Optimizing application performance through optimizing compilation

Francois Flandin

Encadré par
Pr Sid Touati

First semester of year 2024-2025

Objective

Explore how compilers optimizes programs using several optimizations levels : `-O0` , `-O1` , `-O2` , `-O3` , `-Os` .

2 parts to the project

1. Compiling 2 programs in `C`/`C++`with each optimization level and compiler to compare performances
2. Checking which optimization is enabled for each optimization level and compiler

Content

Model name : 11th Gen Intel Core i5-1135G7 @ 2.40GHz

Adress size : 39 bits physical, 48 bits virtual

Cache line size : 64 bytes

Physical cores : 4

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

Table – Computer's topology

Configuration

Computer in a lighweight configuration, avoid OS's optimizations and bloat from other programs or graphical interface.

OS and compilers

OS : Fedora Linux Workstation v40
gcc : version 14.2.1
icx : version 2024.2.1
clang : version 18.1.8
ccomp : version 3.14

Programs

There are 2 programs to compile : Matrix Multiplication (`C`) and Dijkstra's algorithm (`C++`). For each compiler and optimization level.

Object Size

The matrix size is set to two times the size of the largest cache. The graph size is really huge to have a significate execution time.

Mesurement

Function `gettimeofday()` placed before and after the main computation, mesuring initialisation time is not the goal.

Finally

Run each program 12 times to visualize the data with R

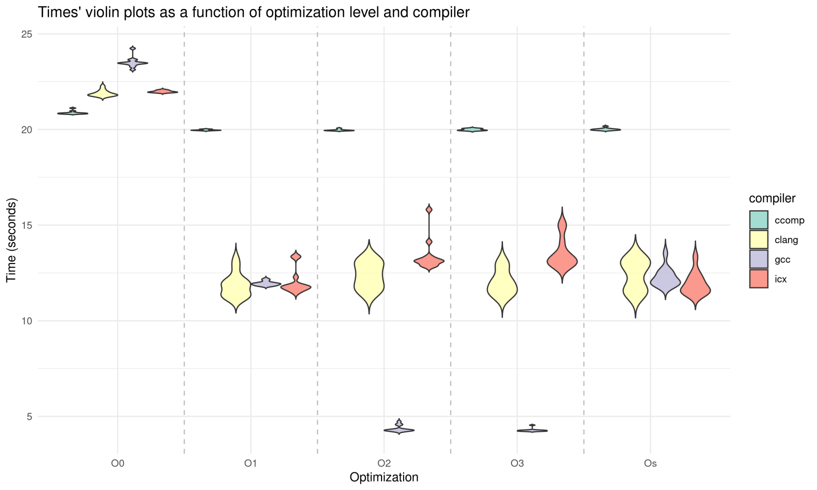

Figure – Evolution of the execution time of the program mat_mult.c as a function of compiler and optimization level.

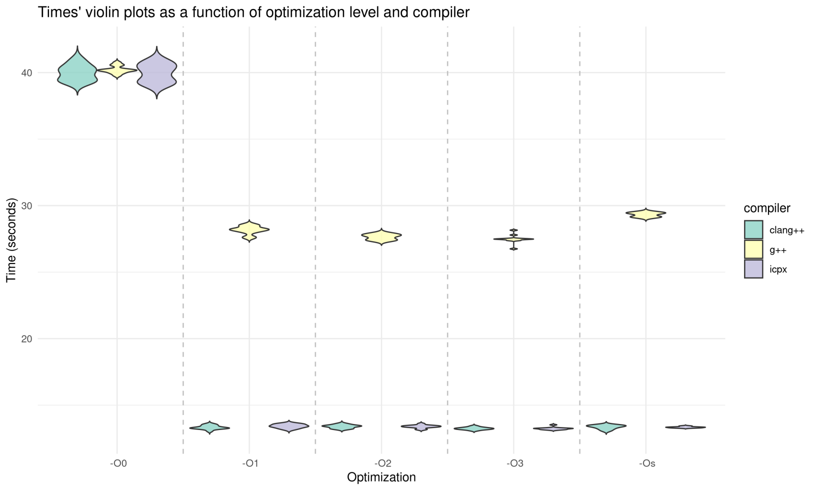

Figure – Evolution of the execution time of the program mat_mult.c as a function of compiler and optimization level.

	gcc	clang	icx
total time taken	136.37s	146.98s	155.65s
dijkstra usage	$\approx 19\%$	$\approx 9\%$	$\approx 11\%$
init/free usage	$\approx 81\%$	$\approx 91\%$	$\approx 89\%$

gcc

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

clang

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

icx and ccomp

Everything is inside their documentation.

Peephole optimizations

Optimize a program over a reduced windows that slides through the program.

Peephole optimizations

Optimize a program over a reduced windows that slides through the program.

Strength reduction

Replace heavy computations with lighter ones.

Peephole optimizations

Optimize a program over a reduced windows that slides through the program.

Strength reduction

Replace heavy computations with lighter ones.

Code motion

Allow the compiler to move instructions around the program.

Peephole optimizations

Optimize a program over a reduced windows that slides through the program.

Strength reduction

Replace heavy computations with lighter ones.

Code motion

Allow the compiler to move instructions around the program.

Elimination of common subexpressions

Reuse previously computed values.

Loop unrolling

Reduces loop overhead by duplicating loop's body several times.

Loop unrolling

Reduces loop overhead by duplicating loop's body several times.

Loop interchange

Improve memory access by modifying the loops order in a serie of nested loops.

Loop unrolling

Reduces loop overhead by duplicating loop's body several times.

Loop interchange

Improve memory access by modifying the loops order in a serie of nested loops.

Loop peeling

Simplifies loops by removing non changing computation out of the loop.

Loop unrolling

Reduces loop overhead by duplicating loop's body several times.

Loop interchange

Improve memory access by modifying the loops order in a serie of nested loops.

Loop peeling

Simplifies loops by removing non changing computation out of the loop.

Loop jamming/fusion

Combine two adjacent loops into a single loop.

Dead code elimination

Removes part of the code that are not used.

Dead code elimination

Removes part of the code that are not used.

Dead store elimination

Removes variable affectations that aren't used through the program.

Dead code elimination

Removes part of the code that are not used.

Dead store elimination

Removes variable affectations that aren't used through the program.

Dead static function elimination

Removes static functions that are never used.

Dead code elimination

Removes part of the code that are not used.

Dead store elimination

Removes variable affectations that aren't used through the program.

Dead static function elimination

Removes static functions that are never used.

Dead argument elimination

Removes unused function arguments.

Coroutines in C++ are functions that can be paused and resumed, enabling asynchronous and cooperative task management.

CoroEarlyPass

Prepares coroutine constructs by splitting their code into manageable parts.

Coroutines in C++ are functions that can be paused and resumed, enabling asynchronous and cooperative task management.

CoroEarlyPass

Prepares coroutine constructs by splitting their code into manageable parts.

CoroElidePass

Eliminates unnecessary coroutine frames when they are not required.

Coroutines in C++ are functions that can be paused and resumed, enabling asynchronous and cooperative task management.

CoroEarlyPass

Prepares coroutine constructs by splitting their code into manageable parts.

CoroElidePass

Eliminates unnecessary coroutine frames when they are not required.

CoroCleanupPass

Finalizes coroutine transformations by removing temporary constructs and generating efficient, optimized code.

Does not activate much, but it has its optimizations !

Constant propagation

Replace constant variable calls with the value.

Does not activate much, but it has its optimizations !

Constant propagation

Replace constant variable calls with the value.

Common subexpression elimination

Reuse previously computed values.

Does not activate much, but it has its optimizations !

Constant propagation

Replace constant variable calls with the value.

Common subexpression elimination

Reuse previously computed values.

Inlining

Replace function calls with function's body.

gcc

We could see that gcc is the most efficient in general-purpose tasks and nested loops in C.

clang

clang was the most performant on C++programs using specific features like type inference.

icx

icx not being the most performant on this configuration could mean that the compiler targets other types of workloads.

ccomp

ccomp focuses on formally verified code translation so optimizations and performance is not a question.

Each compiler has its own purpose, so it's important to choose the right compiler for a specific task.

What next ?

Further exploration on *energy consumption* optimization, in line with the model of Apple, a balance between *power* and *efficiency*.