Refactoring Open Source Project OCC

A Journey of Build Time Optimization

Athanasiadis Panagiotis

Department of Computer Science and Engineering, University of Ioannina

April 22, 2026

Introduction to OCC

What is OCC?

A next-generation quantum chemistry and crystallography program designed for modern computational workflows, implemented in C++17 with Python bindings.

Quantum Chemistry Features:

- Hartree-Fock (RHF, UHF)
- Density Functional Theory
- Density fitting (RI-JK)
- Property calculations
- Implicit solvation (SMD)

Crystal Structure Analysis:

- CIF file processing
- Periodic bond detection
- Energy calculations
- Hirshfeld surfaces
- Dimer identification

OCC:Project Structure

Code Composition:

- **90.8%** C++ (core library)
- **10,000** lines of C++
- **3.7%** JavaScript (web tools)
- **2.8%** HTML (documentation)
- **1.6%** Python (bindings)
- **1.0%** CMake (build system)

Development Activity:

- 1,044 commits
- 45 releases
- 5 active contributors

Project Organization:

- `include/` - Header files
- `src/` - Implementation
- `tests/` - Test suite
- `docs/` - Documentation
- `examples/` - Usage examples
- `3rdparty/` - Dependencies

Dependencies: 11+ modern C++ libraries including Eigen3, libxc, libcint, gemmi

Community Engagement

21 stars • 875/month downloads • Active development since 2020

Project Goal: Improving Build Times

Current Challenge

With 10,000 lines of C++ code across multiple files, build times can significantly impact development productivity.

Build Time Impact:

- Slows down development cycles
- Increases testing feedback time
- Affects CI/CD pipeline efficiency
- Impacts developer experience

Contributing Factors:

- Large C++ codebase (90.8%)
- Template-heavy code (Eigen3)
- Multiple dependencies (11+)
- Complex file structure

Project Objective

Goal: Analyze and optimize the OCC build system to reduce compilation times while maintaining code quality and functionality.

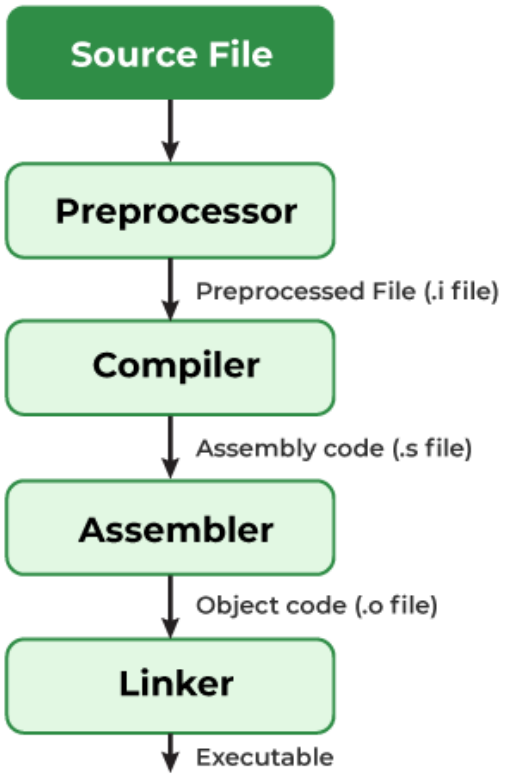
This involves identifying compilation bottlenecks and studying modern C++ build optimization techniques.

C++ Compilation Process

Understanding the Build Pipeline

Compilation Stages:

1. **Preprocessing**
   - Header file inclusion
   - Macro expansion
   - Conditional compilation
2. **Compilation**
   - Parsing source code
   - Semantic analysis
   - Code generation (object files)
3. **Linking**
   - Combine object files
   - Resolve symbols
   - Generate executable



Methodology: Analysis Tools & Techniques

Identifying Compilation Bottlenecks (Part 1)

1. Build Time Profiling

Ninja Log Analysis

- Parse `.ninja_log` files generated during build
- Extract per-file compilation times
- Identify slowest translation units
- Track build time trends across iterations

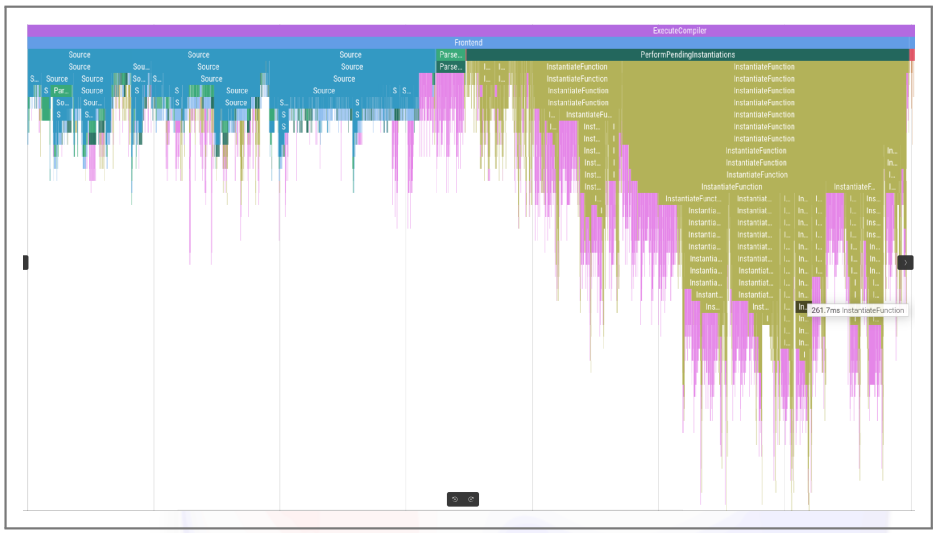
Clang Build Analyzer

- Generate time-trace profiles using `-ftime-trace`
- Detailed breakdown of compilation stages
- Template instantiation time analysis
- Header file inclusion impact measurement
- Function-level compilation time tracking

Methodology: Visualization & Code Review

Identifying Compilation Bottlenecks (Part 2)

Flame Graph Visualization



Hierarchical view showing which files and templates consume the most build time

Manual Code Review

- **Header Structure**
  - Analyze include hierarchies
  - Identify circular dependencies
  - Check for unnecessary includes
- **Template Usage**
  - Heavy template instantiations

Methodology: Static Analysis Tools

CppDepend

Name	# Issues	Added	Fixed	Elements	Group
Avoid methods too big, too complex	CppDepend.Core.CodeModelImpl.NumberOfMatch			methods	Project Rules \ Code Smells
Avoid methods potentially poorly commented	CppDepend.Core.CodeModelImpl.NumberOfMatch			methods	Project Rules \ Code Smells
From now, all methods added should respect basic quality principles	CppDepend.Core.CodeModelImpl.NumberOfMatch			methods	Project Rules \ Code Smells Regression
Assignment to Variable without Use ('Unused Variable')	CppDepend.Core.CodeModelImpl.NumberOfMatch			fields	Project Rules \ Coding Standards \ CWE Coding Standard
Don't assign static fields from instance methods	CppDepend.Core.CodeModelImpl.NumberOfMatch			fields	Project Rules \ Object Oriented Design
Projects with poor cohesion (RelationalCohesion)	CppDepend.Core.CodeModelImpl.NumberOfMatch			projects	Project Rules \ Object Oriented Design
Redundant code: Found a statement that begins with type constant.	CppDepend.Core.CodeModelImpl.NumberOfMatch			fields	Project Rules \ Misc
Unsigned variable varname cant be negative so it is unnecessary to test it.	CppDepend.Core.CodeModelImpl.NumberOfMatch			fields	Project Rules \ Misc
Potentially dead Methods	CppDepend.Core.CodeModelImpl.NumberOfMatch			methods	Project Rules \ Dead Code
Potentially dead Fields	CppDepend.Core.CodeModelImpl.NumberOfMatch			fields	Project Rules \ Dead Code
Instance fields should be prefixed with a 'm_'	CppDepend.Core.CodeModelImpl.NumberOfMatch			fields	Project Rules \ Naming Conventions
Static fields should be prefixed with a 's_'	CppDepend.Core.CodeModelImpl.NumberOfMatch			fields	Project Rules \ Naming Conventions
Types name should begin with an Upper character	CppDepend.Core.CodeModelImpl.NumberOfMatch			types	Project Rules \ Naming Conventions
Avoid methods with name too long	CppDepend.Core.CodeModelImpl.NumberOfMatch			methods	Project Rules \ Naming Conventions

Showing 1 to 14 of 14 entries

CodeChecker

Statistics

Number of processed analyzer result files	292
Number of analyzer reports	1325

Checker statistics

Checker name	Severity	Number of report
bugprone-forwarding-reference-overload	L	2
bugprone-incorrect-roundings	M	1
bugprone-integer-division	M	6
bugprone-misplaced-widening-cast	M	3
bugprone-signed-char-misuse	M	4
bugprone-sizeof-expression	M	1
bugprone-use-after-move	M	1
cert-dcl58-cpp	M	1
cert-dcl59-cpp	M	3
clang-diagnostic-bitwise-instead-of-logical	M	1
clang-diagnostic-deprecated-copy	M	18
clang-diagnostic-deprecated-copy-with-user-provided-copy	M	12
clang-diagnostic-error	C	4
clang-diagnostic-ignored-qualifiers	M	14
clang-diagnostic-mismatched-tags	M	1
clang-diagnostic-missing-field-initializers	M	23
clang-diagnostic-range-loop-construct	M	1
clang-diagnostic-reorder-ctor	M	8
clang-diagnostic-sign-compare	M	408
clang-diagnostic-unused-but-set-variable	M	2
clang-diagnostic-unused-function	M	2
clang-diagnostic-unused-lambda-capture	M	4
clang-diagnostic-unused-local-typedef	M	1
clang-diagnostic-unused-parameter	M	88
clang-diagnostic-unused-private-field	M	6
clang-diagnostic-unused-variable	M	108
core.CallAndMessage	H	2
core.DivideZero	H	1
core.NullDereference	H	2
core.uninitialized.Assign	H	7

Methodology: Quantifying Build Performance

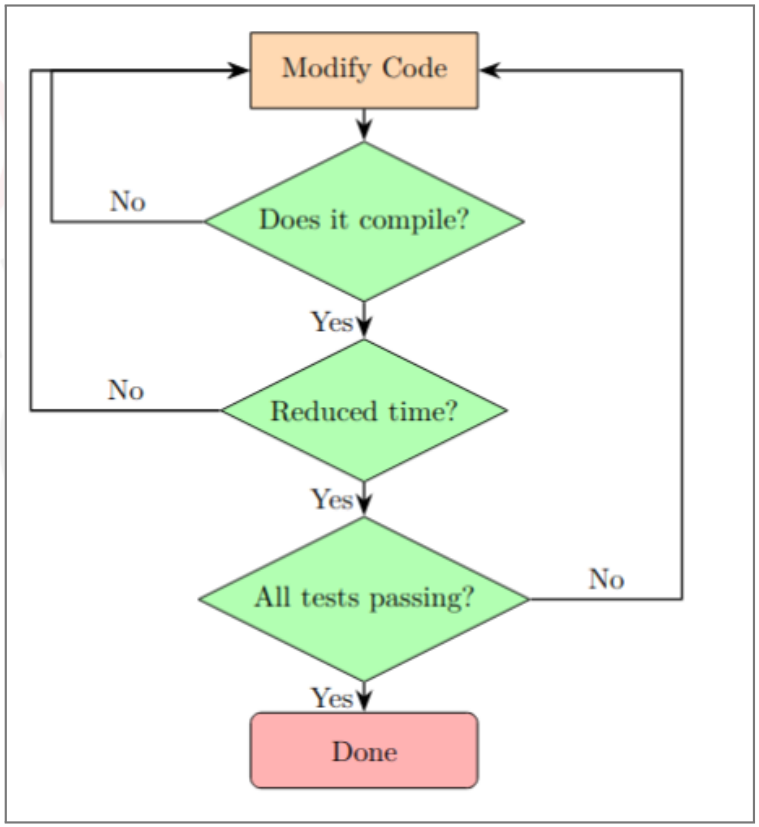
Objective: Establish a reliable baseline to measure the impact of refactoring on `occ` compilation speed.

Measurement Tool: Linux `time`

- Utilized the GNU `time` utility.
- **Key Metrics Tracked**:
  - **Real (Wall) Time**: The actual elapsed time perceived by the developer.

** Final results averaged over 10 clean builds to mitigate caching variance.*

Benchmarking Protocol:



Headers and Includes Audit

The Problem:

- Unnecessary includes in headers and source files
- STL headers are thousands of lines long
- Compiler copies every included file recursively
- No automatic duplicate detection across compilation units

Methodology:

- Bottom-up approach: source files first, then headers
- Used dependency graphs of core classes as guide
- Manual removal with compilation verification

Results: Removed approximately **330** unused includes from **167** files

Ccache and LLD Integration

Ccache (Compiler Cache):

- Speeds up recompilation by caching previous builds
- Hashes source code and compiler options
- Returns cached object files when matches are found
- Critical for rapid testing workflow
- Avoids long recompilation times during development

LLD (LLVM Linker):

- Drop-in replacement for system linkers
- Significantly faster than traditional linkers

Implementation: Configured in CMakeLists.txt to automatically use both tools when available on user's system

Forward Declarations

What are Forward Declarations?

- Technique to break unnecessary header dependencies
- Informs compiler of type's existence without internal details
- Works for functions, classes, and other types

Key Benefit: Reducing Recompilation Cascade

- Modified headers only trigger recompilation of files that explicitly include them
- Decouples source files from implementation details
- Small changes in low-level headers don't trigger project-wide rebuilds

Impact: Critical for rapid testing strategy and incremental development

Compiler Differences

Key Insight:

- Different compilers produce different machine code for same source
- Build times can vary significantly between compilers

Benchmark Results:

- GCC vs Clang comparison conducted
- **GCC was faster by 1 minute average**

Compilers Not Tested:

- Platform-specific: MinGW-w64 (Windows)
- Hardware-specific: Intel ICC
- Proprietary: Microsoft MSVC
- Outdated compilers

Suggestion: Use GCC for optimal build performance

Unity Builds

What are Unity Builds?

- Compilation technique merging multiple source files into single compilation units
- Reduces redundant header parsing overhead

Benefits:

- Compiler sees larger codebase portion $\rightarrow$ better optimizations
- Fewer object files $\rightarrow$ reduced linker workload
- Significant build time reduction

Implementation:

- Configured through CMake
- Batch size of 8 files provided best performance

Challenges:

- Fixed multiple One Definition Rule (ODR) violations
- Some files required exclusion from unity builds

Results: Substantial Build Time Reduction

Test System Configuration:

- Intel Core i7-3740QM processor
- 16GB DDR3 RAM

Full Build Time Results:

Before: 22 minutes (average)

↓ **70.5% reduction** ↓

After: 6.5 minutes (average)

More Information:

- Pull Request #45 (merged) on official GitHub
- Detailed code changes and implementation notes

Pull Request #45: Code Changes

Improving compile time #45

Edit | <> Code

Merged **peterspackman** merged 22 commits into `peterspackman:main` from `KonstantinosVard:main` 3 days ago

Conversation 3 | Commits 22 | Checks 0 | Files changed 162 | +207 −417

PanagiotisAthanasiadis commented on Dec 15, 2025 — Contributor

Audited the includes of source/header files for unused headers, implemented checking for ccache and lld in the Cmakefile, implemented Unity builds and created some forward declarations.

PanagiotisAthanasiadis and others added 20 commits 2 months ago

testing commit	cc40ba4
Test commit	fd813f1
test	f74b4c3
Merge branch 'main' of https://github.com/KonstantinosVard/occ_lambda_8	61a05e0
test on branch	1710bee
added ccache and lld checks	05d1afb
moved PES::solve_linear_system to .cpp to compile once	357a0a4
Removed unnecessary includes	7c6e569
Removed unnecessary libraries from integral_engine_df	48b8525
small changes to cint_interface.h	fed5cc0
Removed unnecessary includes from smd_salvation.cpp,occ_cube.cpp,mo.cpp	0bfdecb
Removed unnecessary libraries	a6bfd19
Removed unnecessary libraries mostly linear_algebra double includes	6b9fb28
Removed unnecessary libraries	bba5b2b
Removed unnecessary libraries from source files	84338ea
Removed unnecessary libraries from source files (final)	e6c1810

Reviewers
No reviews

Assignees
No one assigned

Labels
None yet

Projects
None yet

Milestone
No milestone

Development
Successfully merging this pull request may close these issues.
None yet

Notifications Customize
Unsubscribe
You're receiving notifications because you authored the thread.

3 participants

Allow edits by maintainers

Future Work

Future Work: C++20 Modules

C++20 Modules:

- Modern replacement for traditional header files
- Promising for reducing build times

Current Limitations:

- Not ready for mainstream adoption
- Most third-party libraries still use header files
- Standard library module support incomplete
- Significant refactoring effort required

Status: Monitoring ecosystem maturity before adoption

Future Work: PIMPL Pattern

Pointer to Implementation (PIMPL) Idiom:

- Separates class interface from implementation
- Hides private members in separate struct/class
- Accessed via opaque pointer

Benefits:

- Creates "compilation firewall"
- Implementation changes don't trigger client code recompilation
- Stable Binary Interface (ABI)
- Improved build times in large projects

Future Work: Linear Algebra Library Replacement

Current Challenge: Eigen Library

- Header-only library $\rightarrow$ slow build times
- Extensive template instantiation overhead

Potential Solutions:

- Evaluate compiled linear algebra alternatives
- Consider libraries with binary distributions
- Balance performance vs. compilation speed

Goal: Find alternative without sacrificing numerical performance

Questions?

Thank you for your attention!

Contact: pathanasiadis@cs.uoi.gr

Project Repository:
https://github.com/peterspackman/occ