TER report

Optimizing application performance through optimizing compilation

Prepared by
Francois Flandin

Supervised by
Pr Sid Touati

Master Informatique
at Université Côte d'Azur
First semester of year 2024-2025

Abstract

This report investigates the impact of compiler optimizations on application performance by comparing four compilers—`gcc` , `clang` , `icx` , and `ccomp` —across multiple optimization levels: `-O0` , `-O1` , `-O2` , `-O3` , and `-Os` .
Two programs, a matrix multiplication in `C` and Dijkstra's algorithm in `C++`, were used to evaluate execution times, highlighting how different compilers handle computationally intensive tasks and memory-heavy workloads.
The analysis shows that `gcc` consistently outperforms others in `C` tasks due to its robust optimization capabilities, while `clang` excels in `C++` scenarios, leveraging its deeper understanding of modern constructs.
`icx` and `ccomp` , though less performant, are designed for specific use cases like hardware-specific tuning and formally verified code, respectively.
The findings emphasize the importance of choosing the right compiler for specific tasks and suggest directions for further research, including energy-efficient optimizations and real-world mixed-language applications.

Contents

Introduction

The tutoring project titled "Optimizing Application Performance through Compilation Optimization" explores how compilers improve the generated code using various optimization levels: `-O0` , `-O1` , `-O2` , `-O3` , and `-Os` .
The different optimization levels goes from `-O0` , which means no optimization, to `-O3` , which enables the most optimizations, but also `-Os` , which compiles to have the most compact code.
This project is divided into two main phases:

- Experimentation: Compile two programs with four different compilers (`gcc` , `icx` , `clang` , and `ccomp`) at each optimization level, measure execution times over 12 iterations, and plot the data using `R`.
- Optimization Analysis: Study the optimization options enabled by each compiler at each level, to better understand their internal mechanisms.

These steps allow for the evaluation of compilers not only in terms of speed but also by identifying the specific characteristics of each optimization level.

State of the Art

Compiler optimizations play a crucial role in improving program performance by transforming source code into highly efficient machine code. Modern compilers like `gcc` , `clang` , and `icx` employ a range of techniques to optimize loops, eliminate redundant computations, and streamline memory usage. For example, `gcc` 's loop unrolling and vectorization strategies enable significant performance gains in computationally intensive workloads, while `clang` 's adherence to modern `C++` standards gives it an edge in handling advanced type systems and constructs.

Despite their capabilities, these compilers face trade-offs. `gcc` is known for its aggressive optimizations but may struggle with `C++`-specific features. `clang` prioritizes modularity and diagnostics, making it a preferred choice for developers working on complex `C++` codebases. `icx` , designed for Intel architectures, excels in leveraging hardware-specific features but shows weaker performance in general-purpose workloads. Meanwhile, `ccomp` stands apart as a formally verified compiler, prioritizing correctness over speed, making it indispensable for safety-critical applications.

Recent advancements in compiler technology focus on expanding interprocedural analysis, incorporating energy efficiency into optimization strategies, and enhancing debugging tools. However, challenges remain, such as balancing compile-time performance with runtime efficiency, ensuring cross-platform consistency, and addressing the growing demand for energy-efficient computing. These limitations underline the need for further research into compiler optimization techniques tailored for modern hardware and software ecosystems.

Part I
The experience

1 Experience's environnement

In this part will be presented the environnement of the experimentations, on which hardware it was done, on which software, etc.

Hardware

In this part, the hardware part of the computer will be explored.
To explore the hardware of a computer on linux, the folder `/cpu` can be explored, it gives us the *topology* of the cpu, like which threads are on each cores, which caches are on each cores, discover what is the cache line size etc...
But there are also a lot of linux tools that allows to check the hardware's specifications of a computer, and basically do the job for us.
There are graphical tools such as `likwid`, which provides the command `likwid-topology`, or the tool `hwloc`, which provides `lstopo`, a command that gives a beautiful GUI of the computer's hardware's specification.

Model name : 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz

Adress size : 39 bits physical, 48 bits virtual

Cache line size : 64 bytes

Cores : 4

Graphical Topology				
Coeurs	0 4	1 5	2 6	3 7
Cache L1	48 kB	48 kB	48 kB	48 kB
Cache L2	1MB	1MB	1MB	1MB
Cache L3	8 MB			

Table 1: Computer's topology

Software

For the software part, the experiments were done on a lightweight configuration of the computer, where all unnecessary services were disabled, including the graphical interface, this allows the computer to run as baremetal as possible.
Here is a description of all the software elements used in this benchmark :

OS Fedora Linux v40 WorkStation

gcc version 14.2.1 20240912

icx version 2024.2.1.20240711

clang version 18.1.8

ccomp version 3.14

2 Experiment method

To compare the performance of the different compilers, two programs will be used, the first `matrix_multiply`, is a `C` program that multiplies two matrices to put the result in a third matrix, it is an interesting challenge for optimization because of the presence of 3 nested loops for the calculation. This program will be compiled using `gcc` , `clang` , `icx` and `ccomp` .
The second program is `dijkstra` algorithm written in `C++`, the goal behind this is to compare the compilers optimizations when the program has a lot of memory usage, because it is in `C++`, the program will only be compiled with the `C++` versions of `gcc` , `clang` and `icx` , due to the lack of `C++` compilation from `ccomp` .
Next, we can mesure time taken by the program by placing the c-function `gettimeofday()` around the function that we need to calculate, initialisation mesurement is not the main goal, but it can be mesured too. The programs are executed 12 times for each optimization level to make an interesting violin plot using `R`.

3 Results

3.1 Matrix Multiplication Results

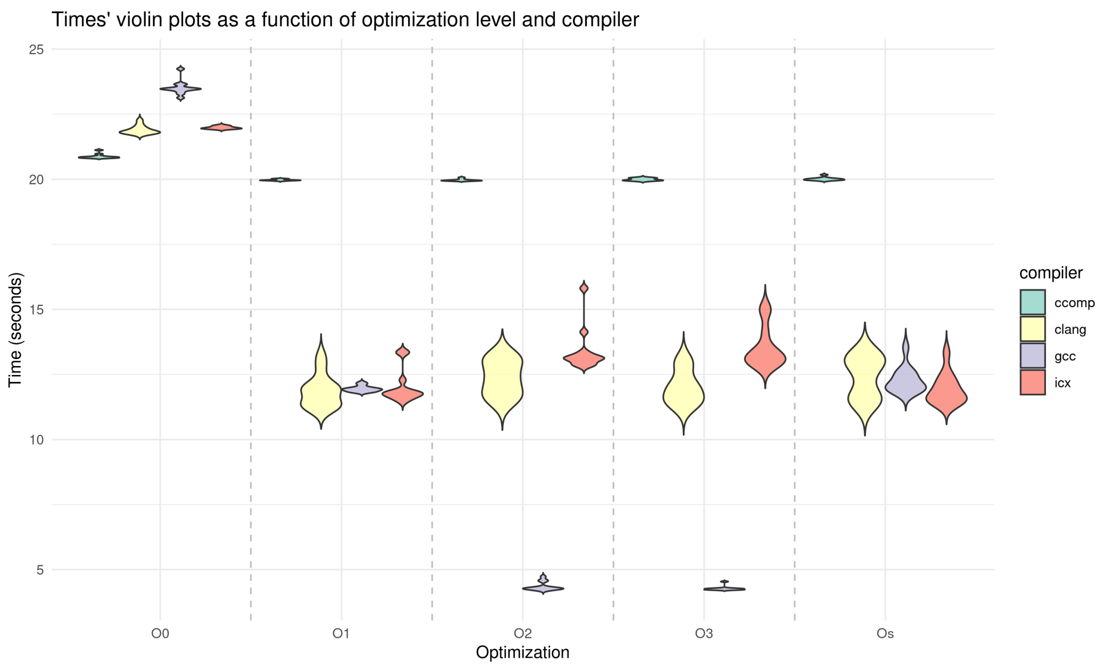

Figure 1: Evolution of the execution time of the program mat_mult.c as a function of compiler and optimization level.

The results for the C program (matrix multiplication) highlight clear differences in how each compiler handles optimizations.
`gcc` stands out as the best-performing compiler for this benchmark. It starts slower at -O0, but

quickly improves with each optimization level, reaching the best execution times at `-O2` and `-O3` . This suggests that `gcc` performs transformations which deal efficiently with the nested loop structure of the matrix multiplication making it suitable for such tasks.
`clang` shows a reliable improvement across optimization levels, but it remains behind `gcc` . While its performance gains are noticeable, it doesn't manage to handle the program as efficiently, especially at higher optimization levels. This difference suggests a less aggressive approach to handling the bottlenecks present in nested loops.
Despite being developed for Intel architectures, `icx` does not perform as well as expected. Its optimizations do improve execution times, but not enough to compete with`gcc` or `clang` . This makes `icx` less effective for matrix-heavy computations, suggesting it might prioritize other types of workloads.
`ccomp` delivers the slowest execution times in this benchmark, which is in line with its goal of producing formally verified code. While correctness is ensured, performance is not a priority, and its constant execution times across optimization levels reflect this limitation.

3.2 Dijkstra Results

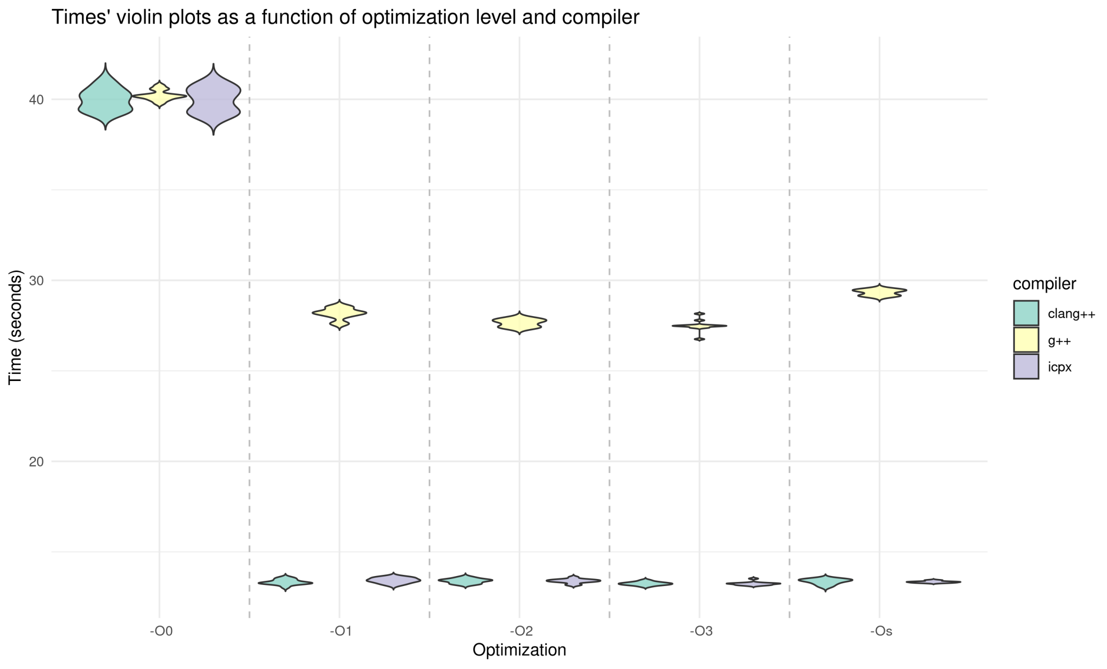

Figure 2: Evolution of the execution time of the program dijkstra.cpp as a function of compiler and optimization level.

What we can notice from these 2 graphs, is how differently `gcc` compares with `clang` and `icx` according to the language used, `C` or `C++`. When using `C`, `gcc` produces way more optimized codes than `clang` and `icx` , whereas in `C++`, `gcc` is the least performant one.
Few experiments were done get some answers to why is it like this.

First, the experiments were made on the `C++` programs, at the compilation level of `-O3` , this will give us some data to work with, and, by using the command `perf record` and `perf report`, we can use material performance counters to get a lot of data on the percentage of time used by a function, we could use profiling applications such as `gprof`, but `perf` allowed us to have more precise data and avoided us to recompile entirely the program.

The results were the following:

	`gcc`	`clang`	`icx`
total time taken	136.37s	146.98s	155.65s
dijkstra usage	$\approx 26s$	$\approx 13s$	$\approx 17s$
init/free usage	$\approx 109s$	$\approx 133s$	$\approx 138s$

What we can get as results from here is that `gcc` is the most optimized in terms of initialisation and memory management, but is the worst one in terms of pure dijkstra calculation.
If we push further the analysis, we can see there are some points in the program where types inference were used, such as `for (auto edge :  temp_neighbors)`. `gcc` might not be able to manage this type system as well as `clang` or `icx` , hence the lack of optimizations.

Part II
Compilers

In this part will be presented all the compilers used in this tutorship project, as well as deepening their optimization options for each optimizations levels.
The optimizations presented are not exhaustive, only the most important one will be presented, and the method to obtain the optimizations will be given for each compiler, so the experiments can be reproduced.

4 `gcc`

`gcc` is the most popular `C` compiler, it comes in the `GNU Compiler Collection`, which includes compilers for `Fortran`, `C++`, `Go`...

Optimizations Obtention Method

`gcc` is really helpful regarding getting informations about what he does, because the compiler has an option for this sole purpose, which is

```
gcc --help=optimizers -Q -On
```

Optimizations introduced by `-O0`

`gcc -O0` was originally meant to disable every optimization flags to give the least optimized code, however, the competition with other compilers led to now have some optimization flags enabled, 53 to be more precise. For the most, optimizations at this level are analysis options.

A part of the optimizations concerns loop optimizations, with the option
`-faggressive-loop-optimizations`, `gcc` enables, as its name suggests, a lot of loop optimizations, such as **loop unrolling**, which reduces loop overhead by duplicating the loop body multiple times, minimizing the number of iterations and improving execution efficiency, **loop fusion**, which combines loops that access the same data into a single loop, reducing overhead from multiple passes over the data, or **loop peeling**, which moves loops special cases outside of the loop.

Next is the dead code elimination, `gcc` can also remove every line of code that isn't used, this doesn't impact the execution speed, but reduces the size of the generated code.
And finally are the peephole optimizations this option introduces optimizations on a reduced window, optimizing redundant register moves, redundant calculations, etc.
Other enabled options at `-O0` are either analysis options or add error-handling instructions.

Optimizations introduced by `-O1`

Intermediate Tree Optimizations
`gcc -O1` optimizations add optimizations to the tree that the compiler builds in the intermediate level.
First, there is the dead code and redundancy elimination, with the options `-ftree-dce`, `-ftree-fre` and `-ftree-dse`, `gcc` removes all the **dead code**, computations that produce the same results as well as all **dead stores**, that means that the cpu has less operations to manage, and the code

size is reduced.
Secondly, there is value and copy propagation, first, the option `-ftree-ccp` tells the compiler to propagate constant values across conditions when it's possible, it simplifies all the control flow and makes place for further optimizations, and can even help for `branch prediction`.
Secondly, the option `-ftree-copy-prop` propagates values with copy instructions, for example `b = a; c = b;` would be replaced with `c = a;`.
And finally, loop optimizations, there are two main options here to consider, the first, `-ftree-sink` moves computations outside of the loop when it's possible, it allows to avoid computing the same thing over and over.
The second is `-ftree-sra` for *scalar replacement of aggregates*, it breaks structures and array into individual variable when needed, it can improve register usage and enable even further optimizations. An example of this would be a variable of the struct `Point {int x; int y}` would be transformed from `Point p;` to `int p_x, p_y;`.

At `-O1` , there are also other options to enable inlining of functions, that means that a function call will be replaced by the function's body, which allows for further optimizations.

Optimizations introduced by `-O2`

Interprocedural Analysis Optimizations
`gcc -O2` introduces `IPA` (Interprocedural Analysis) optimizations, which aim to improve performance by analyzing and optimizing across function boundaries. This includes `-fipa-icf` (Identical Code Folding), which detects and merges functions or read-only variables with identical implementations, reducing code size and memory usage.
Additionally, `-fipa-cp` (Constant Propagation) propagates constant values across function calls, allowing the compiler to simplify computations and eliminate unnecessary branches. These optimizations are particularly effective in programs with repetitive patterns or predictable input values, contributing to leaner and faster executables.

Aligning
`gcc -O2` introduces aligning of functions, jumps, labels, and loops, the aligning allows to ensures that the instructions fit at the beginning of cache lines, which can reduce cache misses. Also, aligning labels and jumps allows to decrease branch prediction misses.

`-O2` also enables `-funroll-loops`, which performs loop unrolling when iteration count is known, which allows the compiler to optimize as it wishes.

Optimizations introduced by `-O3`

Loop Optimizations
`gcc -O3` enables mostly further loops optimizations, such as `loop peeling`, `loop jamming`, which combines independent loops (that may or may not access the same data) into a single loop, and is generally combined with `loop unrolling`.
`loop interchange`, which swap two loops in a nested loop, this allows to put the most changing variable to the consecutive elements in memory, instead of having the most changing variable to separate elements in memory, it's very useful in programs such as `matrix multiplication`.
Finally, `loop splitting` allows simplification of a loop by dividing it into several loops, it simplifies the loops with the pattern `for (int` $i = 0$`;` $i < 10$`;` $i++$`) if(`$i < 5$`)... else...` into two loops, one with the range 0..4 and the other 5..9. It helps in reducing dependencies.

Optimizations enabled by `-Os`

`gcc -Os` enables the same options as `gcc -O2` , but removes the aligning optimizations to free some space.

5 `icx`

It's Intel's compiler for their x86 architecture, it is made to deliver extremely optimized programs on their processor architecture.

Optimization Obtention Method

Everything is available in the online documentation, it's relatively easy to find what we need.

Optimizations introduced by `-O0`

Everything is disabled at `-O0` .

Optimizations introduced by `-O1`

At this level of optimization, `icx` starts by introducing **data flow analysis**, to help gathering data about data flows throughout the whole program, allowing for optimizations.
Then, the compiler can also reorganize code as he wishes with **code motion**, which allows to move non-changing result operations out of loop to avoid recomputing it over and over, and **instruction scheduling**, which allows to change the order of instructions, to hide memory latency for exemple, or doing something while something heavier is done. It is done to reduce CPU stall.

Other optimizations are also enabled, **strength reduction**, that changes heavy operations for cheaper ones, like switching from a multiplication to a bit shift, **test replacement**, that simplifies loops' conditions for example :

```
if (x >= 0 && x < 100) { /* code */ } // base loop
if ((unsigned)x < 100) { /* code */ } // optimized loop
```

And, finally, **split-lifetime analysis**, can split the *lifetime* of a variable into smaller lifetimes, to reallocate the registers when the variable is not used anymore.

Optimizations introduced by `-O2`

At this level of optimization, `icx` introduces more optimizations, like **constant propagation**, or **forward substitution**, which can replace variable with their expressions, for example :

```
int a = 5 + i
int b = a + 3
```

will be replaced by

```
int b = 5 + i + 3
```

These two options complete each other really well, because imagine that the variable `i` was a constant, it would be replaced by its value, hence further optimizing the program.

Additionnal optimizations are introduced, like **loop unrolling**, **peephole optimizations**, which examines small code fragments to identify and replace inefficiencies, such as redundant instructions or suboptimal sequences, improving code quality. Or even **optimized code selection**, which replaces inefficient instructions or code sequences with more efficient alternatives.

There are also code-removing optimizations such as **dead-code elimination**, **dead-store elimination**, and **dead static function elimination** which does not really improves performance but can help reduce the weight put on the CPU.

Optimizations introduced by `-O3`

It enables all the same optimizations as `-O2` but add more aggressive loop transformations such as **loop fusion**, **block-unroll-and-jam**, with these two options, the compiler expands and find the best fusions/jams in the loops and **collapsing-IF-statements**, where multiple successive if are combined into one to reduce branching therefore gaining performances.

Optimizations introduced by `-Os`

The documentation doesn't provide any information about it, except the usual "like `-O2` but optimizes for code size".

6 `ccomp`

Certified Compiler
It's the only C compiler that is formally verified to produce a code described by the source code, optimized or not. By its formally verified nature, it is the one that produces the most inefficient code. It is generally used for safety critical programs where a bug introduced by compilation could lead to serious problems.

Optimization Obtention Method

Everything is available in the online documentation, it's relatively easy to find what we need.

Optimizations introduced by `-O0`

`ccomp` doesn't enable anything at `-O0` , and the main optimization level `-O` gives the same optimizations as `-O1` , `-O2` , `-O3` , because they are just an alias of that optimization option.

Optimizations introduced by `-O1` ,`-O2` ,`-O3`

There are not a lot of options so here is a list of them

-fcont-prop enables constant propagation, a process that replaces variables with their known constant values throughout the program.

-fcse activates the elimination of common subexpressions, reducing redundant calculations by reusing previously computed values.

-fif-conversion with this option, the compiler decides whether to replace simple if-then-else statements or the conditional operator (?:) with conditional assignements.
The heuristic-based approach selectively optimizes small and balanced expressions, provided the target architecture supports a suitable conditional move instruction.

-finline enables or disables the inlining of functions, potentially improving performance by replacing function calls with the actual function code.

-finline-functions-called-once specifically inlines functions that are called only once, reducing overhead while maintaining efficiency.

-fredundancy activates the elimination of redundant computations and unnecessary memory stores, improving execution time by avoiding repetitive operations.

-ftailcalls optimizes function calls in tail position, which can enhance performance by reusing the current function's stack frame.

-ffloat-const-prop 2 enables full propagation of floating-point constants, ensuring arithmetic is performed in IEEE double precision format with round-to-nearest mode.

Optimizations introduced by `-Os`

The documentation doesn't give a lot of informations about it. Just the usual "optimizes for code size".

7 `clang`

It's a compiler made by the LLVM Developer Group, it can replace `gcc` by supporting most of the option flags of `gcc` .
The objective behind this compiler is to create a compiler that allowed: first, better diagnostics, which allows to debug more easily, second, to be seperated from the GNU licence, which forced softwares to be integrated to said licence and therefore having to opensource proprietary software, third, to have a nimble compiler that is simple and easy to develop and maintain.

Optimization Obtention Method

`clang` 's documentation does not explicitly specify all enabled optimizations, however, it has some commands that can give what is enabled with each optimization level, there are 2 commands because of the way clang works.
The compiler is divided in two parts, `clang` , the part where the intermediate representation (IR) is made, and `opt`, the part that optimizes said IR. The first command is used to know which optimizations are enabled by `opt`.

```
llvm-as < /dev/null | opt -On -disable-output -debug-pass-manager
```

The second command gives the optimizations added by `clang` itself

```
diff -wy --suppress-common-lines \
    <(echo 'int;' | clang -xc - -o /dev/null -\#\#\# 2>&1 | tr " " "\n" | grep -v /tmp) \
    <(echo 'int;' | clang -xc -On - -o /dev/null -\#\#\# 2>&1 | tr " " "\n" | grep -v /tmp)
```

Another way of doing this, which gives way more optimizations is the combination of the following commands, which will give us optimizations activated for a specific program.

```
clang -O2 -emit-llvm -S matrix_multiply.c -o mat_mult.ll
opt -O2 -debug-pass-manager mat_mult.ll -o mat_mult.ll
```

Optimizations enabled at `-O0`

`clang` like the other compilers doesn't enable a lot of optimizations, the optimizations are called **AlwaysInlinerPass**, which inlines functions marked with the `always_inline` attribute, **CoroConditionalWrapper**, which is related to coroutines support in C++, which is a function that can be paused and resumed, enabling asynchronous and cooperative task management.
Finally, **VerifierPass** ensures that the LLVM Intermediate Representation generated by the compiler is correct and coherent with the original program.

Optimizations introduced by `-O1`

Loop Optimizations
At `-O1` , `clang` applies several key loop optimizations to improve runtime performance. The **LoopSimplifyPass** restructures loops into a canonical form, making them more straightforward to analyze and optimize in later stages. **LCSSAPass** (Loop-Closed SSA) transforms loops to a representation that simplifies memory access and dependency analysis. **IndVarSimplifyPass** refines induction variables, like loop counters, simplifying conditions and arithmetic for better execution flow. Additionally, the **LoopRotatePass** adjusts loop headers to enhance branch prediction and pave the way for more aggressive transformations, such as unrolling, in higher optimization levels. While full unrolling and vectorization are not yet fully utilized at this stage, these preparatory steps create a solid foundation for more advanced optimizations, all while keeping compilation times reasonable.

Coroutines
For programs utilizing coroutines, `clang` introduces key transformations to handle their unique execution model. **CoroEarlyPass** prepares coroutine constructs by splitting their code into manageable parts for further optimization. **CoroElidePass** eliminates unnecessary coroutine frames when they are not required, reducing runtime overhead. Finally, the **CoroCleanupPass** finalizes coroutine transformations by removing temporary constructs and generating efficient, optimized code for coroutine-based workflows. These coroutine-specific passes ensure that `clang` can handle modern programming paradigms efficiently, even at `-O1` .

Global Optimizations
`clang` 's `-O1` also includes broader global optimizations that significantly impact performance and code quality. **IPSCCPPass** (Interprocedural Sparse Conditional Constant Propagation) enables the propagation of constant values across function boundaries, revealing opportunities for eliminating redundant computations and simplifying control flow. The **LICMPass** (Loop-Invariant Code Motion) complements loop optimizations by moving computations that remain constant across iterations out of the loop, reducing redundant calculations. Additionally, **DeadArgumentEliminationPass** simplifies function signatures by removing unused arguments, and **MemCpyOptPass** optimizes memory operations like `memcpy` by replacing them with more efficient alternatives where possible. Together, these optimizations strike a balance between improving runtime performance and maintaining fast compilation speeds, providing a strong baseline for further enhancements at higher optimization levels.

Optimizations introduced by `-O2` , `-O3` , `-Os`

At `-O2` , `-O3` , and `-Os` , `clang` focuses on impactful optimizations like **GVNPass** (Global Value Numbering), which eliminates redundant computations, and **DSEPass** (Dead Store Elimination), which removes unnecessary memory writes.
SLPVectorizerPass enhances performance by vectorizing independent operations for SIMD, while **JumpThreadingPass** simplifies control flow by bypassing unnecessary conditions. Together, these key passes significantly improve execution efficiency and streamline code generation for both performance-critical and size-constrained applications.

Conclusion

This project has given us a better understanding of how various compilers handle optimizations and how these optimizations impact performance.
The experiments revealed that `gcc` is the top performer when it comes to optimizing nested loops in `C` programs, consistently delivering superior results. On the other hand, `clang` showed its strength with `C++` workloads, surpassing `gcc` in tasks like Dijkstra's algorithm thanks to its deep understanding of modern `C++` features. Despite being designed for Intel architectures, `icx` did not perform well in highly computational scenarios, suggesting that its optimizations may be geared towards other types of workloads. `ccomp` , on the other hand, focused on formally verified code, highlighting the balance between correctness and performance and making it a unique but specialized tool for safety-critical applications.

In conclusion, it is essential to choose the right compiler for specific tasks. While `gcc` is versatile and reliable for general-purpose and performance-critical applications, `clang` shines in `C++` workloads due to its modern language features support. `icx` and `ccomp` , although not leading in raw performance, cater to specific use cases, showing that each compiler has its strength in different scenarios.

This analysis opens the door for future exploration, such as investigating how compilers can optimize code to be more energy-efficient. Because since the beginning of computer science the major concern was pure performance and less energy-efficiency, this could be where technological advancements truly shine–not just in creating more powerful computers–but in developing more efficient ones. A prime example of this approach is Apple's ecosystem, which achieves impressive performance without consuming excessive energy. Future innovations could follow this model, focusing on both power and efficiency.

References

LLVM Group. Clang compiler documentation, 2024. URL https://clang.llvm.org/.

Intel Corporation. Intel® oneapi dpc++/c++ compiler developer guide and reference, 2024. URL https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2024-2/o-001.html.

Xavier Leroy. The compcert c verified compiler, documentation and user's manual. version 3.14, 2024. URL https://compcert.org/man/manual003.html#sec26.