`

- The *first compilation*, which creates the precompiled header file, takes slightly longer than subsequent compilations.
- Precompiled headers are *recompiled* when:
  - Any of the included files are modified.
  - Modification of compiler flags that alter the generated code, such as optimization flags or macro values.
- *Template Code*: Precompiled headers speed up header parsing containing the template definition, but the cost of template instantiation remains.
- Precompiled headers do not affect the *final code generation* in any way. For instance, they don't alter inlining behavior.
- *Modules* are a generalization of precompiled headers that relax several restrictions placed on precompiled headers.

GCC precompiled header process:

- The user needs to directly compile header files to `.gch` files.
- During normal compilation, `GCC` searches the include directories for the precompiled header.

Additional notes:

- The `-Winvalid-pch` flag warns if a precompiled header is found in the search path but cannot be used.
- `GCC` precompiles template definitions, but not their instantiations.

Clang precompiled header process:

- Similarly to `GCC`, `Clang` allows you to compile header files directly into `.pch` files. Alternatively, the compiler provides the `-emit-pch` flag to explicitly generate them.
- During normal compilation, `Clang` requires the flag `-include-pch <file name>.pch` to load the precompiled headers.

Additional notes:

- `Clang` precompiles template definitions, but not their instantiations by default.
- Starting with Clang 11, the `-fpch-instantiate-templates` flag also enables precompilation of template instantiations in header files. The option can improve compile time by up to 30-40% on template heavy libraries.

Microsoft Visual Studio precompiled header process:

- `Microsoft Visual Studio` requires the option `/Y` to create the precompiler header `.Pch`.
- During normal compilation, `Microsoft Visual Studio` requires the flag `/Yu <file name>.Pch` to use the precompiled headers.

Additional notes:

- If the compiler detects an inconsistency, it issues a warning and identifies the inconsistency where possible.
- `Microsoft Visual Studio` precompiles template definitions, but not their instantiations.

Linker Aspects

The linking phase plays a critical role in the C++ compile time because it is strongly *sequential*. In contrast, object compilation can run concurrently across several threads.

Main linking steps and relation with compile time:

- **Collect object files and libraries**. The linker is also responsible for extracting the necessary objects from static libraries $\rightarrow$ *Negligible impact* in general. However, loading many objects or large ones affects this step.

- **Symbol resolution**. Link references and exported symbols of all object files, ensuring that there are no multiple definitions for the same symbol, nor references without a symbol. Identical symbols allowed in multiple translation units, such as inline functions and templates, must be unified → This step can take *significant time* in large projects involving several thousands of symbols. The problem is worsened by templates, many small functions, function generation through macros, and long symbol names.

- **Relocation**. Compilers generate code as if every object starts at address zero. The linker assigns final memory addresses to all references and data → This phase may take a long time in large codebases.

- **Link Time Optimization (LTO), Whole Program Optimization**. Interprocedural optimizations across translation units. The linker merges the intermediate representations (IRs) of compiled files, performs cross-file inlining, dead code elimination, propagates constants, refines data layouts, and improves code locality by placing related code close together → The most expensive step. The linking time will increase by a factor of 10 to 100.

- **Final binary generation**. Merging all resulting files and debugging information → Debugging metadata could require a significant portion of the linking time. The size of the output plays a minor role.

ThinLTO - Thin Link Time Optimization

ThinLTO (Thin Link Time Optimization) is an advanced form of LTO that is scalable, memory-efficient, and incremental, even for very large codebases.

As with regular LTO, the compiler produces a bitcode for each module with ThinLTO. However, the ThinLTO bitcode is augmented with a compact *summary* of the module. During the linking stage, these summaries are combined, and a global analysis is performed.

ThinLTO is supported on:

- `Clang` 3.9+ with the flag `-flto=thin`
- `GCC` 12+ with the flag `-flto=<num_threads>`
- `LLVM lld` linker 3.9+
- `GNU gold` linker
- `mold` linker

ThinLTO - Advantages

Scalable Operates efficiently on large codebases. Parallel execution on multiple threads. Up to 2-10x faster than full LTO.

Memory-efficient 10x less memory footprint. Full LTO could use up to 100 GB of memory for large projects.

Incremental Only the changed modules that require recompilation or re-optimization. This results in up to 100x faster linking time.

- `Consider ThinLTO vs LTO vs no LTO with respect to compile time and runtime performance`
- `ThinLTO - Towards Always-Enabled LTO - Teresa Johnson, CppCon 2017`

`GNU linker (ld)` Default GNU linker.

- Used until GNU binutils 2.19 and GCC 4.4.0 (2008).

`GNU Gold linker` Replaced the GNU linker since 2008.

- 2-5× faster than the default linker for large C++ builds.
- Enabled with `-fuse-ld=gold` with GCC/Clang.
- Supports only the Linux ELF format.
- Supports LTO and ThinLTO via plugins.
- Deprecated in 2025, binutils 2.44, GCC 15.

`LLVM lld` Linker developed as part of the LLVM framework.

- 2–4x faster than Gold in large-scale projects, especially in multi-thread CPUs.
- Enabled with `-fuse-ld=lld` with GCC/Clang.
- Supports all the main executable formats, operating systems, and architectures.
- Supports LTO and ThinLTO natively.
- Provides better error diagnostics.

`Mold linker` Modern open-source linker.

- 2–5x faster than `LLVM lld` in large projects, especially with many files and in multi-thread CPUs.
- Supports only Linux ELF format.
- Enabled with `-fuse-ld=mold` with GCC/Clang.
- Supports LTO and ThinLTO.
- Requires GCC $\geq$ 12.1 or Clang.

Linker Flags - GCC/Clang Compilers

`-fuse-ld=gold`	Enables GNU Gold linker, Linux ELF only.
`-fuse-ld=lld`	Enables LLVM linker, requires LLVM.
`-fuse-ld=mold`	Enables Mold linker, Linux ELF only, requires a supported Linux distribution or a manual build.
`-flto`	Enables Link Time Optimization, also called Whole Program Optimization, for GCC/Clang and MSVC, respectively.
`-flto=thin` `-flto=<N>`	Enables ThinLTO (Thin Link Time Optimization) for Clang and GCC, respectively. `N` represents the number of threads for the parallelization.

Linker Flags - Microsoft Visual Studio

`/LTCG:incremental` Incremental linking for static libraries.

`/cgthreads:<N>` Number of threads for optimization and code generation.

`/debug:fastlink` Generates a partial Program Database (PDB), containing debugging information. 2-4x faster link times.

- /LTCG (Link-time code generation)
- Speeding up the Incremental Developer Build Scenario
- Improved Linker Fundamentals in Visual Studio 2019

Unity Build

Unity build is a compilation technique where multiple translation units are *textually* merged into one and compiled as a single source file.

Unity build could significantly reduce the compile time for the following reasons:

- Headers are parsed only once, rather than once per translation unit.
- The compiler reuses templates that have already been instantiated and that previously belonged to different translation units. This avoids redundant work.
- Decreases linker work because there are fewer object files and, as consequence, fewer symbols to resolve symbols.
- Expensive Link-Time Optimization can be avoided or reduced for optimized/release builds.

Unity build is successfully used in popular projects such as Unreal Engine, Unity, WebKit, and Mozilla Firefox.

Real-world applications showed that *Unity build* can lower the compile time by:

12x `Comparing C/C++ unity build with regular build on a large codebase`

10x `Faster Compiling: Visual Studio Unity (Jumbo) Builds`

5x `The Little Things: Speeding up C++ compilation`

4x `How I Cut Unity Compile Times by 75%`

Unity build can be applied manually (not suggested), or with the support of tools like `cmake`, `Visual Studio` (native), and `FastBuild`.

`unity.cpp`

```
#include "source1.cpp"
#include "source2.cpp"
...
```

With cmake:

```
cmake_minimum_required(VERSION 3.16) # minimum version required
project(MyProject)

add_executable(prog
    main.cpp
    source1.cpp
    # ...
)
set_target_properties(prog PROPERTIES
    UNITY_BUILD            ON
    UNITY_BUILD_BATCH_SIZE 16) # maximum number of source files that can be combined
```

Additional unity build *positive effects*:

- All merged source files can now be *optimized together*, avoiding Inter-Procedural Optimization (LTO).
- *One Definition Rule (ODR) violations*, such as two `inline` functions with the same name, now trigger a compile error.

Unity build compilation time *drawbacks*:

- *Incremental recompilation slow down*. Unity build greatly benefits clean, full builds, while it could increase compilation time compared to modifications to one or few translation units.
- Unity build can translate into CPU under utilization due to *lack of build parallelism*. For example, one of the most common problem is a wrong value of `cmake` `UNITY_BUILD_BATCH_SIZE` option.

Unity build negative side effects:

- *Valid code could not compile*.
  - Collision of symbols with internal linkage and identical name, for example `static` or within an *anonymous* namespace variables.
  - Macro collisions and propagation.
- Bigger translation units translates into larger amount of memory usage during compilation. This could cause compilation process termination or system crash.

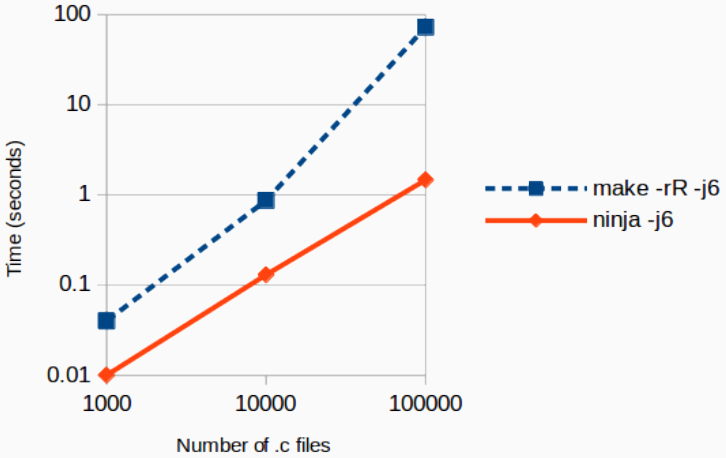
Tools for Reducing Build Time

The Ninja build system is a drop-in replacement for `Make` and `MSBuild` (`Visual Studio`) focused on speed and scalability.

- While `ninja` has comparable time to `Make` for initial builds, it can be up to 100x faster for incremental or no-op builds in large projects.
- Compared to `Make`, `ninja` has a less rich and intuitive syntax. On the other hand, it allows minimizing parsing time and enables fast dependency scanning.
- `ninja` dependency files are intended to be produced by higher-level generators, such as `CMake`, rather than manually as in `Make`.
- `ninja` is actively used in projects like Chrome, Android, LLVM, and Swift.

Key features:

- Fast, especially with incremental or no-op builds.
- Portable. Supported on Linux, Windows, Mac OS X, and FreeBSD.
- Standalone and minimal (~300 KB).
- Automatic parallelization, no need of `-j` flag.
- Queries, such as target dependencies or dependency graph, to analyze and understand build bottlenecks.


100
10
1
0.1
0.01
Time (seconds)
1000
10000
100000
Number of .c files
make -rR -j6
ninja -j6

Enable ninja with CMake:

- Explicitly set the generator in CMake:

```
cmake -G Ninja ..

ninja
# or 'cmake --build .'
```

- Set the `CMAKE_GENERATOR` environment variable:

```
export CMAKE_GENERATOR=Ninja
cmake ..

# or CMAKE_GENERATOR=Ninja cmake ..
```

`ninja -t <tool>`, where `<tool>` is

`targets`	List of targets.
`query <target>`	Inputs and outputs of a given target.
`inputs <targets>`	List of inputs.
`deps [<targets>]`	List of internal and external dependencies.
`commands <targets>`	Print a list of commands to build the target(s).
`clean`	Remove built files.

Browse the dependency graph in a web browser.

```
ninja -t browse --port=8000 --no-browser mytarget
```

lib/libsfml-network-s.a

target is built using rule CXX_STATIC_LIBRARY_LINKER__sfml-network_Release of

lib/libsfml-system-s.a (order-only)
src/SFML/Network/CMakeFiles/sfml-network.dir/Ftp.cpp.o
src/SFML/Network/CMakeFiles/sfml-network.dir/Http.cpp.o
src/SFML/Network/CMakeFiles/sfml-network.dir/IpAddress.cpp.o
src/SFML/Network/CMakeFiles/sfml-network.dir/Packet.cpp.o
src/SFML/Network/CMakeFiles/sfml-network.dir/Socket.cpp.o

dependent edges build:

libsfml-network-s.a
sfml-network

Generate the dependency graph in dot format.

```
ninja -t graph [mytarget] | dot -Tpng -ograph.png
```

Compiler Cache

A compiler cache tool is designed to speed up the process of recompilation by storing the results of previous compilations for reuse when the same compilation occurs again.

Common usages:

- Clean build of a project to avoid stale artifacts, caches, or misdetected dependencies.
- Build a project across different branches or directories.
- Continuous Integration (CI) shared across multiple developers.
- Validate software build status for each commit.

Quote from my team:

> *"... just brought down the time for our fully cached MSVC builds from 1h20m to 15m saving us like $30k/mo in AWS costs."*

How a Compiler Cache Tool Works

1 *Detect if the compilation inputs has been already processed previously.*
This includes the code of a translation unit, compiler information (name, location, size, etc.), compilation flags, and the build directory for debug builds. The information to uniquely identify compilation inputs are hashed by using a cryptographic hash algorithm.

2a *If a match is found*, the compiler cache reuses the result of the compilation without invoking the compiler.

2b *If there is no match*, the compiler cache falls back to a normal compiler call and hashes the compilation inputs. Lookups of previous results could also fail if the modification time of an included file changes, or if the source code depends on time information, namely the `__TIME__` macro.

Compiler Cache - CCache

CCache is one of the most widely used compiler caching software tools. It is commonly used as a wrapper for compiler calls, which makes it easy to use and integrate with build systems such as `Make` and `CMake`.

`CCache` supports several languages, operating systems, compilers, and x86/Arm64 cpu architectures. It relies on `BLAKE3` hash algorithm, `XXH3` for file checksum, and `ZStd` for file compression.

Usage:

```
ccache g++ file.cpp <other flags>          # single translation unit
CXX='ccache clang++' make                  # makefile
export CMAKE_CXX_COMPILER_LAUNCHER=ccache # cmake environment variable
```

Compiler Cache - SCCache

SCCache is a widely used compiler caching software tool focused on remote/cloud backends. SCCache is particularly useful for enabling cache sharing across machines and CI systems.

While SCCache supports local disk cache, it is significantly slower than CCache.

Usage:

```
sccache g++ file.cpp <other flags>          # single translation unit
CXX='sccache clang++' make                  # makefile
export CMAKE_CXX_COMPILER_LAUNCHER=sccache # cmake environment variable
```

See Storage Options for remote storage configuration.

Large C++ codebases can benefit from **distributing builds** across machines on a network. Tools for distributing builds are particularly useful for scaling up the throughput and latency of Continuous Integration (CI) for projects involving several developers.

- `distcc` is a popular tool to distributes compilation jobs. Linking and non-compile steps remain local.
- `icecream` distributes compilation with a central scheduler and ships toolchains to workers to keep environments aligned.
- `sccache-dist` adds distributed compilation to `sccache` caching and scheduling remote compilations.

- `incredibuild` is a commercial distributed build tool that coordinates the compilation CPUs, LAN, and cloud. The tool combines task distribution with a compilation cache.
- `FASTbuild` is a high performance, open-source build system, supporting highly scalable parallel compilation, unity build, caching, network distribution.

Distributing builds are especially effective when used in combination with local compilation cache tools.

- Initially, the tool searches for cached binary results on the local system.
- If a local binary lookup fails, the tool sends the job to the available clients.
- If a remote client returns a hit, the cached binary is sent back to the server.
- Otherwise, the job is assigned to a client that compiles the code and sends the results back to the server.
- The remote binary then populates the local cache.

A **RAM disk**, also known as an *in-memory file system*, can reduce C/C++ compile times by eliminating the need for slow access to *secondary storage*, such as a hard disk or network storage.

Builds that rely heavily on I/O operations involving many small files, or large objects, or big temporary files, can benefit from a RAM disk, especially when using a magnetic-mechanical data storage device (HDD). The advantage is less pronounced with a solid-state drive (SSD).

Note: Large projects, especially when LTO is enabled, can generate several gigabytes of temporary files. If the size of the RAM disk is insufficient, the operating system uses the slow *swap space* on secondary storage. This also applies to processes not involved in the compilation, slowing down the entire system.

A practical way to use a RAM disk is to create a fixed-size `tmpfs` and configure the build system to use it as a build directory [1].

```
mount -t tmpfs -o size=8G tmpfs <path_to_directory>
```

The build time for many projects is spent on CPU-intensive compilation, with I/O representing only a small part of the process. Also, modern operating systems often cache the intermediate objects.

The typical build time improvement is **20–30**% in the best cases [2]. Combining a RAM disk with compiler cache can provide up to **4x** improvements [3].

[1] Chromium - Tips for improving build speed on Linux
[2] Using RAMDisk to Speed Build Times
[3] Massive speedup using ccache in disk backed RAMFS

Include-What-You-Use (IWYU) is a Clang-based static analysis tool that checks which headers should be added or removed. Its goals are to remove unnecessary header inclusion and suggest headers based on symbols usage.

```cpp
#include <vector>
int main() {
    size_t x = 10;
}
```

```
# OUTPUT
main.cpp should add these lines:
#include <stddef.h>  // for size_t

main.cpp should remove these lines:
- #include <vector>  // lines 1-1

The full include-list for main.cpp:
#include <stddef.h>  // for size_t
```

```
sudo apt install libzstd-dev # requires zstd
git clone https://github.com/include-what-you-use/include-what-you-use.git
cd include-what-you-use
git checkout clang_21
mkdir build && cd build
cmake -G "Unix Makefiles" -DCMAKE_PREFIX_PATH=<path_to_llvm_root_dir> ..
make -j20
```

The tricky part is that `include-what-you-use` requires a non-IR version of the LLVM libraries.

```
# if the linux distribution has a supported llvm package
wget https://apt.llvm.org/llvm.sh
chmod +x llvm.sh
sudo ./llvm.sh 21 # version_number
```

Otherwise, you will get an error similar to:

```
lib/libLLVMOption.a: error adding symbols: file format not recognized
```

The solution involves building LLVM from sources:

```
git clone https://github.com/llvm/llvm-project.git
cd llvm-project # referred as <llvm-project_dir> in cmake
git checkout llvmorg-21.1.3
cmake -S llvm -B build -G Ninja \
    -DLLVM_ENABLE_PROJECTS="clang;lld" \
    -DCMAKE_BUILD_TYPE=Release \
    -DLLVM_TARGETS_TO_BUILD="X86" \
    -DLLVM_ENABLE_LTO=OFF
```

Then, provides the LLVM cmake directory to `include-what-you-use`:

```
cmake -G "Unix Makefiles" -DLLVM_DIR=<llvm-project_dir>/build/lib/cmake/llvm/ \
                          -DClang_DIR=<llvm-project_dir>/build/lib/cmake/clang/ ..
```

Function Inlining

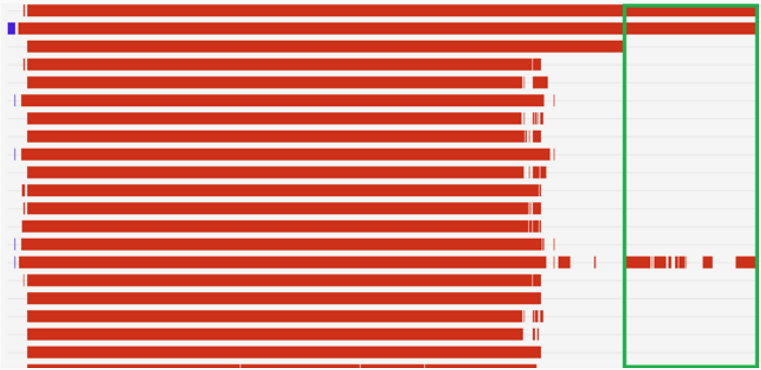
Function Inlining

Function inlining causes the compiler to substitute the function body into each call site, leading to increased code generation and optimization work during compilation.

- Function inlining could consume a significant portion of total build time for large projects.
- Function inlining is one of the first optimization introduced by the compiler, namely `-O1`. There are several factors that affect function inlining, see `Optimization II` lecture.
- Function inlining can be also manually controlled for each function with the attributes `[[gnu::always_inline]]`, `[[gnu::noinline]]`, `[[msvc::forceinline]]`, `[[msvc::noinline]]`

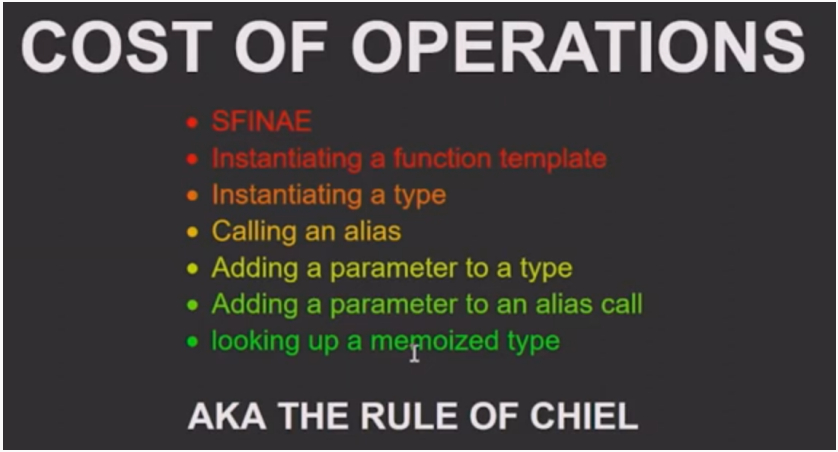
From `GDC 2025: How Build Insights Reduced Call of Duty: Modern Warfare II's Build Times by 50%:`

> *"Two source files were the long pole for compilation time. The entire build pipeline reduced to only two cores for a significant amount of time while it waited for these files to finish building...This represents 12% of the total build time...*
> *One function had **13,700** force-inlined calls, due to a combinatoric explosion in call counts and resulted in **70 seconds** of compile time."*



"This issue often arises in patterns where an inlined function is repeatedly called within constructs like `switch` *statements..*
...restructuring the code to defer the call until after the `switch` *...we were able to reduce it* ***from 70 seconds to just 1.5 seconds****"*

Template



Type Based Template Metaprogramming is Not Dead, *Odin Holmes*, C++Now 2017

(1) SFINAE

```
template<typename T>
std::enable_if_t<std::is_integral_v<T>>
foo(T) {}
```

(2) Instantiating a function template

```
template<typename T>
T my_function(T x) {
    return x;
}
int main() {
    return f(1); // instantiates my_function<int>
}
```

(3) Instantiating a type (class template)

```
template<typename T>
struct MyClass {
   T value;
};
int main() {
   MyClass<int> b{42}; // instantiates MyClass<int>
}
```

(4) Calling an alias

```
template<typename T>
using ptr_t = T*; // alias template

int main() {
   ptr_t<int> p = nullptr;  // "calls" ptr_t with T = int
}
```

(5) Adding a parameter to a type

```
template<typename... Ts>
struct type_list {};

template<typename... Ts>
struct type_list_add_void {
    using type = type_list<Ts..., void>;
};

int main() {
    using List1 = type_list<int, double>;
    // struct instantiation with a new paramter
    using List2 = type_list_add_void<int, double>::type;
}
```

(6) Adding a parameter to an alias call

```
template<typename... Ts>
struct type_list {};

template<typename... Ts>
using type_list_alias = type_list<Ts...>;

template<typename... Ts>
using type_list_alias_add_void = type_list_alias<Ts..., void>;

int main() {
    using List1 = type_list_alias<int, double>;
    // alias call with an additional parameter
    using List2 = type_list_alias_add_void<int, double>;
}
```

(7) Looking up a memoized type

```cpp
#include <vector>

using memoized_type = std::vector<int>;

int main() {
    memoized_type v1; // first instantiation -> expensive
    memoized_type v2; // no new template instantiation, only a lookup -> cheap
}
```

`extern template` allows to suppress *implicit* instantiation of specific template class or function in a translation unit.

Template entities are commonly defined in headers and included in multiple translation units. Every time a template is instantiated, the compiler needs to independently process it for each translation unit, generating redundant work.

Explicit or *implicit* instantiation can be used to select what translation unit processes and stores the template instantation.

- Explicit, `template class MyClass<int>;`
- Implicit, `MyClass<int> m;`

The technique is hard to maintain for all instantions but it is effective when targets specific and expensive template functions or classes.

header.hpp

```cpp
template <typename T>
struct A { /*heavy code*/ };

template <typename T>
void f(T) { /*heavy code*/ };
```

main.cpp

```cpp
#include "header.hpp"
// skip instantiations
extern template class A<int>;
extern template void f<double>();

int main() {
    A<int> a;
    f(3.4);
}
```

source.cpp

```cpp
#include "header.hpp"

void g() {
   f(5.0);
   A<int> a2;
}
// or alternatively:
// template void f<double>();
// template class A<int>;
```

Template structure + `static` data member:

```
template <typename T>
struct A {
    static constexpr int value = sizeof(T) * sizeof(T);
};
```

`constexpr` variable:

```
template <typename T>
constexpr int value_v = sizeof(T) * sizeof(T);
```

Template `constexpr` variables reduce the compile time compared to template class + a `static` data member by:

- Instantiating only the variable specialization itself.
- Reducing the amount of source the compiler needs to parse and process.
- Lowering the complexity of the abstract syntax tree (AST).
- Decreasing the number of symbols generated.
- Simplifying name lookup complexity by removing a level of template indirection. `value_v<int>` avoids the extra indirection of `A<int>::value`.

Tag dispatching is a technique where different function declarations are selected at compile time by overloading on a "tag" parameter. While it avoids SFINAE in signatures, it introduces expensive overload resolution.

```
template <typename T>
T my_abs(std::true_type, T data) {
    return data;
}
template <typename T>
T my_abs(std::false_type, T data) {
    return data < 0 ? -data : data;
}
template <typename T>
T my_abs(T data) {
    return my_abs(std::is_unsigned_v<T>, data);
}
```

Tag dispatching affects the compile time by introduction new functions in the overload set. The overload resolution phase is also repeated for each instantiated "tag".

A better solution is relying on `if constexpr` whenever possible

```
template <typename T>
T my_abs(T data) {
    if constexpr (std::is_unsigned_v<T>) {
        return data;
    }
    else {
        return data < 0 ? -data : data;
    }
}
```

Fold expressions can reduce compile time by avoiding recursive template instantiations and SFINAE.

Problem: compute the total size in bytes of a list of types at compile time.

Supposing a list of 10 types, a *recursive template implementation* requires the instantiation of 11 template structures.

```
template <typename T, typename... TArgs>
struct SizeOfTypes {
    static constexpr auto value = sizeof(T) + SizeOfTypes<TArgs...>::value;
};
template <>
struct SizeOfType<> {
    static constexpr size_t value = 0;
};
```

While an implementation based on *fold expression* only requires one template instantiation.

```
template<typename... TArgs>
struct SizeOfTypes {
    static constexpr auto value = sizeof(TArgs) + ...;
};
```

Concepts

C++20 **concepts** constrain the set of arguments that are accepted as template parameters. They are a superior alternative to SFINAE.

C++20 concepts can reduce compile times when they replace complex SFINAE or tag-dispatch patterns because they avoid template parsing and instantiatiation.

Compile time results related to C++20 concepts usage are mixed. A user reported **20%** faster build with `range v3` library [1], while a stress test comparing `requires` and `enable_if` showed a compile time increase in several cases [2].

[1] `Do C++20 concepts change compile-time, positively or not?`

[2] `A compile-time benchmark for enable_if and requires`

auto

C++11 `auto` keyword, an related C++20 extension to parameters, is mostly a syntactic simplification and has no noticeable compile-time advantage over templates. Under the hood, the compiler still instantiates template code.

`auto` might provide two indirect advantages:

- Less code to parse compared for complex type traits.
- Avoid return type SFINAE when combined with `if constexpr`.

```
template<typename T>
std::enable_if_t<cond1, R1> f(T) {
    /*code1*/
}
template<typename T>
std::enable_if_t<cond2, R2> f(T) {
    /*code2*/
}
```

```
template<typename T>
auto f(T) {
    if constexpr (cond1) {
        return R1;
    }
    else if constexpr (cond2) {
        return R2;
    }
```

`using` Type Aliasing

Type aliasing is merely a naming mechanism and does not by itself reduce template instantiation work.

On the other hand, it is strongly suggested to use *type aliasing* instead of creating new types because it increases the likelihood of reusing a previously instantiated template.

to

```
struct MyNewType : MyComplexType<int> {};
```

```
using MyNewType = MyComplexType<int>;
```

Exception: C++20 Template automatic deduction (CTAD) with type aliasing works only with trivial template argument forwarding. Any transformation on template arguments prevents CTAD.

Other Aspects

Header inclusion is one of the major causes of long compile time. C++20 **modules** are a modern replacement to the traditional `#include` system. *Modules* can be considered a standardized and enhanced form of *Precompiled Header* (`PCH`).

The compiler parses *module*, builds *binary module interface* (`BMI`) files once, and reuse them across translation units, avoiding repeated work.

Key benefits:

- Parsing binary files is more efficient than parsing text files.
- Clear boundaries help to better isolate the code.
- Speed up incremental rebuilds.

- `Boost Asio`-based server + *module*-based `libc++`: **45%** reduction [1].
- `Alibaba Cloud Hologres`, a real-time data warehousing service: **42%** reduction [2, 3].
- A study on scientific software packages showed **10%** reduction for large projects and **22%** on smaller ones [4].

[1] C++20 modules and Boost: deep dive.

[2] 42% Boost in Compilation Efficiency! A Practical Analysis of C++ Modules

[3] Compilation Speedup Using C++ Modules: A Case Study - Chuanqi Xu - CppCon 2022

[4] Experience converting a large mathematical software package written in C++ to C++20 modules

- `clang documentation` highlights that higher optimization level could shrink the advantages of modules. This is because Inter-Procedural Optimization/ Link-Time Optimization require to reprocess the code in the importee units.
- Only 62 over 2,443 popular C++ projects (2.5%) actually implement *modules*, see `Are We Modules Yet?`
- Require recent tools: `gcc` 14+, `clang` 16+, `Visual Studio` 17.4+, `CMake` 3.28+, recent version of `ninja`.
- `Non-trivial #include to modules porting work`: macro reorganization, track “internal" vs “public" headers rigorously, and take care of re-export imported module.

For each function call, the compiler must consider all possible function declarations across the overload set to choose the best match. The main steps are *name lookup* (lightweight) and *overload resolution* (expensive).

The overload resolution process consists in three main phases:

(1) Generates a set of *candidate functions*, namely function overloads.

(2) Filters them into *viable functions*, considering number of parameters, their types, and potential conversions.

(3) Selects the best *viable function* by evaluation exact parameter type match, conversion, and promotion.

Function templates requires additional work because the compiler needs to consider template argument deduction.

The compiler work is proportional to the number of function calls and function overloads.

Practical considerations:

- Filtering based on number of parameters is cheaper than other rules.
- Function template overloads are more expensive to evaluate than standard functions.
- C++20 template constrains help to reduce the number of *viable candidates*.
- Namespace also helps to reduce the number of *viable candidates*, while `using namespace` has the opposite effect, forcing the compiler to consider more overloads.

Other Code Aspects

- **Unused headers** can significantly affect build time but removing them is not a trivial process. It can be done manually (not recommended), or with tools:
  - `clangd` header fixes within the IDE, sometimes not precise.
  - `include-what-you-use` precise but hard to configure. See section 5.

- Defining function bodies **out-of-line** (`.cpp` file) improves build time because implementation changes involves the recompilation of a single translation unit instead of many. See also the PIMPL idiom at section 4.

- `std::function` 37 percent of HyperRogue's compilation time is due to `std::function`.

Pointer Implementation (PIMPL)

Pointer IMPLementation (PIMPL) reduces build time by avoiding header dependencies. The idiom separates class interface and implementation details, namely private data members and methods.

```cpp
#pragma once
#include <memory>

class MyClass {
public:
    MyClass();
    ~MyClass();
    void set(int x);
    int  get() const;
private:
    struct Impl;               // forward declaration, implemented in a .cpp file
    std::unique_ptr<Impl> p_; // opaque pointer to implementation
};
```

Include Guard vs. `#pragma once`

Include guard based on macro definition requires the compiler to read a header multiple times. The header is parsed once, while the following reads exclude the code protected by the guard macro.

With `#pragma once`, the compiler marks the file as "already included" and, on subsequent inclusions, skips reading the file entirely, reducing I/O.

It is important to note that modern compilers, such as `Clang`, `GCC`, and `MSVC`, implement the `#pragma once` optimization even for include guard macro, avoid rereading the same header multiple times.

Static vs. Dynamic Linking

Dynamic linking requires libraries built as independent shared objects. The main executable links against them at run-time without embedding all their code. Dynamic linking doesn't practically affect the linking time.

On the contrary, **static linking** is potentially *expensive* because it involves loading library files (often very large), symbol resolution, section merging, relocation, and potentially *Link-Time Optimization*.

Comments and Formatting

Compile-time impact of comments and empty lines (include only):

Header	Lines	Code	Comments	Empty Lines	Time (msec)
Original header	26,007	3,143	21,807	1,057	42.65
No Comments	26,007	3,143	0	22,864	41.29
No Comments and no Empty Lines	26,007	3,143	0	0	**6.06**

Tested with `gcc 14` and `clang 20`

Removing comments had negligible effect, while eliminating empty lines dramatically reduced compile time.

Different coding style and formatting could also affect compile time. For example, placing opening brackets `{` on the same line or a new one.

External Factors

In addition to examining the building process itself, it is important to consider **external factors**, such as how the operating system manages processes, as these can significantly impact build time:

- *Terminate* memory, computing, I/O intensive, programs and services. Two examples are `Windows file search indexing` and antivirus.
- Set *CPU scaling governor to performance*, see the lecture "`Optimization III: Non-Coding Optimizations and Benchmarking`".
- *Do not use remote file system* as `build` or temporary directories. Network latency can severely slow down the build process.
- Increase *build process priority*.

Tools to Analyze Build Time

`Clang` mainly provides three options to analyze the build progress:

`-fproc-stat-report` Print used memory and execution time of each compilation step (also available in GCC).

`-ftime-report` Print time spent during compilation phases.

`-ftime-trace` Generate Clang timing information in the Chrome Trace Event format (JSON).

`-ftime-report`:

```
phase last asm           :   0.08 (  2%)   0.00 (  0%)   0.09 (  1%)    2921 kB (  1%)
|name lookup             :   0.18 (  5%)   0.10 (  7%)   0.27 (  4%)    4820 kB (  2%)
|overload resolution     :   0.13 (  4%)   0.12 (  9%)   0.31 (  5%)   26374 kB ( 10%)
dump files               :   0.08 (  2%)   0.01 (  1%)   0.04 (  1%)       0 kB (  0%)
callgraph construction   :   0.08 (  2%)   0.02 (  1%)   0.11 (  2%)   12160 kB (  4%)
callgraph optimization   :   0.02 (  1%)   0.02 (  1%)   0.04 (  1%)      87 kB (  0%)
ipa function summary     :   0.00 (  0%)   0.01 (  1%)   0.02 (  0%)     808 kB (  0%)
ipa cp                   :   0.01 (  0%)   0.01 (  1%)   0.01 (  0%)     564 kB (  0%)
ipa inlining heuristics  :   0.01 (  0%)   0.00 (  0%)   0.02 (  0%)     807 kB (  0%)
ipa function splitting   :   0.01 (  0%)   0.00 (  0%)   0.02 (  0%)     233 kB (  0%)
ipa pure const           :   0.02 (  1%)   0.00 (  0%)   0.01 (  0%)      22 kB (  0%)
ipa icf                  :   0.01 (  0%)   0.00 (  0%)   0.02 (  0%)       0 kB (  0%)
ipa SRA                  :   0.07 (  2%)   0.02 (  1%)   0.10 (  2%)    6231 kB (  2%)
```

`-fproc-stat-report` :

example:

```
clang -fproc-stat-report=report.csv main.cpp
```

output:

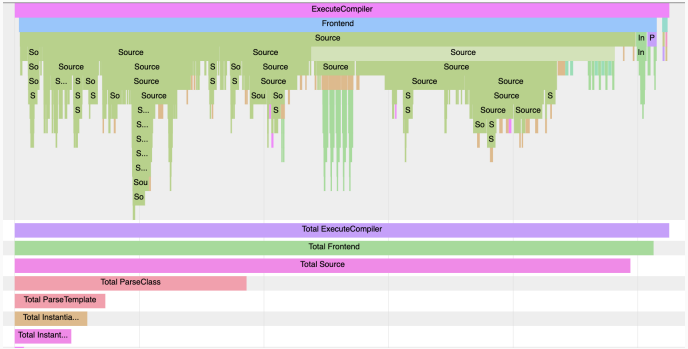
```
# total execution time in microseconds, user mode, peak memory usage in Kb
clang-11,"/tmp/main-123456.o",92000,84000,87536
ld,"a.out",900,8000,53568
```

`-ftime-trace` :

example:

```
clang -ftime-trace=report.json main.cpp
```

output:

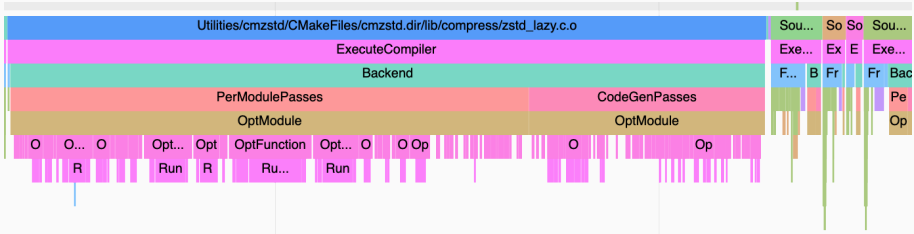


ninjatracing is a tool to convert `.ninja_log` files to chrome's `about:tracing` format.

```
cmake -G Ninja ..  # use the Ninja build system
ninja              # generate .ninja\_log
ninjatracing .ninja_log > trace.json
#open about:tracing in a chrome-based browser
```

ninjatracing can combine the output of ninja and clang -ftime-trace with the option -e.

```
ninjatracing -e .ninja_log > trace.json
```



Templight

Templight is a Clang-based tool to profile the time and memory consumption of template instantiations.

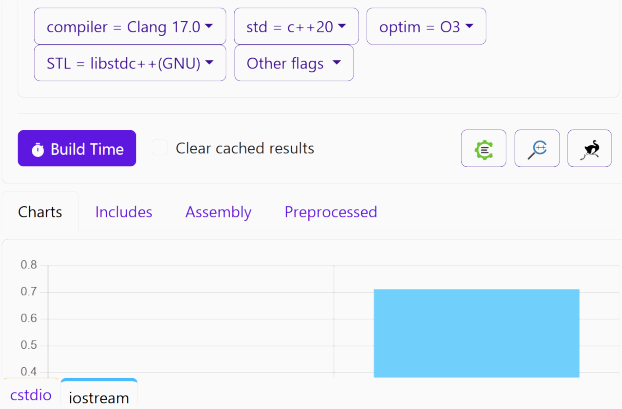
```
TemplateInstantiation CompleteTranslationUnit 6.635999943
0.48000000 TemplateInstantiation std::make_shared<TDataFrameAction
    <(lambda at /home/eguiraud/ROOT/root_install/include/ROOT/TDataFrame.hxx:739:31),
    TDataFrameImpl>
TemplateInstantiation std::make_shared<TDataFrameAction
    <(lambda at /home/eguiraud/ROOT/root_install/include/ROOT/TDataFrame.hxx:744:31),
    TDataFrameImpl>
TemplateInstantiation std::make_shared<Operations::FillTOperation, std::shared_ptr<TH1F>&,
    unsigned int> 0.12800000
TemplateInstantiation std::make_shared<TH1F, TH1F&> 0.11600000
TemplateInstantiation std::make_shared<Operations::FillOperation, std::shared_ptr<TH1F>,
    unsigned int&> 0.11599999
TemplateInstantiation std::make_shared<bool, bool> 0.10800000
```

Templight has not been updated recently. For instance, LLVM 15 is not supported.

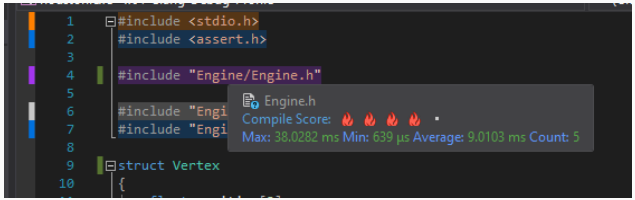
Build Bench

`Build Bench` is a tool to quickly and simply compare the build time of code snippets with various compilers.

"Hello world" example, `cstdio` vs. `iostream`



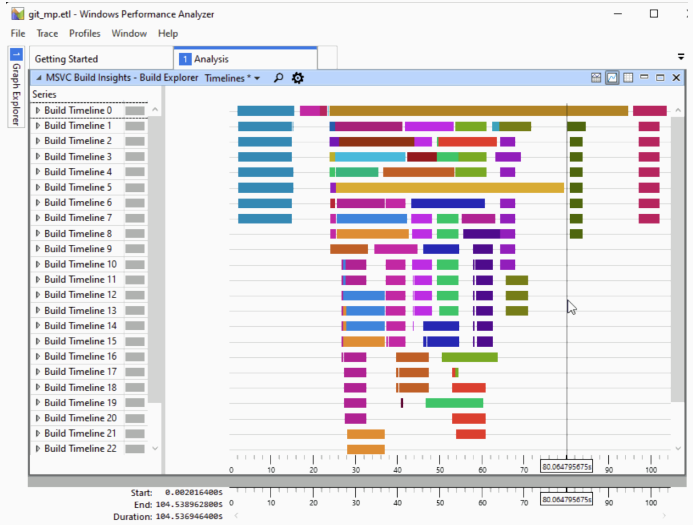
`VisualStudio - CompileScore` is an utility to display compilation profiling data directly inside Visual Studio. The tool uses `clang` to profile the compilation process.



C++ Build Insights is a set of build-profiling tools for MSVC that collects detailed compilation trace data and help finding where build time is being spent.

Such insights include:

- degree of parallelization
- switch to precompiled header (PCH)
- dominant compilation phase (parsing, code generation, or linking)
- aggregated statistics such as “most expensive headers/files to parse”


git_mp.etl - Windows Performance Analyzer
File Trace Profiles Window Help
Getting Started
1 Analysis
Graph Explorer
MSVC Build Insights - Build Explorer Timelines *
Series
Build Timeline 0
Build Timeline 1
Build Timeline 2
Build Timeline 3
Build Timeline 4
Build Timeline 5
Build Timeline 6
Build Timeline 7
Build Timeline 8
Build Timeline 9
Build Timeline 10
Build Timeline 11
Build Timeline 12
Build Timeline 13
Build Timeline 14
Build Timeline 15
Build Timeline 16
Build Timeline 17
Build Timeline 18
Build Timeline 19
Build Timeline 20
Build Timeline 21
Build Timeline 22
0 10 20 30 40 50 60 70 80.064795675s 90 100
Start: 0.002016400s
End: 104.538962800s
Duration: 104.536946400s
0 10 20 30 40 50 60 70 80.064795675s 90 100

`GitHub Copilot Build Performance` will use an agent to:

- Kick off a build and capture a trace for you
- Identify expensive headers and other bottlenecks
- Suggest and apply optimizations like precompiled headers
- Validate changes through rebuilds so your code stays correct
- Show you measurable improvements and recommend next steps

References

References

- The Hitchhiker's Guide to FASTER BUILDS, Viktor Kirilov (2019 edition)
- Chromium - Tips for improving build speed on Linux
- Common-sense acceleration of your MLOC build, Matt Hargett, CppCon14
- The Complete Guide to Speed Up Your C++ Builds
- Further Build Performance Optimizations