UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

ENHANCING THE CLANG COMPILER WITH OPENMP PARALLELISM: A NOVEL APPROACH USING CLANGIR

WALTER JOSÉ TROIANI VARGAS

Thesis supervisor: JOSE MIGUEL RIVERO ALMEIDA (Department of Computer Science)

Thesis co-supervisor: ROGER FERRER IBÁÑEZ

Degree: Bachelor's Degree in Informatics Engineering (Computing)

Bachelor's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Abstract

This undergraduate thesis endeavors to elevate the C/C++ front-end of the LLVM (Clang) by enriching it with new capabilities through the ClangIR high-level intermediate representation. This innovative approach promises to revolutionize the conventional compilation and optimization workflows for C/C++/Objective-C++ languages, by improving its analysis, bug diagnostics, and optimization capabilities.

Our focus lies in implementing OpenMP task directives, a pivotal addition that empowers this compiler novel intermediate representation to embrace parallelism through the OpenMP shared-memory multithreading API. By integrating OpenMP support into ClangIR, we aim to augment the performance and coverage of the code generation of this new compilation flow of Clang, while fostering a conducive environment for future contributors to extend and enhance its functionality.

Through this open-source contribution, we aspire to propel ClangIR, thereby advancing the capabilities of Clang and opening avenues for further innovation in the realm of compiler technologies for C-based languages. Finally, another key goal of this dissertation is to become a valuable introductory resource for anyone wishing to delve into modern compiler development.

Abstract

Aquest treball de fi de grau s'esforça per enriquir el front-end C/C++ del projecte LLVM (Clang) amb noves capacitats a través de la representació intermèdia d'alt nivell ClangIR. Aquest enfocament innovador promet revolucionar els fluxos de compilació i optimització convencionals per a llenguatges C/C++/Objective-C++, millorant-ne l'anàlisi, la diagnosi d'errors i les capacitats d'optimització.

El nostre focus resideix en la implementació de directives de tasques OpenMP, una incorporació clau que permet aquesta nova representació intermèdia del compilador explotar el paral·lelisme mitjançant l'API de multithreading de memòria compartida d'OpenMP. Mitjançant la integració del suport d'OpenMP a ClangIR, aspirem a augmentar el rendiment i la cobertura de la generació de codi d'aquest nou flux de compilació de Clang, mentre que fomentem un entorn propici per a futurs col·laboradors per estendre i millorar la seva funcionalitat.

A través d'aquesta contribució de codi obert, aspirem a impulsar ClangIR, avançant així les capacitats de Clang i obrint vies per a una major innovació en l'àmbit de les tecnologies de compiladors per a llenguatges basats en C. Finalment, un altre objectiu d'aquesta dissertació és esdevenir un recurs introductori valuós per a tot aquell que vulgui adentrar-se en el desenvolupament de compiladors moderns.

Abstract

Este trabajo de fin de grado se esfuerza por enriquecer el front-end C/C++ del proyecto LLVM (Clang) con nuevas capacidades a través de la representación intermedia de alto nivel ClangIR. Este enfoque innovador promete revolucionar los flujos de compilación y optimización convencionales para lenguajes C/C++/Objective-C++, mejorando su análisis, diagnóstico de errores y capacidades de optimización.

Nuestro foco reside en la implementación de directivas de tareas OpenMP, una incorporación clave que permite a esta nueva representación intermedia del compilador explotar el paralelismo mediante el API de multithreading de memoria compartida de OpenMP. Mediante la integración del soporte de OpenMP en ClangIR, aspiramos a aumentar el rendimiento y la cobertura de la generación de código de este nuevo flujo de compilación de Clang, mientras fomentamos un entorno propicio para futuros colaboradores para extender y mejorar su funcionalidad.

A través de esta contribución de código abierto, aspiramos a impulsar ClangIR, avanzando así las capacidades de Clang y abriendo vías para una mayor innovación en el ámbito de las tecnologías de compiladores para lenguajes basados en C. Finalmente, otro objetivo de esta disertación es convertirse en un recurso introductorio valioso para todo aquel que desee adentrarse en el desarrollo de compiladores modernos.

Acknowledgements

I am profoundly grateful to all the teachers who have guided me during this remarkable four-year journey. Among them, I owe special gratitude to Conrado Martínez Parra, whose passion for computer science ignited my interest and whose unwavering support has been invaluable. I am equally indebted to José Miguel Rivero Almeida, whose enthusiastic instruction has equipped me with the knowledge and skills essential to understanding compilers. I'm grateful for the great teachers that my faculty has provided me, that have sparked my curiosity and hunger for knowledge.

A heartfelt shoutout goes to Roger, whose mentorship has been crucial in navigating the complexities of this thesis and whose inspiration has motivated me to embark on this transformative journey. This work would not have been possible without the contributions of the CIR community, particularly Bruno Cardoso Lopes, Fabian Mora, and Kiran Chandramohan.

To the dear reader who has taken the time to explore this captivating niche of compilers and computer science, I extend my sincere appreciation. Your interest and curiosity fuel my enthusiasm for sharing knowledge.

Lastly, I extend my deepest gratitude to the friends who have accompanied me along this journey and had a direct and positive impact on my life, especially Alex Herrero [1], for the guidance he has provided throughout the thesis, as well as Lluc Clavera and Pau Ribelles. If you are reading this dissertation, there is a 99% chance that you belong to this set of friends. Words cannot express how grateful I am to have gotten to know you!

To my former computer architecture pupils, whose camaraderie has enriched my experience and shaped my growth, and whose trust in me to educate them instilled a sense of purpose and responsibility that motivated me to rise early on the roughest days of my degree, thank you. Last but not least, to my partner Patricia, whose unwavering support and the mental respite she provided me, even amidst the busiest and hardest of times, have been a constant source of strength and encouragement.

Contents

1 Introduction **8**
1.1 The problem . . . 8
1.2 Context of the thesis . . . 8
1.3 Concepts and Terms . . . 9
1.3.1 Compilers . . . 9
Front-end . . . 9
Middle-end . . . 10
Back-end . . . 11
1.3.2 LLVM Project . . . 12
OpenMP . . . 12
LLVM IR . . . 12
MLIR . . . 12
ClangIR . . . 13
1.4 Stakeholders and Reach . . . 13

2 Justification **15**

3 Scope **15**
3.1 Objectives and Requirements . . . 15
3.1.1 Main Objectives . . . 15
3.2 Requirements . . . 16
3.2.1 Functional Requirements . . . 16
3.2.2 Non-functional Requirements . . . 16
3.3 Obstacles and Risks . . . 16

4 Methodology and Rigor **17**

5 Description of the Tasks **17**
5.1 Management . . . 17
5.2 Documentation . . . 18
5.3 Environment . . . 18
5.4 Research . . . 19
5.5 Implementation . . . 19
5.6 Miscellanous . . . 20

6 Human and Material resources **21**

7 Estimates and Gantt diagram **22**

8 Risk management **23**
8.1 Interacting with Open-Source and skill level . . . 23
8.2 Changes in direction LLVM . . . 24
8.3 Programming errors and bugs . . . 24
8.4 Deadlines and delivery of the thesis . . . 24
8.5 Scope creep . . . 24

9 Budget **25**
9.1 Hardware . . . 25
9.2 Software . . . 25
9.3 Human Resources . . . 26
9.4 Indirect Costs . . . 26
9.5 Contingency . . . 28
9.6 Unexpected costs . . . 28
9.7 Final Budget . . . 29
9.8 Control Management . . . 29

10 Sustainability Report **30**
10.1 Economic Dimension . . . 30
10.2 Environmental Dimension . . . 30
10.3 Social Dimension . . . 31
10.4 Sustainability Matrix . . . 31

11 Updates **32**
11.1 Context and Scope . . . 32
11.2 Tasks and Scheduling . . . 32
11.3 Budget . . . 36
11.3.1 Software . . . 36
11.3.2 Hardware . . . 37
11.3.3 Indirect Costs . . . 37
11.3.4 Human Resources and Contingency . . . 38
11.3.5 Updated final budget . . . 38
11.4 Sustainability . . . 39
11.4.1 Matrix Updated . . . 39
11.4.2 Considerations . . . 40

12 Laws, Licenses and Regulations **41**
12.1 Apache 2.0 LLVM License . . . 41
12.2 License of generated files . . . 41
12.3 Law BOE-A-2011-9617 (Science, Tech, and Innovation) . . . 41

13 Context and knowledge integration **43**
13.1 Clang . . . 43
13.1.1 Definition . . . 43
13.2 LLVM IR . . . 45
13.2.1 Definition . . . 45
13.2.2 Usage . . . 48
13.3 MLIR . . . 50
13.3.1 Definition . . . 50
13.3.2 Creating new dialects . . . 51
13.3.3 Dialect conversions . . . 53
13.4 ClangIR . . . 57

14 Practical Work **60**
14.1 OpenMP Specification and coverage . . . 60
14.2 Justification . . . 63
14.2.1 Evolution over time . . . 63
14.3 Implementation . . . 64
14.3.1 Simple directives . . . 64
14.3.2 Directives with captured statements . . . 68
14.3.3 Clause Handling . . . 71
14.3.4 Casts and dialect conversions . . . 75
14.4 Design decisions . . . 77

15 Results **78**
15.1 Verification and testing . . . 78
15.2 Experimentation . . . 78

16 Conclusions **81**
16.1 Reflection . . . 81
16.2 Clang vs GCC . . . 81
16.3 Compiler development . . . 82
16.4 The importance of Open Source . . . 83
16.5 Acknowledging the role of FIB-UPC and BSC . . . 84

17 Future improvements and unimplemented features **85**
17.1 Data sharings . . . 85
17.2 Taskloop . . . 86
17.3 Depend Clause . . . 86

A Appendix **87**
A.1 Environment and building Clang . . . 87
A.2 Compiling files and testing . . . 88
A.3 Contribution Git work-flow . . . 90
A.3.1 LLVM API static example . . . 91
A.4 Experimentation . . . 94
A.4.1 Script . . . 94
A.4.2 Sample code . . . 96

List of Figures

1 An example of performing lexical and syntax analysis on a C conditional, Retrieved from [8] 10
2 Illustration of the utility of an intermediate representation, Retrieved from [11] . . . 11
3 Typical flow of a compiler that uses LLVM from the front-end to the back-end, Retrieved from [13] . . . 12
4 Example of a custom compilation workflow using scf/affine/parallel dialects in addition of polyhedral, Retrieved from [14] . . . 13
5 Example of a C++ compiler Pipeline using ClangIR, made by Roger Ferrer Ibañez . . . 14
6 Management DAG, self elaborated . . . 18

7 Documentation DAG, self elaborated . . . 18
8 Environment DAG, self elaborated . . . 19
9 Research DAG, self elaborated . . . 19
10 Implementation DAG, self elaborated . . . 20
11 Gantt Diagram of the project, self elaborated using the 'pgfgantt' package from LaTeX . . . 23
12 Updated Gantt Diagram of the project, self elaborated using the 'pgfgantt' package from LaTeX . . . 35
13 Core of Clang AST, self-elaborated using the LaTeXpackage "tikz" . . . 44
14 Hierarchical structure of an LLVM program, retrieved from [44] . . . 45
15 Example of Phi function, Retrieved from UTSA CS5363 slides [45] . . . 46
16 Simplified example of the Clang-AST produced using the source file, self-elaborated . . . 47
17 MLIR typical compilation flow and ecosystem, retrieved from [47] . . . 50
18 IR landscape in 2019, retrieved from [15] . . . 57
19 Fork-join model of execution, Retrieved from [51] . . . 60
20 Comparison of compilation times, self-elaborated . . . 79
21 Comparison of execution times, self-elaborated . . . 79
22 Illustration of a Git rebase, Retrieved from [61] . . . 90

List of Tables

1 Task resources (summary), self elaborated . . . 21
2 Summary of project tasks, self elaborated . . . 22
3 Hardware cost and amortization, self elaborated . . . 25
4 Software cost and amortization, self elaborated . . . 26
5 Human Resources Cost (a Factor of 1.35 is used to compute social security taxes), self elaborated . . . 26
6 Project Consumables . . . 28
7 Indirect costs, self elaborated . . . 28
8 Contingency costs, self elaborated . . . 28
9 Unexpected Human Resources Cost, self elaborated . . . 29
10 Final budget sorted, self elaborated . . . 29
11 Sustainability Matrix, self elaborated . . . 31
12 Summary of project tasks updated, self elaborated . . . 34
13 Time Deviation for tasks . . . 36
14 Updated Software cost, self elaborated . . . 37
15 Updated indirect costs, self elaborated . . . 38
16 Updated human resources costs, self elaborated . . . 38
17 Updated contingency costs, self elaborated . . . 38
18 New Final Budget, self elaborated . . . 39
19 Updated sustainability matrix, self elaborated . . . 40

1 Introduction

1.1 The problem

Compilers are undoubtedly crucial and relevant to any process or project involving high-level programming languages, which are the ones we commonly use in our everyday lives, such as C, C++, Java, Rust, Julia, and Fortran. Without these marvelous abstractions, programmers would be forced to code in low-level basic processor instructions. Moreover, programs would need to be written for different processor architectures such as x86, ARM, MIPS. Compilers save programmers from redundant and time-consuming tasks that would otherwise be Sisyphean.

These compilers primarily have one seemingly simple but truly complex task: Given a high-level supported programming language, they must generate a mapping to an equivalent set of instructions in the desired target assembly language. Additionally, compilers should perform analysis and optimization tasks to produce the most efficient machine code possible. They must also provide an easy interface to the end user, who can be completely abstracted from the intricacies of tokenization, parsing, semantic analysis, and be up-to-date with the latest optimization algorithms, register allocation, instruction scheduling, and all the knowledge required to build a useful and optimal compiler. Therefore, not every programmer understands how a compiler works, but they surely use it on a daily basis whenever they need to execute their fancy high-level programs. Instead, this task is delegated to specialized compiler engineers and researchers, who devote themselves to creating, improving, and maintaining these compiler infrastructures.

Existing open-source tools like GCC (GNU project[2]) or Clang (LLVM project[3]) have been utilized for decades by countless programmers to compile their source codes in C/C++. The primary objective of this undergraduate thesis will be to enhance the Clang compiler, which has gained increasing popularity since its inception in 2007. This enhancement focuses on implementing parallelism constructs and directives through OpenMP, a shared memory multiprocessing API[4] that facilitates code parallelization through straightforward directives. This implementation will not be built on top of the actual Clang compiler but instead on the new ClangIR[5] dialect of MLIR (a state-of-the-art intermediate representation under the LLVM umbrella), aimed at changing and improving the traditional compilation workflow. However, it still has a long way to go before it can be widely used. ClangIR is currently under development, so many features, such as OpenMP, are not yet implemented.

More specifically, various OpenMP constructs and directives related to task creation and thread synchronization, such as task, taskwait, taskgroup, critical, barrier to name a few will be implemented. Ultimately, the author hopes this work will serve as a potential pathway for students and newcomers to learn about compilers and the current LLVM compiler ecosystem. This endeavor can provide an accessible entry point for those who wish to contribute to the field, despite the initial challenges it may present.

1.2 Context of the thesis

This project constitutes a segment of the author's final bachelor thesis in Informatics Engineering (GEI) at the Facultat d'Informàtica de Barcelona (FIB), affiliated with the Universitat Politècnica de Catalunya (UPC).

The inception of this work stemmed from a collaborative effort between the author and the Barcelona Supercomputing Center (BSC). The BSC provided invaluable support under the guidance of Roger Ferrer Ibañéz, a Senior Compiler Researcher, offering extensive mentoring and weekly supervision throughout the thesis. Our paths organically converged due to our mutual enthusiasm for compilers.

1.3 Concepts and Terms

Before introducing more intricacies of this problem or to deep dive into specific details, some important compiler related concepts and tools should be addressed, so the reader may understand this work without previous knowledge of compilers.

1.3.1 Compilers

In plain words, a compiler[6] is a tool that recognizes a high level program and assesses that it is valid (Front-end phase), and that later on generates a set of instructions to the selected computer architecture (Back-end phase). We could also, for the sake of efficiency and to avoid redundant work, add a middle-end phase, which comprises a wide range of functionalities such as analysis, optimization, and most importantly generate the intermediate representation (IR) code (But it could also be done in the front-end, it depends on the implementation). Concepts of this pipeline will be explained grouped up by its corresponding phases below, so one can easily digest them in an ordered manner without diving deep into Theory of Computation definitions and concepts.

Front-end This part is the one that recognizes the structure of the source program (which will be treated as a huge string of characters), and it recognizes whether it's a valid C program, for example. We won't enter into too much detail of how this is implemented, but to give some insights, this can be divided in 3 main substages (To be straightforward, other additional stages as preprocessing won't be described), which are:

- Lexical Analysis: This is the action of decomposing the main string of the source program into sub-strings which are usually called tokens, and each programming language has its own tokens, take as an example the following C expression: **"2 * 5"**. A simple tokenization algorithm would produce a similar sequence of tokens: NUMBER("2") MULT("*") NUMBER("5"). Depending on the implementation of the compiler, **The Symbol Table** (Which is a crucial data-structure that contains relevant information about symbols such as functions, global variables...) may be generated on this stage or on the following.

- Syntax Analysis (Parsing): This is the action of recognizing if our now sequence of tokens is a valid construction of the **language grammar** (Referring to a Formal Grammar, which is a finite set of formal rules for generating syntactically correct sentences or meaningful correct sentences, in a theory of computation context, take for instance the C language grammar[7]). For example, C recognizes NUMBER MULT NUMBER as a valid construction but not NUMBER MULT MULT NUMBER, which would result in a syntactic error. This phase also builds what is known as the **Abstract Syntax Tree (AST)**, which is a recursive data-structure which contains different kinds of nodes, corresponding to a wide range of constructions of the language, such as expressions, loops, conditionals, inclusions, functions, variable declarations... The following example1 illustrates both lexing and parsing phases leading to the construction of the **AST**.

- Semantic Analysis: Finally, some checks will be performed on the AST using some special attributes node must have (Which are specific to each language and can be inherited from parent nodes or synthesized attributes which are computed depending on the children nodes) will be checked in order to guarantee that the source program is correct. Consider the expression **int a = b - 42;**, which is going to be recognized as a valid derivation of the C grammar, but it may not be correct due to the variable **b**, which could have not been declared before before **a**, therefore making this statement erroneous. Various checks are performed such as type checking, variable declarations, function overloading, correctness of function calls, analyzing control flow of loops, ensuring that array loads fall within the bounds of that specific array, verifying that pointers are given a reference before usage, and the list goes on and on.

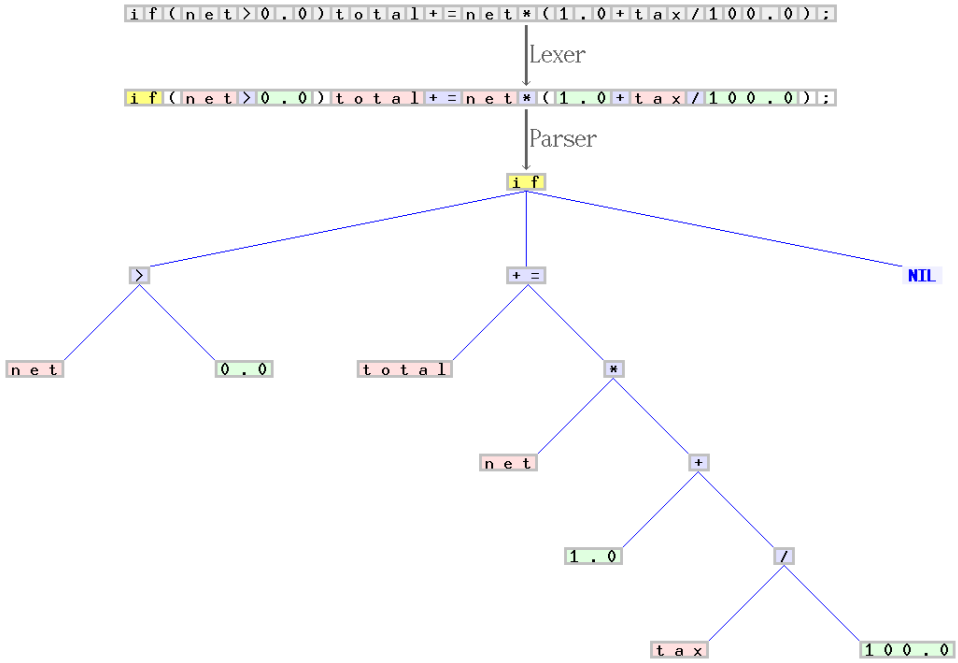

Figure 1: An example of performing lexical and syntax analysis on a C conditional, Retrieved from [8]

Middle-end In this stage the focus falls onto 2 main purposes these are Optimization and Generation of Intermediate Representation code. The following relevant concepts that are needed to understand all the fuzz inside the LLVM/MLIR project and why the ClangIR may enhance the Clang:

- **Lowering:** Is the action of transforming the current representation of the program into a simpler one (Such as lowering the AST representation to Intermediate representation (IR) or even to a machine language such as x86), which usually results in the loss of some information due to the simplicity/generality of the lower language.

- **Intermediate Representation IR**: Intermediate language that acts as a bridge between the front-end language and the target back-end machine language. This means that IR's can be either high language or low language, depending on its purpose. At first, it may seem redundant, but this helps to avoid creating a compiler for every single combination of high level language and target assembly languages. Consider that we have N languages and M assembly languages to translate from, then we would need $O(N*M)$ compilers (2) in order to achieve this. To avoid this extreme redundancy, and

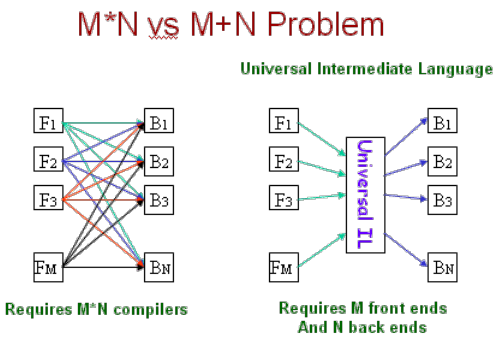

Figure 2: Illustration of the utility of an intermediate representation, Retrieved from [11]

also avoiding a compiler engineer to have to write M compilers for his own language (one for each assembly), universal intermediate representations are used, which reduce the number of compilers down to N front-end to IR, and M IR to back-end compilers, thus $O(N + M)$ and we would also have the additional advantage that once the IR supports translation to the M machine languages (Which is the aim of IR's such as LLVM IR), then creating a new programming language (Like Mojo for instance [9]) with support to every single machine language would require to program a single compiler that implements its own front-end and the IR code generation for this hypothetical IR.

This is why optimization is often done in this stage and not on the frontend, for the sake of not reinventing the wheel several times, just a single implementation of these generic optimization algorithms is needed, targeting the intermediate representation. Optimization algorithms are on its own a whole complex research area therefore they will be out of the scope of this work, for more information take a close look at [10].

Back-end Lastly, after recognizing the main patterns of the source program, lowering the AST down to an intermediate representation and further analysis and optimization, now is ready for the last translation, to a specific machine language.

This may seem straightforward at first glance, but it has its own intricacies and processes, and on top of that there's now the opportunity of applying machine specific optimizations. For example, AMD programmers, being full aware of its own architecture, could write optimizations aimed at graphic cards to enhance texture cache hit rates or to enforce SIMD instructions...

There's a series of processes the IR code should undertake in order to generate the correct instructions: , machine instruction selection, register allocation, instruction scheduling, where optimizations of functions, vectorization, and target dependent improvements are performed along. Obviously they are also out of the scope of this work, so for more detailed information, the author would advise the reader to check out both resources [10][12].

1.3.2 LLVM Project

Low-level virtual machine (LLVM) is a wide umbrella project which comprises a wider range of compiler projects such as LLVM IR, Clang, Flang, OpenMP, MLIR to name the most relevant to this thesis.

OpenMP OpenMP, integrated into the LLVM project, offers a user-friendly API based on C/C++/Fortran directives, facilitating the utilization of parallelism in programming tasks. It significantly reduces the complexity associated with parallel programming by providing simple directives that specify parallel regions and tasks.

The primary focus of this project is to leverage explicit tasking capabilities provided by OpenMP directives. These directives enable developers to define tasks that can be executed in parallel, enhancing program performance and scalability. The current version of OpenMP supported by Clang and soon to be supported by ClangIR is 4.5.

LLVM IR The LLVM Intermediate Representation (IR) serves as the foundational core of LLVM, offering a comprehensive infrastructure for applying optimizations and implementing back-ends. This language-agnostic intermediate representation acts as a central hub in the compilation process, often serving as the primary and last IR before transitioning to the compiler back-end.

This IR facilitates the optimization and transformation of code across various stages of compilation, enabling developers to leverage a wide range of optimization techniques and targeting diverse hardware architectures (3). Its flexibility and extensibility make it a key enabler for creating efficient and high-performance compilation workflows.

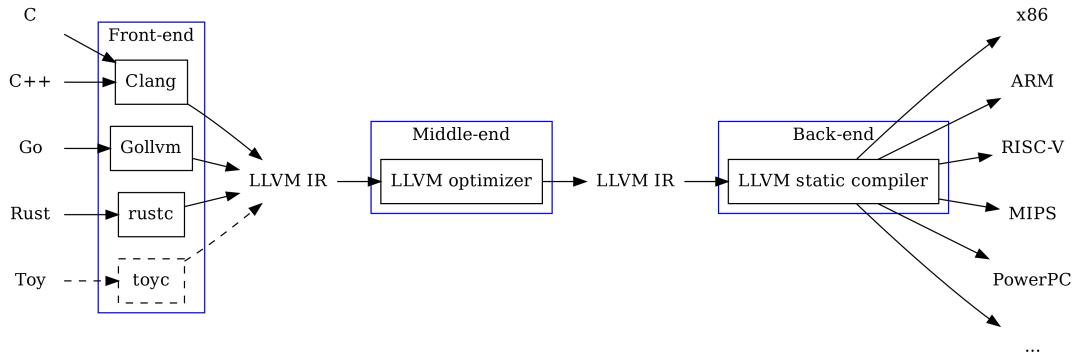

Figure 3: Typical flow of a compiler that uses LLVM from the front-end to the back-end, Retrieved from [13]

MLIR MLIR (Multi-Level Intermediate Representation) is a novel infrastructure developed under the LLVM umbrella project aimed to facilitate and abstract the interactions between multiple intermediate representations. It is designed to facilitate the construction of high-performance domain-specific languages and compilers. MLIR introduces a flexible, multi-level intermediate representation that enables efficient representation and transformation of code across various stages of compilation. Originally conceived for optimizing common machine learning code, GPU/DPU computation, or other domain-specific

code, MLIR serves as a versatile tool for optimizing and enhancing compiler workflows.

At its core, MLIR provides a framework for building compiler pipelines with an emphasis on modularity, reusability, and optimization. It offers a unified representation for diverse compiler tasks, ranging from high-level program analysis and transformation to low-level code generation and optimization.

One of the key features of MLIR is its support for "Dialects", which are sets of specific operations, attributes, and types that collectively define a new intermediate representation. There is a wide range of dialects, each tailored to a specific niche, including **amdgpu**, **omp**, **nvgpu**, **tensor**, **arith**,**linalg**, **x86vector**, and **ClangIR**. By leveraging multiple intermediate representations (4), compilation flows can transition from higher levels of abstraction to lower levels, enabling the application to benefit from advanced and multiple optimization passes that may not be feasible using traditional LLVM IR alone. This is particularly valuable as it retains the wealth of the program information throughout the lowering process, information such as complex types, operations, locations and high-order abstractions.

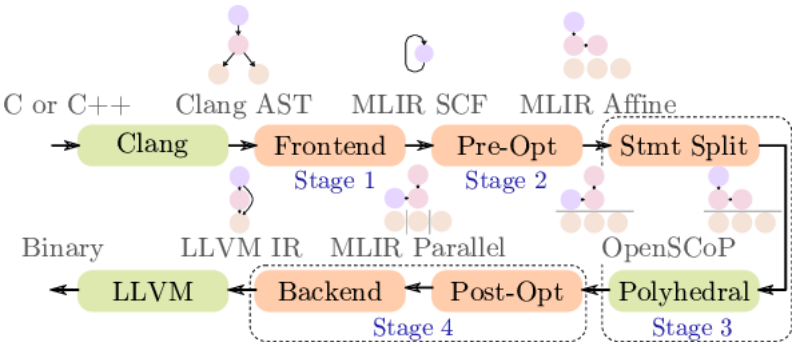

Figure 4: Example of a custom compilation workflow using scf/affine/parallel dialects in addition of polyhedral, Retrieved from [14]

ClangIR CIR, short for Clang Intermediate Representation, serves as a language-specific dialect within MLIR, similar to MIR (Rust), SIL (Swift), and Julia IR (Julia). While these IRs leverage MLIR as a foundation, they are not technically MLIR dialects themselves [15]. CIR retains pertinent information from the Clang AST post-lowering. Consequently, it facilitates the creation of more domain-specific and implementable optimizations due to how high-level this intermediate representation is. It's worth noting that while this approach sacrifices some generality, it enables optimizations in C/C++ that can lead to substantial speed improvements, particularly in systems heavily reliant on these languages.

However, this increased flexibility comes at the cost of a larger and more complex compiler. This trade-off is particularly evident with features like C++20 coroutines, which, while technically implemented in low-level LLVM IR, suffered from redundancy and complexity. CIR higher-level abstractions can potentially streamline such implementations. The novel compilation flow is illustrated in Figure (5).

1.4 Stakeholders and Reach

The significance of the Clang compiler inherently lends itself to an extensive reach. As ClangIR gets more prevalent in the near future, its potential impact on program correctness and efficiency could be

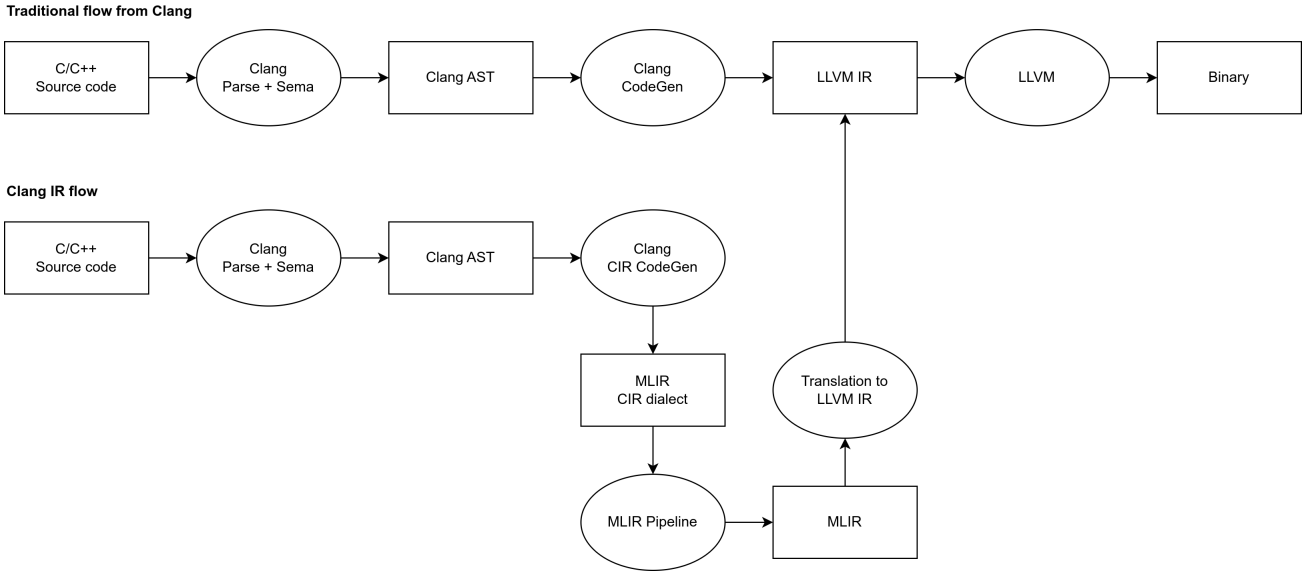

Figure 5: Example of a C++ compiler Pipeline using ClangIR, made by Roger Ferrer Ibañez

considerable, possibly even surpassing that of GCC. The substantial community of contributors and vast number of end-users utilizing this compiler underscore the relevance of this work and its potential impact on society, even if it is tiny. The key stakeholders involved in this endeavor include:

- Researchers: This project contributes to the ClangIR subproject of LLVM, attracting attention from various PhD candidates and researchers who stand to benefit from an established framework.
- LLVM Community: The ambitious ClangIR project will receive additional support, enabling further extension and development by other contributors within the LLVM community.
- Programmers (End-users): The flow of compilation in Clang, particularly for C/C++/Objective-C/Objective-C++ programmers utilizing OpenMP, could see improvements in execution time.
- Private Companies: Large corporations such as Google, Meta, AMD, Intel, Nvidia, ARM, Qualcomm, Samsung, and Xiaomi, which are devoted users and contributors of MLIR and LLVM, stand to benefit from additional contributions and support.

2 Justification

The justification for the relevance of this project revolves around two points: the author's decision to implement OpenMP in ClangIR (and not other features, potentially) and the consideration of alternative approaches and previous solutions. From the outset, the project aimed to improve the existing Clang infrastructure, with ClangIR offering a promising way for enhancing the aforementioned. However, modifying Clang itself or exploring other LLVM subprojects like Polly were significant challenges, especially given the author's limited experience and knowledge, given the limited time-frame of the project.

Initially, alternative options were explored instead of integrating OpenMP into ClangIR, including the consideration of OpenACC or OpenCL. However, feedback from ClangIR project leaders suggested that these alternatives might be too complex for newcomers to effectively implement within the project's timeline. While numerous contributions could be made to ClangIR, the appeal of OpenMP was great, its relevance is high, and it was at the reach of the author skills.

It's crucial to evaluate already existing OpenMP implementations to understand the importance of this thesis. Projects such as Clang, Flang, or GCC repositories provide implementations directly into the compilers. While these serve as references, coding such implementations can be time-consuming and complex (reinventing OpenMP yet another time). Instead, using MLIR OMP dialect poses an easier and more compelling approach, using already existing infrastructure to implement OpenMP easier.

Moreover, the eventual integration of ClangIR into Clang will need the implementation of OpenMP to ensure full compatibility with the existing features. Therefore, these implementations are not merely optional and "nice to have", but prerequisites for the successful integration of ClangIR into the Clang ecosystem.

3 Scope

3.1 Objectives and Requirements

3.1.1 Main Objectives

As previously mentioned, the primary focus will be on two central objectives. The first is to give the reader an invaluable resource for understanding modern compiler development. The second will be the implementation of OpenMP inside ClangIR, which is almost none existent at the moment, only the most basic form of the basic `#pragma omp parallel` has been implemented in January of this year [16], so there's a vast amount of features that are yet to be implemented.

- Implementation of the `Task`, `Taskgroup`, `Taskwait` and `Taskyield` directives in the ClangIR dialect to enable comprehensive task management.
- Integration of Synchronization constructs `Barrier`, `Critical` and thread execution directives `Single` and `Master`
- Consideration of the implementation of the `#pragma omp taskloop` directive in ClangIR, although it remains an optional feature.

3.2 Requirements

To measure the success of this project, we establish additional requirements categorized into two sets: functional and non-functional requirements.

3.2.1 Functional Requirements

- **Behavior**: The implementation of the OpenMP API in ClangIR should exhibit performance comparable to that achieved using the OpenMP API in the conventional Clang compilation flow.

3.2.2 Non-functional Requirements

- **Adaptability**: The proposed solution must demonstrate adaptability within the rapidly evolving Clang project, anticipating potential changes from its current state to its integration into the broader LLVM project. Subsequent versions should require minimal to no modifications.
- **Complexity**: The solution's complexity should be minimized, limiting the introduction of additional IR instructions to streamline system complexity and optimize compilation and execution efficiency.
- **Correctness**: Ensuring the correctness of the generated code is paramount. Unit tests will be employed to validate the code and mitigate the risk of potential bugs.

3.3 Obstacles and Risks

This work is not immune to possible setbacks and risks that may arise during its development and deployment. Nonetheless, a list of the most concerning issues that could potentially arise includes:

- Collaboration overhead: Given that the project is open-source and adheres to strict style and pattern guidelines, there is a possibility that the author may overlook some of these guidelines due to my newness to the project. This oversight could result in code refactoring and wasted time in getting Git pull requests approved by project leaders.
- Poor Documentation: The ClangIR project is relatively novel and lacks in-depth documentation, tutorials, and user guides. As a result, extra time must be dedicated to self-learning and understanding the project's intricacies.
- Author's Experience: Before embarking on this thesis, the author had no prior experience with LLVM/MLIR/ClangIR tools or working on modern compiler projects. Therefore, a significant portion of this work involves familiarizing oneself with the state-of-the-art infrastructure, algorithms, and decisions made within these projects, which are not arbitrary. These knowledge gaps will likely be addressed through a combination of articles, documentation, mentorship, and other informational sources.
- Complexity of Testing: the nature of testing a compiler is often complex, so unit tests should be written to enforce expected behavior. In this paradigm, bugs are hard to catch, so specialized LLVM debuggers and meticulous methodologies will be employed.
- Unexpected Challenges: Unforeseen personal issues or changes in the direction of the ClangIR project could potentially arise without warning.

4 Methodology and Rigor

Various development methodologies are prevalent in the industry, but for this project, the author has opted for a simplified version of Agile to ensure incremental and efficient progress. Sprints will be employed, with weekly face-to-face catch-up meetings between the author and the tutors to minimize communication overhead.

Each week, the sub-tasks of the current sprint will be outlined in a dedicated notebook. Tasks unfinished at the end of the sprint will either be reassigned to the following week or discarded. This approach avoids elaborate Kanban boards and agile software, focusing instead on practical, handwritten notes and sketches for efficient task management.

GitHub serves as an indispensable tool in this process, storing the repositories for both the LLVM project[17] and the forked version containing ClangIR[18]. Beyond its necessity, GitHub facilitates organized progress tracking through commits and pull requests.

To conclude, documentation will be authored in LaTeX using the web-app Overleaf[19], which provides a user-friendly yet comprehensive environment for LaTeX coding and writing.

5 Description of the Tasks

To achieve a satisfactory long term planning of this project, which takes place between the last two weeks of January and the third week of June (Included), a division of sub-tasks must be established from the main tasks to keep track. From the following scheduling we can infer that the thesis will approximately take up to 730h until its completion (Detailed in table 2), which would require a dedication of 30h/week (Note that workload may not be evenly distributed in certain weeks like the third one 7. After the revision of this document, its important to note that this definition of tasks is outdated due to the changes in direction that the project has suffered.

5.1 Management

Firstly, this group will encompass all the tasks related to both the management of this thesis. This was achieved using a computer, a notebook and an online LaTeX IDE, Overleaf[19]. Therefore, we define the following Directed Acyclic graph (DAG) which defines the requisites of each minor endeavor6.

- **M1 - Context and scope**: The definition of the introduction, context, stakeholders, and scope has an expected duration of 25 hours.
- **M2 - Planning and scheduling**: The creation of the project plan might take up to 20 hours.
- **M3 - Budget and sustainability**: Estimating a budget and the sustainability matrix of this project will take up to 18 hours.
- **M4 - Revised document**: The revision and incorporation of feedback in the three documents and mixing them together could potentially take up to 10 hours.

- **M5 - Weekly meetings**: Each week a meeting will be hosted with the tutors to assess the author's work which take up to 1 hour, therefore 22 hours.

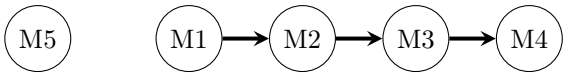

Figure 6: Management DAG, self elaborated

5.2 Documentation

Another relevant group of tasks to define is the documentation tasks, which range from the formulation of the final document of this thesis and creating the presentation to writing a comprehensive documentation of the code the author will be building. Again, the only tools needed to achieve this are a computer and Overleaf, which also is capable of producing high quality slides. The following DAG defines the order in which tasks must be finished7.

- **D1 - Code documentation**: The documentation of the code merged into ClangIR repository could even take up to 30 hours, but it might be less.
- **D2 - Final thesis document**: The elaboration of said document and the posterior revisions will take around 60 hours to complete
- **D3 - Final presentation**: Elaborating this document using LaTeX could take up to 40 hours.

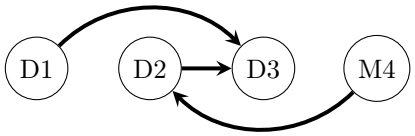

Figure 7: Documentation DAG, self elaborated

5.3 Environment

To set up the computer environment and an efficient and simple workplace needed to contribute to the ClangIR project, the following group of tasks with its corresponding DAG8 is defined. The only tool required is a computer and some additional software to build and compile C/C++ projects, such as CMAKE(Build creator)[20] and NINJA(Build system)[21].

- **E1 - Understanding CMAKE and NINJA**: In order to get familiar with these sophisticated build systems that are the standard in the LLVM project, the author will engage with a series of documentation and tutorials that will take up to 5 hours.
- **E2 - Compiling the LLVM/ClangIR project and setting up the environment**: Multiple builds of the LLVM project until the ideal environment was established. Moreover, the environment variables, the partition management (LLVM compiled in debug mode takes up to 100 GB) and other extensions on Visual Studio Code were explored, which took no more than 10 hours.

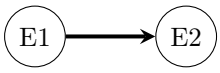

Figure 8: Environment DAG, self elaborated

5.4 Research

It's absolutely required, to achieve a successful implementation, the research and getting in touch with tools and concepts which were foreign to the author at the start of this journey. The aim of these tasks is to fill gaps in knowledge and also to understand the huge infrastructure and ecosystem that are LLVM and Clang. The following DAG9 represent the tasks and their dependencies:

- **R1 - Delve into LLVM framework**: Firstly, familiarization with LLVM is needed, not only getting to know what it is in a general sense, but also reading hands-on books (The author suggests [12]), going through the entire Kaleidoscope tutorial[22] step by step. Furthermore, reading the LLVM documentation and watching the LLVM introduction course by Compilers Lab [23] will be of great help. This task is estimated to take around 100 hours.
- **R2 - Becoming familiar with MLIR**: Secondly, a comprehensive understanding of MLIR is indispensable, given that ClangIR operates as a dialect within the MLIR ecosystem. This entails gaining not only contextual insights, but also a deep understanding of how this tool reshapes the compilation process. Due to limited resources, newcomers may find the MLIR toy tutorial[24] useful. Additionally, insights from Lei Zhang's blog on MLIR[25] provided plenty of context. This task would require approximately 45 hours.
- **R3 - Reviewing OpenMP**: Since the author already had some previous knowledge using OpenMP API, a brief refresher is required to recall the intricacies of the task directives and the wide range of options they offer. This should take no more than 5 hours.
- **R4 - Getting to know ClangIR**: Finally, a wide understanding and familiarity with ClangIR is needed to avoid redundant work. This is specially important, knowing that ClangIR reuses several already built structures, like other dialects of MLIR. This will approximately take up to 30 hours.

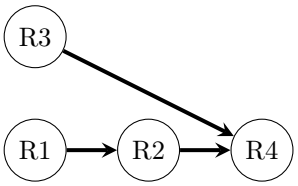

Figure 9: Research DAG, self elaborated

5.5 Implementation

Last but not least, this set of tasks consist of the coding and implementation part of this thesis, targeted to implement several OpenMP directives into the fresh new ClangIR. The duration of each task is merely an estimation, due to numerous factors like code revision (Git pull requests), the complexity of the algorithms, the lack of experience in this field among others. Therefore, the expected completion times may increase significantly, but these figures may serve as lower bound. The following DAG10 represents the dependencies among tasks, note that in actuality there's no real dependency between those 4 tasks.

- **I1 - Initial implementation of taskwait**: Even if this directive may seem the easiest of them all, the expected time to finish this task will encompass 40 hours, considering that the author has to gain familiarity with the code and pull request process.
- **I2 - Development of the task directive**: Given the complexity inherent in the directive, including multiple constructs, it is anticipated that various underlying intricacies will emerge during the development process. Therefore, a total of 100 hours will be allocated to this task.
- **I3 - Implementation of taskgroup**: After implementing the aforementioned directives, task group won't probably take more than 40 hours after all the gathered experience.
- **I4 - Optional implementation of taskloop**: Finally, if all the previous tasks are completed in less time than the expected, the development of taskloop could be considered, but due to its complex nature, it could take up to 70 hours of work to implement.

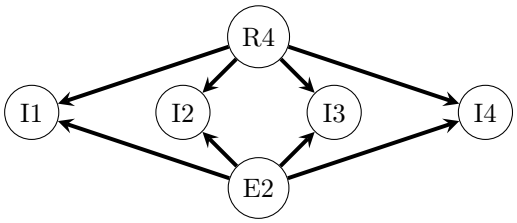

Figure 10: Implementation DAG, self elaborated

5.6 Miscellanous

Not a single project can escape from the harsh realities of uncertainty, from unpredictable issues that may arise to even changes in direction of the ClangIR project that would lead to an impact to the final product. For example, it could be the case that the ClangIR project leaders proposed major changes that could have an impact on my development or even that LLVM community decided to not go further with ClangIR for some unexpected reason. It should also be taken into account that there is some uncertainty surrounding the forecast time required to finish the project. Not only that, but also issues related to the author's personal life could also influence and set back the completion of this project by the stipulated deadline.

6 Human and Material resources

In one hand, the material resources needed are none other than a powerful computer (Laptop in this case), the only tool needed to code, execute, compile and redact the thesis and its final presentation. Therefore, some software will be needed, such as the programming IDE Visual Studio how Code, Overleaf[19] and GitHub to name a few.

On the other hand, as far as human resources are concerned, the author will require the assistance of 3 individuals: the director of this thesis José Miguel Rivero Almeida (**DIR**), the co-supervisor, Roger Ferrer Ibañez (**COS**); and finally the GEP tutor assigned to me, Xavier Bellés (**GEP**). Rivero will assist me on theoretical gaps of knowledge around compilers, while Xavier will revise the quality of my management work to enhance it. Roger will guide me on both sides, mentoring me through the implementation of ClangIR thanks to his expertise in the field and also helping me schedule the project with small incremental deadlines.

Moreover, the completion of the thesis will require me to assume a wide range of roles during the different phases of such:

- **Project Manager**: Before the start of any significant endeavor, it is essential to arrange exactly which tasks are going to be performed, within which time frame and all the requirements associated. This hopes to establish a schedule and a framework to recognize success, failure, or common delays in the project, which is critical in any professional project. This role will tackle tasks M1,M2,M3,M4 and D2,D3.
- **Compiler developer/engineer**: this niche role will be assumed in order to accomplish tasks E2,I1,I2,I3,I4 which involve creating the solutions and setting up the workplace. To address the current lack of expertise tasks E1,R1,R2,R3,R4 will be performed, as any intern or junior engineer would have accomplished to success in their careers.

Tasks	Corresponding Role	M. Resources	H. Resources
R1,R2,R3,R4,E1,E2,I1,I2,I3,I4, D1	Compiler Engineer	Laptop	**DIR, COS**
M1, M2, M3, M4, M5, D2,D3	Project Manager	Laptop	**GEP, COS**

Table 1: Task resources (summary), self elaborated

7 Estimates and Gantt diagram

In this section a time estimation table2 and a GANTT diagram[26] of how these tasks will be scheduled7 are put forward in order to justify how the tasks will be organized and balanced, to not exceed the given time frame (The project was set on motion on the penultimate week of January and its completion should not exceed the third week of June).

Task Description	**Acronym**	**Duration**	**Dependent tasks**	**Requirements**	**Human Resources**
Context and scope	M1	25 hours	-	*Overleaf*, PC	GEP
Project planning	M2	20 hours	M1	*Overleaf*, PC	GEP
Budget and sustainability	M3	18 hours	M2	*Overleaf*, PC	GEP
Revised document	M4	10 hours	M3	*Overleaf*, PC	GEP
Weekly meetings	M5	22 hours	-	-	DIR, COS
Total Management Hours: 95					
Code documentation	D1	30 hours	-	PC, VsCode, Github	-
Final thesis document	D2	60 hours	M4	PC, Overleaf	DIR
Final presentation	D3	40 hours	D1, D2	PC, Overleaf	DIR
Total Documentation Hours: 130					
Understanding CMAKE/NINJA	E1	5 hours	-	PC, VsCode	-
Compiling LLVM/ClangIR and setting up the env.	E2	10 hours	E1	PC, VsCode	COS
Total Environment Hours: 15					
Delve into LLVM framework	R1	100 hours	-	PC, VsCode, [23], [22], [12], [3]	DIR,COS
Becoming familiar with MLIR	R2	45 hours	R1	PC, VsCode, [24], [25], [27]	-
Reviewing OpenMP	R3	5 hours	-	PC, VsCode, [28]	-
Getting to know ClangIR	R4	30 hours	R2,R3	PC, VsCode, [5], [18]	-
Total Research Hours: 180 hours					
Initial implementation of taskwait	I1	40 hours	R4, E2	PC, VsCode, Github, [5]	COS
Development of the task directive	I2	100 hours	R4,E2	PC, VsCode, Github, [5]	COS
Implementation of taskgroup	I3	40 hours	R4,E2	PC, VsCode, Github, [5]	COS
Optional implementation of taskloop	I4	70 hours	R4,E2	PC, VsCode, Github, [5]	COS
Total Implementation Hours: 250 hours					
Miscellaneous	MISC	50 hours	-	-	Unexpected
Total: 730 hours					

Table 2: Summary of project tasks, self elaborated

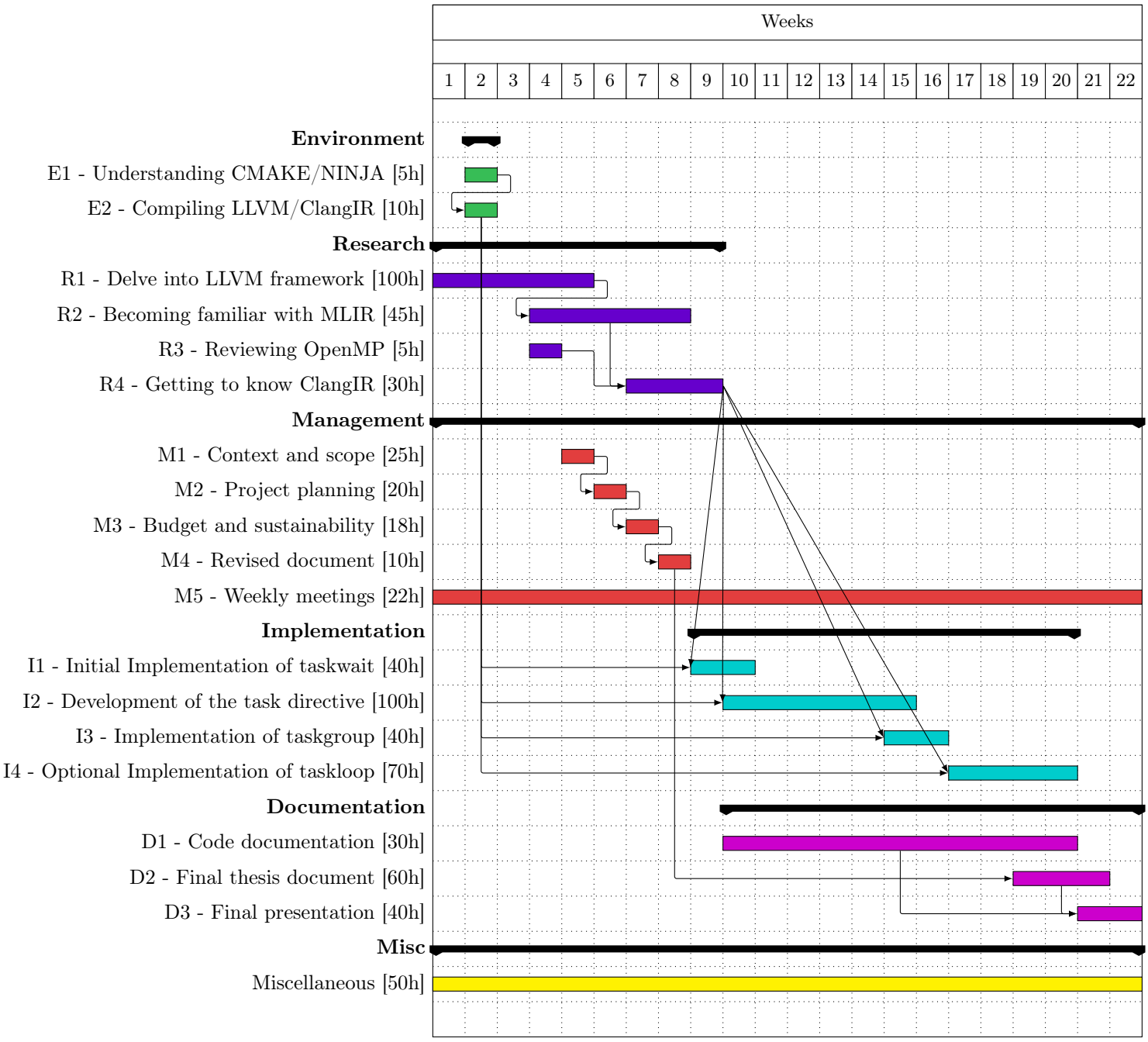

Figure 11: Gantt Diagram of the project, self elaborated using the 'pgfgantt' package from LaTeX

8 Risk management

8.1 Interacting with Open-Source and skill level

This collaboration between the author and the ClangIR community has one potential risk associated with the pull requests, which may delay or even reject the features that the author will implement. This could lead to serious delays which could jeopardize part of what makes this project shine (And also lead to a delayed implementation of the optional taskloop which might not be ready to showcase at the end of the thesis, but it is already expected), which is contributing to a meaningful compiler project. To

be fair, the community seems open to newcomers, so the risk assumed is moderate, however the steep learning curve of this project discourages the leaders to accept any work easily. To overcome this obstacle, the author's counts with the seasoned expertise of Roger and also the research that will be done in order to greatly understand the intricacies of the project to ensure certain quality on the implementations.

As a contingency to ensure that the implementation of the taskloop directive is met on time, extra budget and time-frame is allocated (In addition to the time allocated on task Miscellaneous), described below 8. However, these delays are not critical for development purposes but only on the goal of contributing to the aforementioned community.

8.2 Changes in direction LLVM

Again, one possible obstacle that could completely jeopardize the project would be if the ClangIR project was abandoned and discarded by the LLVM community, leading to a dead road, so the risk is hypothetically immense. This is extremely unlikely to happen due to the great necessity of this dialect to improve and further optimize the last versions of the C based languages and the vast amount of resources being allocated from different companies such as Meta, Google, Arm...

As an individual, the author has limited influence over the decisions made by project leaders within companies. However, the author has chosen to focus on this particular project due to its growing popularity and the positive feedback it has received from the broader community [29].

8.3 Programming errors and bugs

One potential concern could arise from bugs within the already existing LLVM software and errors that the author may introduce. The former is minimal given LLVM's robust foundation established by thousands of programmers over the past two decades, while the latter will be addressed by implementing unit tests with each task implementation.

As a contingency measure, tasks have a loose time frame to face this probable issues that will arise during the development of these features. Not only that, but to minimize the impact of the aforementioned, it is already planned that the task R1 involves getting familiar and hands-on experience with the LLDB (LLVM debugger).

8.4 Deadlines and delivery of the thesis

Moreover, the looming June deadline presents another potential challenge, given its critical importance for the progress and evaluation of the thesis. In anticipation of this, generous time buffers have been incorporated into nearly every task, with additional time provisions factored into the budget (see Table 8). These measures aim to minimize the associated risks.

8.5 Scope creep

Finally, as the author delves deeper into the project (specially considering the author's tendencies), there may be a temptation to add extra features which were originally out of the scope of this project. Despite the positive effects of producing additional results on the same time frame, this can lead to serious delays

and deviations. The risk is moderate to high.

To address this issue, extensive project management has been conducted to ensure that at least a core set of relevant features are implemented (taskwait, taskgroup and task) and additional features will be considered if appropriate. Furthermore, the flexible time frames allocated for each task, the possibility of utilizing time from the MISC task and the consideration of contingencies 8, provide the opportunity to explore other potential features (taskyield, barrier, critical, section, master, flush...).

9 Budget

In the following sections the different costs of the project will be unraveled, taking a wide range of factors into account such as economic costs (Both material and human), indirect costs, unexpected costs... and the most relevant, the environmental cost. However, this is in fact a hard task to estimate, therefore some assumptions will be made.

9.1 Hardware

The device that will be used for every single tasks such as research, implementation, documentation and management, will be a laptop (Asus ZenBook UX425UA_UM425UA[30] which had a cost of 799€). The following table 3 will illustrate the amortization of this device (Considering a useful life of 5 years and an average dedication of 30 hours per week):

$$Amortization = 799€ \cdot \frac{1\text{useful life}}{5years} \cdot \frac{1year}{52weeks} \cdot \frac{1week}{30hours} \cdot 730 = 74.78€$$

Device	Useful Years	Price	Amortization
Asus Zenbook UM425UA	5 years	799€	74.78€

Table 3: Hardware cost and amortization, self elaborated

9.2 Software

A range of software tools will be employed to accomplish this project. Not only the author will be using the aforementioned Overleaf, Vs Code and GitHub, but also other relevant software must be considered, such as the operating system, which is Ubuntu OS 22.04 LTS, the compiler used which will undoubtedly be Clang (Stable version 14), a browser like Google Chrome and occasionally the text editor VIM. All the aforementioned are Open-Source except for GitHub and Google Chrome (Which are free nonetheless). The amortization of the aformentioned software is summarized in the following table (4):

Product	Cost	Amortization
Ubuntu 22.04 LTS	0€	0€
LaTeX	0€	0€
VIM	0€	0€
Clang	0€	0€
GitHub	0€	0€
Chrome	0€	0€
Visual Studio Code	0€	0€
Total	**0€**	**0€**

Table 4: Software cost and amortization, self elaborated

9.3 Human Resources

The largest cost will indeed come from the human resources allocated, which will entirely rely on the roles assumed by the author. We will over simplify this estimation by using the average yearly gross data for each role in a 0-1 year range given by Glassdoor[31](Previous work experience of the author won't be considered). Another assumption is that given that compiler engineer is a niche inside the broader computer engineering field and almost no public information is available of the few compiler developers that are located in Spain (Barcelona to be specific), software engineering salaries [32] will be used instead, but it should be noted that this assumption probably makes seem the project cheaper than in reality (Considering how well paid are junior compiler developers in countries like USA or Germany[33] or even this reference from BSC [34]). The following formula has been used to compute the hourly salary:

$$HourlySalary = YearlySalary \cdot \frac{1year}{12months} \cdot \frac{1month}{20\text{days}} \cdot \frac{1day}{8hours}$$

It's worth noting that both Rivero and Bellés will assist me on the various parts of the thesis, but their contributions are assumed to be part of their job as teachers affiliated with Universitat Politècnica de Catalunya, not only that, but the number of hours would be hard to infer beforehand. A similar situation regards Roger, who will provide me an invaluable guidance and uncountable efforts voluntarily. This stems from both our shared passion for the subject, and also that it is in his best interest that the LLVM ecosystem is improved. If this wasn't the case, the efforts put by any of the aforementioned should be tracked and added to the budget of the project, which would greatly increase the cost of the project. Lastly, one should keep in mind that probably the assistance of those 3 individuals could be needed in the task so called "Miscellaneous", although it is not predictable.

Role	Salary	Expected Hours	Total Cost	Total + Social Security
Software Engineer	18.23€/hour (35K€/year)[32]	535 hours	9753.05	13166.62€
Project Manager	20.31€/hour (39K€/year)[35]	195 hours	3960.94€	5347.27€
Total	-	**730 hours**	13713.99€	**18513.89€**

Table 5: Human Resources Cost (a Factor of 1.35 is used to compute social security taxes), self elaborated

9.4 Indirect Costs

Costs such as location, electricity, internet connectivity and training costs are often overlooked factors that will be discussed in this section.

- **Electricity**: The laptop [36] used to make this thesis, used for 730 hours, has an estimated battery life of up to 16 hours, recharge times of approximately 2 hours and 67 Wh of capacity, which taking into account a typical joule effect of 10% would need 73.7W to be fully charged. Therefore, computing the number of recharges and Watts consumed would lead us to the total cost (Knowing that mean electricity price as today, March 3, 2024 is 0.0489€/kWh[37]):

$$730 \text{ hours} \cdot \frac{1recharge}{16hours} = 45.625 \approx 46 \text{ recharge}$$

$$46 \text{ recharge} \cdot \frac{2hours}{1recharge} = 92 \text{ hours}$$

$$46 \text{ recharge} \cdot \frac{73.7W}{1recharge} \cdot \frac{1kW}{1000W} = 3.39kW$$

$$ElectricityCosts = 92h \cdot 3.39kW \cdot \frac{0.0489€}{1kWh} = 15.25€$$

  Its worth noting that such laptop will be used for several other simultaneous projects or subjects in which the author is involved, so one must remember that after all, this is a mere estimation.

- **Connectivity**: The author's internet connection is mainly established by public Wi-Fi's (Which are indeed free) or through its own router, which has a cost of 30€/month. The assumption that connection is needed in 80% of the project, will be made, since there's little to no task that doesn't use internet at all, except for the implementation tasks or maybe reading the LLVM essentials handbook. Therefore, the costs associated with the project (the fraction) will raise up to:

$$ConnectivityCosts = \frac{30€}{1month} \cdot \frac{1month}{30days} \cdot \frac{1day}{24hours} \cdot 730 \cdot 0.8 \text{ hours} = 24.33€$$

- **Training**: The training materials used to expand the author knowledge of the subject must be considered since some of them had an economic cost which shouldn't be overlooked. The vast majority of resources could be found with no additional costs, such as the dragon book [10], the LLVM[22] or MLIR[24] tutorials, the ClangIR documentation[5] or even the LLVM course by Compilers Lab[23].However, the author spent the total of 20€ in buying one key resource to its self-taught journey, which was LLVM Essentials[12].

- **Location**: The author's rent should be considered inside the indirect costs of this thesis (Approximately the 33% of the area will be employed for this purpose), since his apartment will be employed as an office for at least, half of the time. However, the rest of the time, local libraries and UPC infrastructure will be employed as a work place, with no additional expenses. Thus:

$$LocationCosts = \frac{255€}{1month} \cdot \frac{1month}{30days} \cdot \frac{1day}{24hours} \cdot 730 \cdot 0.5 \cdot 0.33 \text{ hours} = 43.09€$$

- **Other consumables**: To put an end to this list, other consumables such as pens, notebooks, and other office supplies must be considered. The table (6) breaks down which items, prices, and quantities for every single consumable.

Consumable	Price (€)	Quantity	Total Price (€)
BIC Pen	0.46	2	0.92
Notebloc Notebook	5.20	1	5.20
FinoCam Journal	11.50	1	11.50
Total Price			**17.62**

Table 6: Project Consumables

To conclude, it is also wise to notice that in case of a general outage of electricity in my neighborhood, the author could always just take advantage of the free power and Wi-Fi offered by Universitat Politècnica de Catalunya and the local library.

Resource	Price
Location	43.09€
Internet	24.33€
Training	20.00€
Consumables	17.62€
Electricity	15.25€
Office	0.00€
Total	**120.29€**

Table 7: Indirect costs, self elaborated

9.5 Contingency

Part of the budget will be destined to cover possible unfortunate events that may occur and delay the project, to ensure its realization. Thus, 8% of additional resources will be allocated to mitigate possible obstacles that may arise.

Source	Price (€)	Percentage	Cost (€)
Hardware	74.78	10%	7.48€
Human Resource	18513.89	10%	1851.39€
Indirect Costs	120.29	10%	12.02€
Total	-	-	**1870.89€**

Table 8: Contingency costs, self elaborated

9.6 Unexpected costs

Finally, similar to contingency costs, a small fraction of the budget should be targeted towards covering possible incidents, such miscalculations of the expected time of completion of the 4 implementations or in the redaction of the thesis or its presentation that would lead to working overtime to polish details. This is mitigated making use of the unexpected costs (9).

Position	Extra Hours	Probability	Cost	Cost with Social Security
Software Engineer	30 hours	20%	546.9€	738.32€
Project Manager	10 hours	5%	203.1€	274.19€
Total	**40 hours**	-	-	**1012.51€**

Table 9: Unexpected Human Resources Cost, self elaborated

Moreover, bearing in mind that the laptop that will be used is already 2 years old, a failure could happen (low probability of 15%) which would require contacting the repair service of ASUS, which could lead into a cost of 65€ (mean). One can infer then that **the sum amounts to 1077.51€**.

9.7 Final Budget

To put the budget in a nutshell, the following table will illustrate and summarize all the elements reviewed so far (10). The whole sum will give a detailed understanding of the project overall costs and how this is distributed among different factors.

In summary,a consolidation of the budget elements that have been reviewed so far will be given. This will give us a comprehensive understanding of the project's overall expenses and how the budget is distributed among its various aspects.

Source	Cost
Human Resources	18513.89€
Contingency	1877.76
Unexpected Costs	1077.51€
Indirect Costs	120.29€
Hardware	74.78€
Software	0€
Total	***21664.23€***

Table 10: Final budget sorted, self elaborated

9.8 Control Management

Finally, after the whole budget has been established, it becomes mandatory to track potential deviation between estimations and actual results throughout the completion of this project. We will rely on two key indicators to highlight the said deviations:

$$\text{Cost Deviation(CD)} = Cost_{expected} - Cost_{real}$$

$$\text{Efficiency Deviation(ED)} = Time_{expected} - Time_{real}$$

This deviation will be computed for each accomplished task and this gives insights into how each task is diverging from the initial planning. Thus, adding 2 more key performance indicators to the project which ideally should be maximized to avoid missing any deadline and allow for the possibility of implementing more features like taskloop or even further. If either CD or ED becomes negative and keeps growing its absolute value, this would warn that some projects decisions should be made, such as re-scheduling or discarding less important tasks. The results can be found at (18) and (13).

10 Sustainability Report

10.1 Economic Dimension

The grand total cost of this project is a significant sum of **21664.23€**, which given the inherent abstract nature of this unique problem, requires several human resources. The last is without any doubt the biggest expense of them all but also the most relevant that should not be affected by cuts in any case. The equipment and indirect costs are also basic costs that couldn't be reduced at all, but its is true that the estimation of the salaries may be on the upper side of the spectrum. Also, it's worth noting that thanks to the usage of free and Open Source software, no unnecessary costs have been added to the budget.

The implementation of ClangIR onto Clang will avoid memory mangement bugs and potentially reduce execution times of millions of executable files around the world, both for end-users and companies, which benefit greatly of improvements in the C compiler infrastructure. It is also worth noting that companies as Google use Clang and MLIR to compile TensorFlow, so any improvements in this field would enhance execution times of training and inference on neural networks, which are used on the vast majority of companies to implement machine learning models on a wide range of sectors like retail, healthcare, energy, transportation, manufacturing, finance... Finally, one should infer that the only economic risk linked to this thesis would be if ClangIR did not yield the expected outcomes, therefore "wasting" the budget, funds that would be invested. But this is not the case of this thesis.

10.2 Environmental Dimension

The estimation computed in the **indirect costs** section, gives us insights on how much energy is being spent on this project, which amounts to $92h \cdot 3.39kW = 311.88kWh$. According to the official guide to compute the carbon footprint given by the Spanish Government [38], the carbon emission rate for each single kWh of energy used is roughly $0.302kgCO_2/kWh$. Therefore, the completion of this thesis will produce approximately $311.88kWh \cdot 0.302kgCO_2/kWh = 94.19kgCO_2$. To diminish the carbon footprint of this project, the author has opted to commute to the office using rented bicycles, thereby mitigating any additional CO2 emissions. Thus, emissions are kept to the lowest and most essential, even though estimations of the consumption due to connectivity have been omitted due to the great complexity of computing such figures and the lack of data from internet providers.

Taking into account that a single BIC pen weighs around 6g and that the production of 1kg of plastic leaves a carbon footprint of 6 kg of CO_2[39], then the pens produced about 0.072 kg of CO_2. Not only that but also we can roughly estimate that the journal and notebook, which both contain around 100 pages and to simplify the computations, assume that both have a standard A4 surface. Then the environmental cost of both is about 9.06 kg of CO_2[40]. Therefore, everything altogether has only produced 9.132 kg of CO_2, which is miniscule compared to the potential reduction in emissions that this project may achieve.

The impact of this project into the environment is clear, after ClangIR is introducing in the Clang compilation flow and execution times get reduced, the energy used by a computer will be lowered (Even if it's a minuscule value) and as a consequence of this, fewer tons of CO_2 will be produced by clang users. Given the novelty of the problem at hand, there are currently no alternatives, but as previously mentioned, OpenMP has been subject to multiple implementations, such as the OMP dialect inside MLIR for example, which probably will significantly reduce the number of human hours and energy needed to

develop these features. Lastly, it's important to highlight that this project poses no environmental risks, given its computer science research-oriented nature.

10.3 Social Dimension

The author has performed an introspective analysis of how could this project impact personal growth. Not only it will serve to partially fulfill a deep hunger of knowledge (Compilers have been the author's favourite subject , sparking curiosity out of him throughout the degree), but also will broaden his horizons and enable entry into this niche world of compiler engineering. The need to quickly absorb and integrate new knowledge and learn to contribute in open source communities will enhance the authors abilities and toolbox. The tutors of this thesis will also have the chance to enhance their mentoring and project management skills further beyond.

Additionally, it will serve a purpose, improving the already existing Open-Source Compiler infrastructure from LLVM and paving the way for other contributors to build even more features on top of the ones proposed in this thesis. Although the contribution is tiny, this will enable other more seasoned engineers inside the ClangIR project to avoid allocating their times to such features and therefore, they will be able to tackle more challenging tasks, saving hundreds of human hours.Note that this project poses no discernible risk or harm to the social dimension.

10.4 Sustainability Matrix

To summarize all the previous sections, the following sustainability matrix (11) encapsulates the key dimensions discussed:

Dimension/Metrics	**PPP**	**Useful Life**	**Risks**
Economics	21664.23€ and up to 800 human hours	Indirect benefits (from saving) and no further costs	None
	6/10	7/10	0/0
Environment	103.322 CO_2	Eventual reduction of carbon footprint generated by Clang program builds around the world	None
	8/10	10/10	0/0
Social	Positive impact for all the involved parties	Pave the way for others and saved hundreds of hours to other contributors	None
	10/10	9/10	0/0
	24/30	26/30	0/0

Table 11: Sustainability Matrix, self elaborated

11 Updates

11.1 Context and Scope

The project began smoothly with a focus on understanding the nuances of LLVM, Clang, CIR, MLIR, and OpenMP. However, as development progressed, the author encountered the complexities of the Clang code-base, which required significant time to become familiar with.

The execution of the thesis underwent changes due to the author's initial uncertainty and lack of experience. This included not only unfamiliarity with compiler development workflows and limited knowledge of advanced Git usage but also significant communication overheads within the ClangIR community, including pull requests, seeking guidance, and making design decisions that required prior consultation. This was already expected as a possible risk as stated on 8, but it was certainly underestimated.

While the primary goal of the thesis remains dissemination, the specific development goals evolved due to the complexity and early stage of the CIR project and the OpenMP dialect of MLIR. Unexpectedly, barrier and taskyield directives were implemented alongside taskwait due to the symmetry of implementation with other directives. Consequently, the project's emphasis shifted from the quantity of directives implemented to the support for a greater number of clauses within each finally implemented directive. Due to time constraints, taskloop wasn't developed at the end.

11.2 Tasks and Scheduling

One crucial point to note is that while initially committing to dedicating 30 hours per week to the thesis seemed attainable, by the end of February, the author's schedule became increasingly crowded with additional responsibilities, including part-time work and coursework. As a result, after the sixth week, the average weekly dedication decreased to 27.5 hours, showcasing the author's remarkable time management abilities. Consequently, this shift in workload has led to adjustments in the total time invested in the thesis.

Not only did some tasks consume significantly more development time than originally anticipated due to the aforementioned issues, but also a new task (I5) emerged, focusing on implementing data sharings on the task and taskloop constructs. A summary of the changes in direction or timeline is presented:

- **E2 - Compiling LLVM/ClangIR:** A critical oversight in scheduling was failing to consider that every code change would necessitate recompiling a substantial portion of the Clang project. These compilations typically take around 20 minutes for small incremental changes, but when switching git branches, a full project recompilation was required, taking at least 1 hour each time. Though not meticulously tracked, an estimated 50 hours were approximately invested in compilation time.

- **M5 - Weekly Meetings:** These meetings, originally estimated to last 1 hour each, were held weekly except for 2 weeks. Several meetings exceeded the allotted time, and the 5 scheduled meetings with Rivero were initially overlooked. Realistically, a total of 27 hours were allocated.

- **D1 - Code Documentation:** Documentation for directives was deemed redundant and unnecessary due to the nature of CIR documentation and an already existing MLIR documentation for

the OMP dialect. Consequently, this task was discarded, and the allocated time was reassigned to other tasks.

- **I1 - Initial Implementation of taskwait/taskyield/barrier:** The objectives of this task changed to try to implement 2 other directives at the same time, and the invested time increased to approximately 75 hours taking into account the delays associated with the pull request process.

- **I4 - Optional Implementation of taskloop:** Unfortunately, due to time constraints and unexpected project complexities, the taskloop directive was not implemented. However, a high-level description of its potential implementation outside the thesis scope is provided on 17.

- **I5 - Exploring Data-Sharings:** Extensive research was conducted to explore two main approaches for implementing this feature. However, due to the early stage of data sharings in the MLIR OpenMP dialect and the significant time investment required (which could constitute an entire thesis on its own), this task was ultimately discarded. Further discussion is available in Section 17.

- **Miscellaneous:** Allocating time for unexpected events proved beneficial, as the author experienced illness on several occasions, resulting in reduced or zero productivity. Additionally, time was dedicated to learning advanced git concepts and gaining practical experience, along with community engagement through Discord, discourse threads, GitHub issues, and CIR monthly meetings.

The upcoming table provides updates on all mentioned tasks 12, including those not explicitly discussed but still affected by any deviations, as outlined in Table 13. Additionally, a revised Gantt diagram 11.2 (Note that dependencies have changed slightly) will be presented to illustrate the project's evolution and the distribution of workload across the weeks, in comparison to the original planning.

Task Description	**Acronym**	**Duration**	**Dependent tasks**	**Requirements**	**Human Resources**
Context and scope	M1	25 hours	-	*Overleaf*, PC	GEP
Project planning	M2	20 hours	M1	*Overleaf*, PC	GEP
Budget and sustainability	M3	18 hours	M2	*Overleaf*, PC	GEP
Revised document	M4	5 hours	M3	*Overleaf*, PC	GEP
Weekly meetings	M5	27 hours	-	-	DIR, COS
Total Management Hours: 95					
Final thesis document	D2	60 hours	M4	PC, Overleaf	DIR
Final presentation	D3	40 hours	D1, D2	PC, Overleaf	DIR
Total Documentation Hours: 100					
Understanding CMAKE/NINJA	E1	5 hours	-	PC, VsCode	-
Compiling LLVM/ClangIR and setting up the env.	E2	60 hours	E1	PC, VsCode	COS
Total Environment Hours: 65					
Delve into LLVM framework	R1	100 hours	-	PC, VsCode, [23], [22], [12], [3]	DIR,COS
Becoming familiar with MLIR	R2	30 hours	R1	PC, VsCode, [24], [25], [27]	-
Reviewing OpenMP	R3	10 hours	-	PC, VsCode, [28]	-
Getting to know ClangIR	R4	20 hours	R2,R3	PC, VsCode, [5], [18]	-
Total Research Hours: 160 hours					
Initial implementation of taskwait	I1	75 hours	R4, E2	PC, VsCode, Github, [5]	COS
Development of the task directive	I2	100 hours	R4,E2	PC, VsCode, Github, [5]	COS
Implementation of taskgroup	I3	35 hours	R4,E2	PC, VsCode, Github, [5]	COS
Exploring Data-Sharings	I5	10 hours	R4,E2	PC, VsCode, Github [5]	COS
Total Implementation Hours: 220 hours					
Miscellaneous	MISC	40 hours	-	-	CIR community
Total: 680 hours					

Table 12: Summary of project tasks updated, self elaborated

Considering all of these updates and the 2 key indicators that were presented at the control management section 9.8, deviations will be taken into account in the final budget. Efficiency Deviation is summed up on table 13, so the reader can have a greater understanding on the evolution of the project.

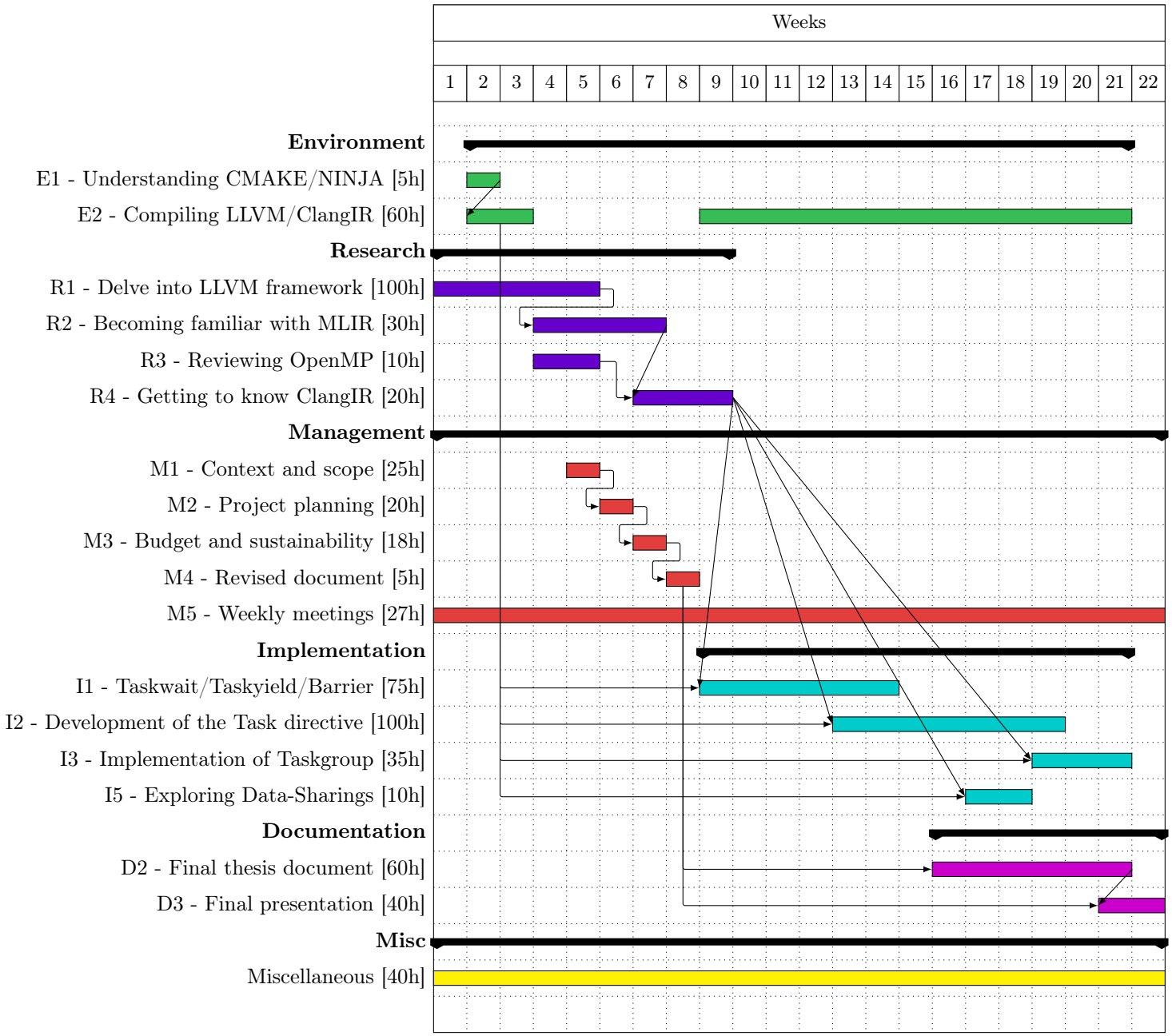

Figure 12: Updated Gantt Diagram of the project, self elaborated using the 'pgfgantt' package from LaTeX

Task	**Acronym**	**Expected Duration (hours)**	**Real Duration (hours)**	**Efficiency Deviation (hours)**
Context and scope	M1	25	25	0
Project planning	M2	20	20	0
Budget and Sustainability	M3	18	18	0
Final Document	M4	10	5	5
Weekly Meetings	M5	22	27	5
Code documentation	D1	30	0	30 (Discarded)
Final thesis document	D2	60	60	0
Final presentation	D3	40	40	0
Understanding CMAKE/NINJA	E1	5	5	0
Compiling ClangIR	E2	10	60	50
Delve into LLVM	R1	100	100	0
Becoming familiar with MLIR	R2	45	30	15
Reviewing OpenMP	R3	5	10	5
Getting to know ClangIR	R4	30	20	10
Initial Implementation of Taskwait/Taskyield/Barrier	I1	40	75	35
Development of the Task directive	I2	100	100	0
Implementation of Taskgroup	I3	40	35	5
Optional implementation of Taskloop	I4	70	0	70 (Discarded)
Exploring Data-Sharings	I5	10	10	0
Miscellaneous	MISC	50	40	10

Table 13: Time Deviation for tasks

11.3 Budget

11.3.1 Software

During the course of this thesis, the author opted to subscribe to Overleaf's premium plan (8€ per month for 4 months, bought exclusively for the thesis). This decision was motivated not only by the need to circumvent the installation and configuration of a local LaTeX environment, given the lengthy compile times and limitations of the free version, but also to ensure agility, ease of collaboration with tutors, and the ability to edit the document on any device with a browser. The other software used throughout remained unchanged.

Product	Cost	Amortization
Overleaf Premium	32€	32€
Ubuntu 22.04 LTS	0€	0€
LaTeX	0€	0€
VIM	0€	0€
Clang	0€	0€
GitHub	0€	0€
Chrome	0€	0€
Visual Studio Code	0€	0€
Total	**32€**	**32€**

Table 14: Updated Software cost, self elaborated

11.3.2 Hardware

The hardware used has remained the same, but bearing in mind the minor change that the total time invested has changed to 680 hours. Recalling the formula of section9.1, the approximate amortization is:

$$Amortization = 799€ \cdot \frac{1\text{useful life}}{5years} \cdot \frac{1year}{52weeks} \cdot \frac{1week}{30hours} \cdot 680hours = 69.57€$$

11.3.3 Indirect Costs

As the thesis completion time was cut down to 680 hours, thus some items of the indirect cost must be revised (The rest remain unchanged):

- **Electricity**:

$$680 \text{ hours} \cdot \frac{1recharge}{16hours} = 42.5 \approx 43 \text{ recharge}$$

$$43 \text{ recharge} \cdot \frac{2hours}{1recharge} = 86 \text{ hours}$$

$$43 \text{ recharge} \cdot \frac{73.7W}{1recharge} \cdot \frac{1kW}{1000W} = 3.169kW$$

$$ElectricityCosts = 86h \cdot 3.169kW \cdot \frac{0.0489€}{1kWh} = 13.33€$$

- **Connectivity**:

$$ConnectivityCosts = \frac{30€}{1month} \cdot \frac{1month}{30days} \cdot \frac{1day}{24hours} \cdot 680 \cdot 0.8 \text{ hours} = 22.67€$$

- **Location**:

$$LocationCosts = \frac{255€}{1month} \cdot \frac{1month}{30days} \cdot \frac{1day}{24hours} \cdot 680 \cdot 0.5 \cdot 0.33 \text{ hours} = 39.74€$$

Resource	Price
Location	39.74€
Internet	22.67
Training	20.00€
Consumables	17.62€
Electricity	13.33€
Office	0.00€
Total	**113.36€**

Table 15: Updated indirect costs, self elaborated

11.3.4 Human Resources and Contingency

Similar to previous sections, the budgets of human resources should be revised:

Role	Salary	Expected Hours	Total Cost	Total + Social Security
Software Engineer	18.23€/hour (35K€/year)[32]	485 hours	8841.55	11936.09€
Project Manager	20.31€/hour (39K€/year)[35]	195 hours	3960.94€	5347.27€
Total	-	**680 hours**	12802.49€	**17283.36€**

Table 16: Updated human resources costs, self elaborated

The miscellaneous time allocation encompasses three primary sub-tasks:

- Covering two full workdays of sick leave, adhering to Spanish regulations where the first three days of common sickness are unpaid. As these days are not included in the budget, they will not be accounted for.
- Time dedicated to learning advanced concepts and management of Git.
- Communication between the author and the CIR community, including interactions with individuals such as Bruno Cardoso, Fabian Mora and Kiran Chandramohan. While this communication involves the use of the author's free time, it remains integral to project progress and collaboration.

Additionally, it's noteworthy that other human resources, such as members of the CIR community, have contributed predominantly during their leisure time. Finally, taking all this information into account, the new contingency budget is the following:

Source	Price (€)	Percentage	Cost (€)
Hardware	74.78	10%	7.48€
Human Resource	17283.36	10%	1728.33€
Indirect Costs	113.36	10%	11.34€
Total	-	-	**1747.15€**

Table 17: Updated contingency costs, self elaborated

11.3.5 Updated final budget

To wrap things up, a table is put forward, so the reader can grasp how the budget has diverged from the original one that was proposed at an early stage of this thesis. The metric to assess this divergence will be the aforementioned cost deviation defined in section 9.8:

Item	Expected Cost	Real Cost	Cost Deviation
Human Resources	18513.89€	17283.36€	1230.53€
Contingency	1877.76€	1747.15€	130.61€
Unexpected Costs	1077.51€	0€	1077.51€
Indirect Costs	120.29 €	113.36€	6.93€
Hardware	74.78€	69.57€	5.21€
Software	0€	32€	32€
Total	21664.23€	19245,44€	2418.79€

Table 18: New Final Budget, self elaborated

It's worth noting that despite the project undergoing notable changes in the defined tasks, unexpected costs were ultimately not utilized. The author initially intended to allocate these costs towards implementing taskloop or data sharings. However, considering the substantial development time required for these tasks, it became evident that they would far exceed the 40 extra hours originally allocated in the unexpected costs budget.

11.4 Sustainability

11.4.1 Matrix Updated

Taking all these budget and time deviations into account and know that the authors has a wider knowledge of the impact and implications of this thesis, the sustainability matrix should be updated:

- **Economics**: The updated total cost of the project amounts to 19245.44€, with approximately 750 human hours invested into it. It's crucial to consider the time contributed by the tutors José Miguel and Roger, the GEP tutor Xavier, the CIR community, and, last but not least, the jury that will read this thesis and evaluate its presentation.

- **Environment**: Given that the energy consumption made has changed:

$$86h \cdot 3.169kW = 272.53kWh$$

$$272.53kWh \cdot 0.302kgCO_2/kWh = 82.31kgCO_2$$

$$82.31kgCO_2 + 9.132kgCO_2 = 91.44kgCO_2$$

Dimension/Metrics	**PPP**	**Useful Life**	**Risks**
Economics	19245.44€ and up to 750 human hours	Indirect benefits (from saving) and no further costs	None
	7/10	8/10	0/0
Environment	103.322 CO_2	Eventual reduction of carbon footprint generated by the execution of programs compiled using CIR	None
	9/10	10/10	0/0
Social	Positive impact for all the involved parties	Pave the way for others and saved hundreds of hours to other contributors	None
	10/10	9/10	0/0
	26/30	27/30	0/0

Table 19: Updated sustainability matrix, self elaborated

11.4.2 Considerations

Given that this project is at its final stage, and that more than half of this project has been accomplished when writing these updates, a retrospective assessment should be done, to evaluate the sustainability of the project. This could be achieved addressing relevant questions that were suggested to the author when taking the project management course:

If you carried out the project again, could you use fewer resources?

For what material resources concerns, it is already cut down to the bare minimum minimalism. But it is true that less power could have been used up if the author already possessed an efficient compiler development computer. Also a ton of time could have been saved if the author had a bigger prior knowledge of the subject, clang and contributing to open source, however this uncertainty and discovery phase was needed and it's a common barrier in most real world projects.

Is the expected cost similar to the final cost? Have you justified any differences (lessons learnt)?

The modified cost has already been justified and it's lower than initially expected, due to the removal of tasks I4 and D1.

Has undertaking this project led to meaningful reflections at the personal, professional or ethical level among the people involved?

It has undoubtedly sparked connections and relationships between the author and a host of main contributors, such as Bruno Cardoso, Roger Ferrer and José Miguel, every one of them assisting and enhancing the author's knowledge in modern compilers and specifically the LLVM ecosystem. Not only that, but also raising the author problem-solving tool set, time management and professional prospects.

Who will benefit from the use of the project? Could any group be adversely affected by the project? To what extent?

Future contributors of ClangIR which want to implement OpenMP features, the big tech companies

behind Clang, and the final users who use this compiler. Given the additive nature of a contribution, and that these are revised meticulously by the reviewers, It can't have a direct negative impact.

12 Laws, Licenses and Regulations

These sections aim to give the reader intuition of which major laws affect this project and how these are abided with it. Obviously, there are other minor licenses such as MIT (Visual studio Code) which aren't worth covering.

12.1 Apache 2.0 LLVM License

The contributions are being updated into ClangIR, therefore the license of this project must be revised. ClangIR is a fork of the great LLVM project, therefore its license is implicitly inherited, which is **Apache 2.0 with LLVM exceptions**.

Apache 2.0 is a permissive open-source license that allows contributors to use, modify, distribute and even sublicense the software under certain conditions. Given that this thesis only uses and modifies the original code without changing the license, retaining copyright notices and providing attribution to the original authors, it meticulously adheres to the license.

After reading the **LLVM Exceptions** section, one can infer that embedded portions of the LLVM project can be freely used in proprietary software, even object and binary files of LLVM, may be redistributed without complying with certain conditions of the Apache License nor acknowledging this license.

In contrast to Clang, GCC uses a GPLv3 copyleft license. Both Apache and GPLv3 allow using the output objects in embedded or proprietary software. Nonetheless, the major difference in terms of license which makes Clang unique and so versatile, is that the source code can be integrated into another proprietary or Open Source project without the obligation of making it Open source and free. On the contrary, GCC regulations forces the license of any derivative work to be GPLv3, ensuring that it remains free and open. That is one of the key reasons for the industrial success of the LLVM community.

12.2 License of generated files

On the previous section, the license of the source code of the project was discussed, however what is even more crucial is to acknowledge is which regulations should be abided by the object files (Output generated by the compiler). Following the Apache 2.0 License, it is undeniably clear that these object files will obey the regulations imposed by the source code, as a consequence all the objects files generated throughout the thesis are guaranteed to comply with all relevant legal requirements.

12.3 Law BOE-A-2011-9617 (Science, Tech, and Innovation)

As this project may be regarded as a research undertaking carried out at the Facultat de Informàtica de Barcelona, it must adhere to laws and regulations related to research of the country of this faculty, Spain. The most relevant law is BOE-A-2011-9617[41] which specify a set of requirements that research projects at public universities must abide. Given the relevance of this legislation, a justification of article

15 (Duties of the researcher) will be provided:

b) Avoid plagiarism and unauthorized appropriation of authorship of scientific or technological works of third parties

On every document related to the project (This dissertation and the presentation), all the individuals and organizations that provided assistance or knowledge, will be acknowledged.

c) Inform the entities for which they work of all findings, discoveries, and results susceptible to legal protection, and collaborate in the processes of protection and transfer of the results of their research

Almost every week, Roger Ferrer (representative from BSC) and José Miguel Rivero (representative from UPC) are regularly provided with updates on the project's progress and advancements.

d) Disseminate the results of their research, as indicated in this law, so that the results are utilized through communication and transfer to other research, social, or technological contexts, and if applicable, for their commercialization and valorization. In particular, research personnel must ensure and take the initiative to ensure that their results generate social value

This dissertation will be undeniably shared, but the results and code developed will be exposed in the LLVM code repository, therefore it will be publicly exposed and will unquestionably provide value as exposed in the next item.

e) Ensure that their work is relevant to society

This thesis will provide improvements to the original Clang codebase, which is a popular c/c++ compiler that is in its way to becoming the standard compiler. An increasingly big number of C/C++ programmers use Clang every day to generate builds of their projects. When CIR gets upstreamed into the stable version of Clang, all the builds done using OpenMP, related to the implemented directives, will be generated thanks to the code generation created in this thesis.

h) Inform the entities for which they work or that finance or supervise their activity of possible delays and redefinitions in the research projects for which they are responsible, as well as the completion of projects, or the need to abandon or suspend projects earlier than planned

The author completed a project management course before commencing the actual thesis, which has been meticulously supervised by the entities through the project management document and these updates, alongside the aforementioned meetings.

j) Use the name of the entities for which they work in carrying out their scientific activity, in accordance with the internal regulations of such entities and the agreements, pacts, and conventions subscribed to by them

The collaboration of the Barcelona Supercomputing Center and University Politècnica de Catalunya has been stated several times throughout the thesis.

l) Adopt the necessary measures to comply with applicable regulations on data protection and confidentiality

As stated before, this project won't store any personal or confidential data.

13 Context and knowledge integration

In this section, a comprehensive explanation of the technologies and frameworks used throughout this thesis will be detailed, so the reader can grasp the current state-of-the-art compiler development. Moreover, this explanation will include an overview of the ecosystem, the main ideas, and the practical usage of these tools. As a result, this will give a holistic and detailed view of the compiler front-end Clang, and the middle-end (LLVM, MLIR and ClangIR) so the upcoming practical work section can be understood with ease.

13.1 Clang

13.1.1 Definition

Clang is the compiler front-end for the C-based languages, being C, C++, Objective-C, Objective-C++, CUDA, OpenCL of the LLVM project. Clang was born out of Apple's need to develop and customize the existing Objective-C compiler in 2005, especially due to the stricter code-sharing requirements introduced by GCC's shift from GPLv2 to GPLv3. Back in the day, the only stable compiler for Objective-C was GCC, but given the cumbersome code-base and the lack of priority given to Objective-C inside the GNU community, Apple decided to part ways and starting a new compiler from scratch, that's when Clang was born, and 2 years later It would be opened to open-source. The goals of Clang were very clear from the get-go: reducing the complexity of the code-base to avoid churn in development time, to retain a greater amount of information during the front-end to give more detailed error reports and to achieve a modular design.

Clang is a widely used compiler frontend that supports a broad range of C++ standards, including C++98, C++11, C++14, C++17, C++20 and C++23. It has been integrated into popular IDEs and tools such as Visual Studio Code, CLion, XCode, and CMake, making it a convenient choice for developers. Clang's tight integration with the LLVM compiler infrastructure provides several advantages. Firstly, Clang natively supports LLVM IR, enabling interaction with LLVM's extensive optimization tools and backends. This translates to a complete compiler toolchain with support for a wide range of target architectures, including x86, arm, and others. It's important to note that this integration also facilitates cross-compilation, where source code can be compiled on one platform for execution on another. Furthermore, Clang offers performance benefits compared to GCC. It generally consumes less memory and exhibits slightly faster compilation times, especially on modern hardware (as noted in [42]). These makes Clang an attractive option for resource-constrained environments and large-scale projects.

The simple and robust Clang AST is composed of various nodes, each corresponding to different constructs in the source code. It preserves detailed and accurate source location, type, scope and linkage information. Its design is intended to be abstract, modular, and extensible, to enable future modifications naturally when new frameworks, C-based languages or new C++ standards appear [43]. Figure (13) shows a subset of relevant nodes of the Clang AST.

Clang provides several tools and APIs for working with the AST:

- **AST Matchers**: A high-level library for matching specific patterns in the AST, useful for static analysis and transformations.

- **RecursiveASTVisitor**: A base class for creating custom AST visitors to traverse and process AST nodes.
- **LibTooling**: A library that facilitates the development of custom tools for analyzing and transforming the AST.

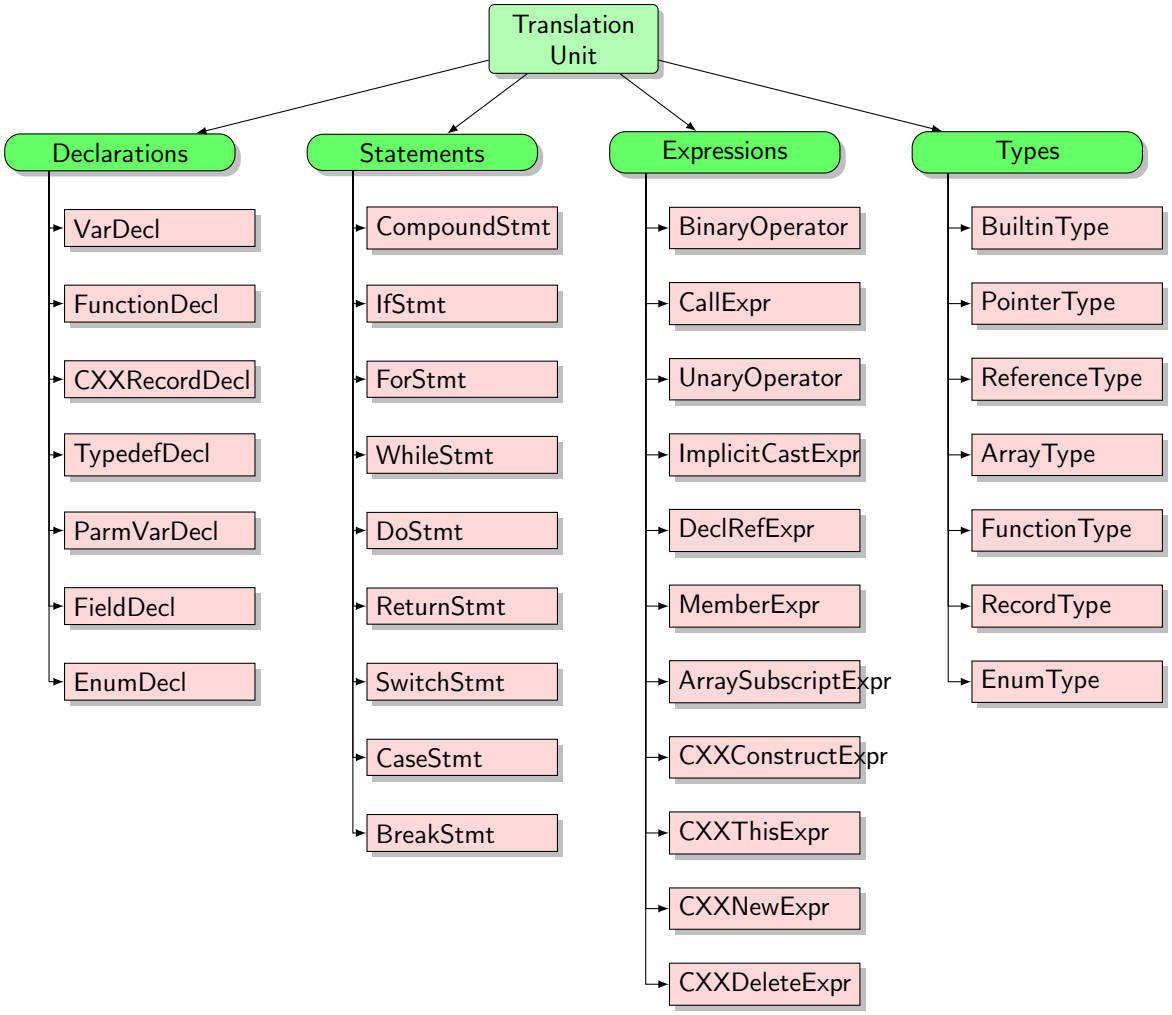

Figure 13: Core of Clang AST, self-elaborated using the LaTeXpackage "tikz"

13.2 LLVM IR

13.2.1 Definition

In the introductory section, some discussion about LLVM was performed, describing it as an umbrella project encompassing essential components such as OpenMP, Clang, LLD (Linker), LLDB (Debugger), and Flang. However, at its core lies the LLVM intermediate representation (LLVM IR), serving as the foundational IR for all associated compilers and tools.

The LLVM IR plays a pivotal role in the compilation process. Following translation from source code to LLVM IR, subsequent optimizations and translations into assembly languages occur. Therefore, for organizations or individuals aiming to develop a robust and modern compiler for a custom language, minimal effort is required. By crafting the front-end for the source language and implementing code generation from the AST to LLVM IR via its C++ API, extensive support for a variety of target assembly languages and optimizations can be achieved. This intermediate representation is organized into four layers: Instructions, Basic Blocks, Functions, and Modules. Instructions represent individual operations, Basic Blocks represent groups of instructions, Functions encapsulate code logic, and Modules contain functions and global data. This hierarchical structure simplifies code representation and manipulation within LLVM IR.

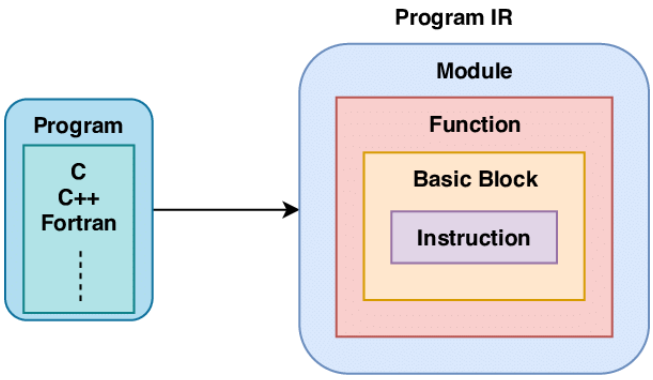

Figure 14: Hierarchical structure of an LLVM program, retrieved from [44]

To deepen the comprehension of LLVM IR, it's crucial to recognize that it operates within an assembly-esque framework. It falls below the abstraction level of C/C++ but remains higher in abstraction than, say, x86 assembly language. To be more concise, it is a strongly typed reduced instruction set computer (RISC) instruction set, that serves as an abstract language that aims to generalize real assembly languages. A notable difference with real low level languages is that instead of using a limited set of real and tangible registers, all the results can be stored using an infinite number of virtual registers (On the back-end process each target language defines the register allocation algorithms for achieving equivalent programs with fewer registers) which are commonly described as "SSA values", which are identified in the LLVM IR code as **%x**, where x can be any string, like variables.

Programs translated to LLVM IR are always produced in the aforementioned SSA (Single static assignment) form, where each variable is assigned exactly one time (Therefore it can be referenced several times, but the value cannot be changed, a new virtual register has to be created to modify a value). This

structure is followed to enable efficient translations to machine code, but most importantly, it makes numerous analyses needed prior to optimization easier to perform, such as generating use-define chains. Moreover, SSA-based optimizations are often simpler and more powerful than their non-SSA form prior equivalent algorithms.

To fully support SSA form and produce equivalent control flows, LLVM introduces a special instruction called the **phi** Φ function. The **phi** function is used in situations like if-then-else statements to decide which input value to select as the output value based on the control flow path taken. For example, if a variable a is assigned different values depending on whether the program follows the **then** or **else** path, the **phi** function determines the correct value of a for the subsequent code. This involves creating multiple versions of **a** (e.g.: a1, a2, and a3) to represent its different states. The **phi** function then selects the appropriate version based on the execution path, ensuring the variable maintains a single assignment property consistent with SSA form.15

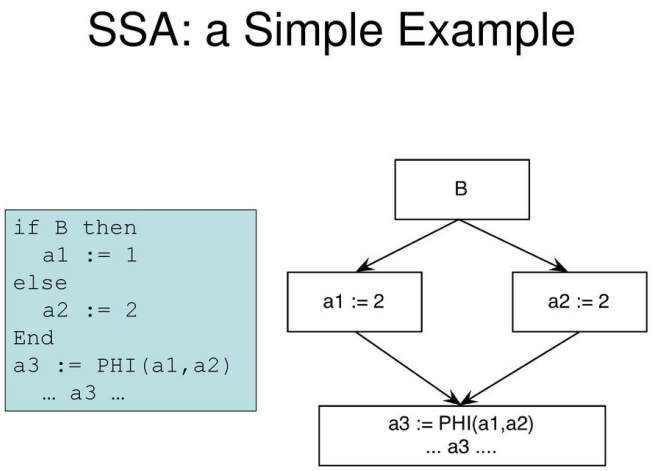

Figure 15: Example of Phi function, Retrieved from UTSA CS5363 slides [45]

To illustrate all the aforementioned concepts, consider the following source C code, which is first translated to a Clang AST and then generated into LLVM IR:

```
1  float add(float a, float b){
2      return a + b;
3  }
4
5  int main(){
6      float x = 3.14;
7      float y = 420.0;
8
9      return add(x,y);
10 }
```

Listing 1: Sample source code in C, self-elaborated

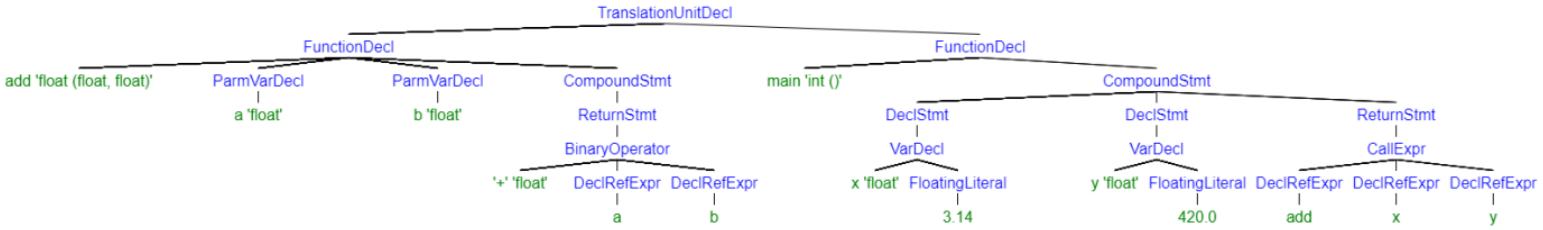

Figure 16: Simplified example of the Clang-AST produced using the source file, self-elaborated

```
1
2  define dso_local float @add(float noundef %a, float noundef %b) #0 {
3  entry:
4    %a.addr = alloca float, align 4
5    %b.addr = alloca float, align 4
6    store float %a, float* %a.addr, align 4
7    store float %b, float* %b.addr, align 4
8    %0 = load float, float* %a.addr, align 4
9    %1 = load float, float* %b.addr, align 4
10   %add = fadd float %0, %1
11   ret float %add
12 }
13
14 define dso_local i32 @main() #0 {
15 entry:
16   %retval = alloca i32, align 4
17   %x = alloca float, align 4
18   %y = alloca float, align 4
19   store i32 0, i32* %retval, align 4
20   %0 = bitcast float* %x to i8*
21   call void @llvm.lifetime.start.p0i8(i64 4, i8* %0) #2
22   store float 3.14, float* %x, align 4
23   %1 = bitcast float* %y to i8*
24   call void @llvm.lifetime.start.p0i8(i64 4, i8* %1) #2
25   store float 420.0, float* %y, align 4
26   %2 = load float, float* %x, align 4
27   %3 = load float, float* %y, align 4
28   %call = call float @add(float noundef %2, float noundef %3)
29   %conv = fptosi float %call to i32
30   %4 = bitcast float* %y to i8*
31   call void @llvm.lifetime.end.p0i8(i64 4, i8* %4) #2
32   %5 = bitcast float* %x to i8*
33   call void @llvm.lifetime.end.p0i8(i64 4, i8* %5) #2
34   ret i32 %conv
35 }
```

Listing 2: Simpliefied translation from AST to LLVM IR, self-elaborated

The generated LLVM IR code is significantly more detailed and complex than the original C source code, resembling machine code more closely than a high-level programming language. For instance, memory alignments must be explicitly specified, and different arithmetic instructions are required for various data types, such as i32 for integers and float for floating-point numbers. Additionally, the code features special LLVM intrinsic functions, which provide access to low-level operations and optimized routines

specific to LLVM. It's important to note that this example is simplified, with headers removed for clarity. These headers typically contain critical information about data layout, endianness, and the target triple, which specifies the CPU architecture, device type, and operating system. This information is essential for adhering to ABI conventions, ensuring that the generated code can run correctly on the intended hardware and software environment.

It supports two distinct source file formats: the human-readable assembly format, denoted by the extension .ll, and the compact bitcode format, represented by files with the extension .bc, which uses binary encoding. Moreover, LLVM boasts a robust ecosystem of tools designed to interact naturally with these files. These tools facilitate various essential processes, including assembly (llvm-as) for converting textual representations to bitcode, disassembly (llvm-dis) for the reverse operation, compilation (llc) for translating LLVM IR to machine code, interpretation (lli) for direct execution of LLVM IR, and linking (llvm-link).

13.2.2 Usage

In order to make practical use of the full capabilities of LLVM, as already mentioned, one can use its API to take advantage of its robust code generation, backend translation, and extensive analysis and optimization rich infrastructure. Bearing in mind the program hierarchy defined in 15, the important classes that one will use to generate this IR and later on optimizing it, fall out naturally:

Modules: The 'llvm::Module' class is the top-level container in LLVM IR, representing a single unit of code that can contain functions, global variables, and other data. Modules allow for the organization and encapsulation of code, making it easier to manage, compile, and link large code-bases.

IRBuilder: The 'llvm::IRBuilder<>' class is a utility that simplifies the creation of LLVM instructions. It provides a convenient interface for constructing instructions and managing their insertion into basic blocks, thus streamlining the process of generating LLVM Intermediate Representation (IR) code.

Functions and Basic Blocks: Functions in LLVM are instances of the 'llvm::Function' class, which represent individual routines or procedures. Each function contains basic blocks, instances of the 'llvm::BasicBlock' class, which are sequences of instructions with a single entry point and a single exit point. Basic blocks help structure the control flow within functions.

Pass Manager: The 'llvm::legacy::PassManager' class is responsible for managing and running a sequence of optimization and analysis passes over the LLVM IR. Passes can perform various transformations and analyses to improve the efficiency and performance of the generated code. There are other types of pass managers, such as 'llvm::FunctionPassManager' for function-level optimizations and 'llvm::LoopPassManager' for loop-level optimizations, all of which serve as interfaces for optimizing LLVM IR code.

Target Machine: The 'llvm::TargetMachine' class encapsulates the details of the target architecture. It is used for generating machine-specific code from the LLVM IR. This class is crucial for backend translation and ensuring that the generated code is optimized for the specific hardware and abstracts all the complex information associated with each machine code language: ABI call conventions, the instruction

set, the registers, special features and specific optimizations.

Execution Engine: The llvm::ExecutionEngine class provides a way to execute LLVM code directly in-memory, facilitating just-in-time (JIT) compilation. It allows for dynamic execution of generated code, making it a powerful tool for runtime code generation and optimization.

Data Layout: The llvm::DataLayout class encapsulates information about how data is laid out in memory for a specific target machine. This is critical for generating correct and optimized code, as it provides details such as alignment requirements and size of data types.

Global Variables: The llvm::GlobalVariable class represents global variables in LLVM IR. These are variables that are accessible throughout the module and can be used for various purposes such as constants, static data, or external symbols.

On the appendix (A.3), an example on how to statically generate LLVM IR code, optimize it, and translate it to the system's assembly language is given. This serves as an extremely simplified implementation of the front-end, middle-end and back-end of a concrete static snippet of code, so the reader can acquire familiarity with LLVM API.

However, this approach was entirely static and does not reflect typical use cases of a real compiler. In practice, code generation is often dynamic, utilizing the visitor design pattern as commonly implemented in compilers to generate code for AST nodes recursively. For an example of code generation for a simple AST binary operation node, refer to the following example 13.2.2. Further information on the LLVM API [3] and its usage can be found in [46].

```
class BinaryOpNode : public ASTNode {
    std::unique_ptr<ASTNode> LHS, RHS;
    char op;

public:
    BinaryOpNode(char op, std::unique_ptr<ASTNode> lhs, std::unique_ptr<ASTNode> rhs)
        : op(op), LHS(std::move(lhs)), RHS(std::move(rhs)) {}

    llvm::Value* codegen(llvm::LLVMContext& context, llvm::IRBuilder<>& builder) override {
        llvm::Value* left = LHS->codegen(context, builder);
        llvm::Value* right = RHS->codegen(context, builder);

        switch (op) {
        case '+':
            return builder.CreateFAdd(left, right, "addtmp");
        default:
            std::cerr << "Unknown binary operator\n";
            return nullptr;
        }
    }
};
```

Listing 3: Basic sample of addition code generation, self-elaborated

13.3 MLIR

13.3.1 Definition

Multi-Level Intermediate Representation (MLIR) [27] is a versatile software framework designed to create reusable and extensible compiler infrastructure. It shares many similarities with LLVM IR, making it an evolution or specialization aimed at domain-specific languages and improving both compilation and compiler development time on heterogeneous hardware (e.g.: CPU, GPU, TPU, ASIC, FPGA). MLIR maintains a hierarchical representation of translated programs, retaining the concepts of modules, functions, and basic blocks (renamed as regions). Instructions are replaced by operations, which are high-level abstractions that receive attributes and operands as input and return values, similar to the concept of functions. This design allows for higher-level abstractions in the IR. Other concepts like SSA-values stored in virtual registers remain unchanged.

A notable feature of MLIR is its multiple intermediate representations, each with different levels of abstraction, sets of operations, types, and attributes, known as dialects. This allows developers to define their own compilers using multiple IRs, either intermixed or used successively in a process referred to as "lowering." Lowering involves transitioning from higher-level abstraction IRs to lower-level abstraction IRs, similar to code generation.

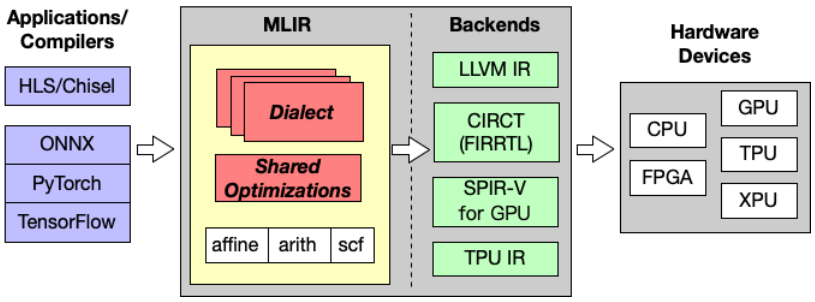

Figure 17: MLIR typical compilation flow and ecosystem, retrieved from [47]

Despite its powerful representation capabilities, MLIR documentation states that it is not intended to be a source language (like C or CUDA), nor does it support low-level machine code generation algorithms (such as legalization, register allocation, and instruction scheduling). To achieve low-level compilation, given that it lacks a backend, MLIR must be used in conjunction with LLVM, as illustrated in the diagram 5 thanks to its LLVM dialect. This is exemplified by the use case shown in 13.2.1

```
module {
  func @add(%arg0: f32, %arg1: f32) -> f32 {
  entry:
    %0 = llvm.alloca f32, 4
    %1 = llvm.alloca f32, 4
    llvm.store %arg0, %0
    llvm.store %arg1, %1
    %2 = llvm.load %0 : !llvm.ptr -> f32
    %3 = llvm.load %1 : !llvm.ptr -> f32
    %4 = llvm.fadd %2, %3
    llvm.return %4 : f32
```

```
  }

  func @main() -> i32 {
  entry:
    %0 = llvm.alloca i32, 4
    %1 = llvm.alloca f32, 4
    %2 = llvm.alloca f32, 4
    llvm.store %3.14, %1
    llvm.store %420.0, %2
    %3 = llvm.load %1 : !llvm.ptr -> f32
    %4 = llvm.load %2 : !llvm.ptr -> f32
    %5 = call @add(%3, %4) : (f32, f32) -> f32
    %6 = llvm.fptosi %5 : f32 to i32
    llvm.return %6 : i32
  }
}
```

Listing 4: Over-simplified translation from LLVM IR to MLIR, self-elaborated

The MLIR ecosystem encompasses a wide array of dialects, each serving distinct purposes and offering varying levels of abstraction. High-level dialects include affine, linalg, math, tensor, async, sparse_tensor, ml_program, etc. while low-level dialects comprise llvm, amdgpu, x86vector, arm_neon, nnvm, etc. In this thesis, notable examples of dialects utilized are the built-in and OpenMP (omp) dialects.

The strength of MLIR is further amplified by its robust support for computation data flow graphs and other features that make it highly compatible with machine and deep learning frameworks such as PyTorch and TensorFlow. This versatility and adaptability are key to its broad adoption and effectiveness in handling complex compilation tasks across diverse hardware platforms, and has become a standard on the machine learning realm.

13.3.2 Creating new dialects

Furthermore, MLIR greatly simplifies the process of creating your own dialect, operations and attributes thanks to its own tool, **mlir-tblgen** to avoid using the MLIR complex API similar to the process described in the LLVM section 13.2.2. To achieve this goal, we can modify the original MLIR project, creating a new directory inside **mlir/lib/Dialect** named **ThesisDialect** for example, and create a ThesisDialect.td file. This **.td** files that follow an ODS format contain table definitions of this custom dialect, such as new operations, types, and attributes:

```
// ThesisDialect.td
include "mlir/IR/Dialect.td"

// Define the Dialect.
def ThesisDialect : Dialect {
  let name = "thesis";
}

// Define an example operation: addition of two integers.
def ThesisAddOp : Op<"thesis.add", [NoSideEffect]> {
  let summary = "Adds two integers";
  let description = [{
    Adds two integer values.
```

```
  }];

  let arguments = (ins I32:$lhs, I32:$rhs);
  let results = (outs I32:$result);
}

// Define an example operation: multiplication of two integers.
def ThesisMulOp : Op<"thesis.mul", [NoSideEffect]> {
  let summary = "Multiplies two integers";
  let description = [{
    Multiplies two integer values.
  }];

  let arguments = (ins I32:$lhs, I32:$rhs);
  let results = (outs I32:$result);
}

def ThesisType : Type<"ThesisType"> {
  let cppNamespace = "::thesis";
  let description = [{
    A custom type for the Thesis dialect.
  }];
}
)
```

Listing 5: Example of creating a custom dialect

Immediately after creating the tablegen file, a placeholder cpp file is needed for the custom dialect object13.3.2, alongside minor modifications to the CMake files to enable the compilation of this new source code. After this step, we could finally produce MLIR code with our own custom dialect 13.3.2, but in order to be usable conversions between dialects (Primarily llvmir dialect) should be programmed.

```
module {
  func @main() -> i32 {
    %lhs = constant 42 : i32
    %rhs = constant 58 : i32
    %sum = thesis.add %lhs, %rhs : i32
    %product = thesis.mul %lhs, %rhs : i32
    return %sum : i32
  }
}
```

Listing 6: Sample code using custom "thesis" operations

```
#include "ThesisDialect.h"
#include "ThesisOps.h"
#include "mlir/IR/Builders.h"
#include "mlir/IR/Operation.h"
#include "mlir/IR/PatternMatch.h"
using namespace mlir;
using namespace thesis;

void ThesisDialect::initialize() {
  addOperations<
    #define GET_OP_LIST
```

```
    #include "ThesisOps.cpp.inc"
  >();
  addTypes<
    #define GET_TYPEDEF_LIST
    #include "ThesisTypes.cpp.inc"
  >();
  addAttributes<
    #define GET_ATTRDEF_LIST
    #include "ThesisAttributes.cpp.inc"
  >();
}
#define GET_OP_CLASSES
#include "ThesisOps.cpp.inc"
#define GET_TYPEDEF_CLASSES
#include "ThesisTypes.cpp.inc"
#define GET_ATTRDEF_CLASSES
#include "ThesisAttributes.cpp.inc"
```

Listing 7: Example of creating a custom dialect

This approach of creating MLIR dialects is the recommended, not only for agility but also to avoid having to change the source code if there's some change inside MLIR in future versions. Further information can be found at the MLIR documentation [48].

Beyond the core MLIR language, the ecosystem provides tools like mlir-translate and mlir-opt for enhanced development. mlir-translate acts as a bridge, converting code from various sources (C, LLVM) into MLIR for analysis and manipulation, or vice versa for integration with existing tools. Meanwhile, mlir-opt optimizes MLIR code, streamlining it through dead code removal, constant folding, and loop improvements, etc.

13.3.3 Dialect conversions

Despite all the remarkable features and abstractions that MLIR has to offer, it adds some inherent complexity to the development of compilers, which is defining how the interactions between these dialects should be done for our specific compilation pipeline. These interactions, known as **lowerings** or dialect conversions, can be easily specified by the developer by defining the following five key components:

1. **Type Converters**: These objects handle the logic of converting types between dialects. They define the mappings of data types from one dialect to another, which may not have the same set of types. For example, if some dialect supports **bfloat16**, but the target LLVM IR dialect does not directly support this data type, the type converter defined by the developer would specify how to map bfloat16 to an appropriate type for LLVM IR, such as i16. Moreover, the type converter ensures that the semantics of bfloat16 are preserved through appropriate conversions and handling of operations involving this type. Note that complex data types like matrices or tensors may require of the bufferization process (Converting tensor semantics to **"memref"** semantics, which is the standard dialect for references to memory and buffers).

```
class MyTypeConverter : public mlir::TypeConverter {
public:
  MyTypeConverter() {
```

```
4         addConversion([](mlir::Type type) -> Optional<mlir::Type> {
5           if (type.isBF16()) {
6             // Map bfloat16 to i16 in the LLVM IR target dialect
7             return mlir::IntegerType::get(type.getContext(), 16);
8           }
9           return type; // No conversion needed for other types
10        });
11    }
12  };
```

Listing 8: Example of a simple lowering type converter

2. **Conversion Patterns**: These rewrite patterns dictate how operations of one dialect get mapped or transformed into operations of another dialect. For instance, consider a compiler that uses the tensor dialect and implements an addition operation, but uses LLVM IR as its low level intermediate representation for generating the back-end. LLVM IR doesn't directly support tensor operations. Through conversion patterns, the developer defines which set of LLVM IR operations are created when encountering with a tensor addition operation. For example, it might be broken down into element-wise additions using loops and scalar addition operations in LLVM IR.

```
1   class TensorAddToLLVMPattern : public OpRewritePattern<tensor::AddOp> {
2   public:
3     using OpRewritePattern<tensor::AddOp>::OpRewritePattern;
4
5     LogicalResult matchAndRewrite(tensor::AddOp op, PatternRewriter &rewriter) const override {
6       auto loc = op.getLoc();
7       auto lhs = op.getOperand(0);
8       auto rhs = op.getOperand(1);
9
10      // Check if the LLVM function already exists
11      auto llvmFunc = rewriter.getNamedFunction("rtt_tensor_add");
12      //If it doesn't exist, then create this function
13      if (!llvmFunc) {
14        auto llvmFuncType = LLVM::LLVMFunctionType::get(rewriter.getF32Type(), {rewriter.getF32Type
              (), rewriter.getF32Type()}, /*isVarArg=*/false);
15        llvmFunc = rewriter.create<LLVM::LLVMFuncOp>(loc, "rtt_tensor_add", llvmFuncType);
16        /*
17          Create pointer and loop instructions
18          ...
19        */
20
21      }
22
23      // Replace the original tensor::AddOp with a call to the LLVM function
24      rewriter.replaceOpWithNewOp<LLVM::CallOp>(op, llvmFunc, lhs, rhs);
25      return success();
26    }
27  };
```

Listing 9: Example of a a rewrite pattern for the tensor addition operation

3. **Conversion Targets**: These specify the target (or targets) dialects and constraints on the IR during the conversion process. This involves defining which operations or dialects should be considered **legal** or **illegal** after the conversion has taken place. For example, if you are converting

operations to the LLVM IR dialect, you would specify that after the conversion, only LLVM IR dialect operations should be considered, and the rest, illegal. These are enforced by **legalization passes**.

```
1
2   mlir::ConversionTarget target(getContext());
3   target.addLegalDialect<mlir::LLVM::LLVMDialect>();
4   target.addIllegalDialect<mlir::StandardOpsDialect, mlir::arith::ArithDialect, mlir::
        BuiltinDialect>();
5
6   // Mark specific operations as legal or illegal
7   target.addLegalOp<mlir::LLVM::AddOp>();
8   target.addIllegalOp<mlir::AddFOp>();
```

Listing 10: Example of a conversion target for a lowering

4. **Lowering passes**: Implement the overall passes that make use of type converters and conversion patterns to lower operations to a more concrete or hardware-specific form. This is basically defining how the lowering process should take place, iterating over the IR, deciding which conversion patterns to apply and using type converters to handle type mismatches. It basically acts as a wrapper of the 3 aforementioned concepts.

```
1   class LoweringPass : public mlir::PassWrapper<LoweringPass, mlir::OperationPass<mlir::ModuleOp>>
        {
2   public:
3     void runOnOperation() override {
4       mlir::ModuleOp module = getOperation();
5
6       mlir::ConversionTarget target(getContext());
7       MyTypeConverter typeConverter;
8
9       // Setup conversion target and legalizations
10      target.addLegalDialect<mlir::LLVM::LLVMDialect>();
11      target.addIllegalDialect<mlir::StandardOpsDialect>();
12
13      //Populate all the rewrite patterns
14      mlir::OwningRewritePatternList patterns(&getContext());
15      patterns.insert<TensorAddToLLVMPattern>(&getContext());
16
17      // Apply a full conversion
18      if (failed(mlir::applyFullConversion(module, target, std::move(patterns))))
19        signalPassFailure();
20    }
21  };
```

Listing 11: Example of a conversion target for a lowering

5. **Legalization Passes**: These ensure that all the operations in the intermediate representation conform to the constraints of the target dialect, using conversion targets. They check for the integrity of the produced IR and try to amend possible issues if possible, or report errors otherwise. For example, a typical legalization pass would be to check if there is any illegal tensor addition operation on resulting IR which has been lowered to LLVM IR dialect. These passes are hugely

similar to lowering passes, but the main difference resides on the use of partial conversions, where only a subset of operations is targeted for conversion, contrary to full conversions where it's assumed that all operations have a mapping and must be converted.

This is the great tradeoff of using MLIR, it adds a lot of reusability, extensibility, and modularity at the cost of introducing the abstraction of dialect conversion. Note that the existence of lowerings also implies the existence of **Liftings** (Commonly used on the process of decompilation).

MLIR is a powerful but complex tool to learn, and due to the size constraints of this thesis, further intricate topics will not be explained in detail. However, for readers who are keen on deepening their understanding, the author recommends referring to the comprehensive MLIR documentation [27]. Key areas to explore include bufferization, data layouts, diagnostic infrastructure, operation canonicalization, shape inference (deducing tensor shapes at compile time), and the definition, attachment, and use of operation interfaces.

13.4 ClangIR

ClangIR (CIR) is a novel intermediate representation introduced to the Clang community in 2022, as detailed in the Clang RFC document [49]. CIR is designed as a high-level IR for Clang, leveraging the powerful capabilities of MLIR (Multi-Level Intermediate Representation). The concept of CIR, however, was not unprecedented. When MLIR was launched by Google in 2019, Chris Lattner mentioned the need for a specific intermediate representation or language for Clang to enable further optimizations and enhance the Clang static analyzer. He referred to this hypothetical IR as CIL.

The inspiration for CIR drew from the positive results achieved by other high-level intermediate representations, such as Julia IR (for the Julia programming language) and MIR IR (for Rust). Although this approach might seem counterintuitive—considering LLVM's goal of unifying all IRs into one—creating a custom, ad-hoc IR can yield significant benefits by exploiting the specific features of a given language. This approach aims to maximize the performance and optimization potential, rather than settling for a generic but less effective solution such as plain LLVM IR (see Figure 18 for a visual representation of various MLIR dialects and their interactions).

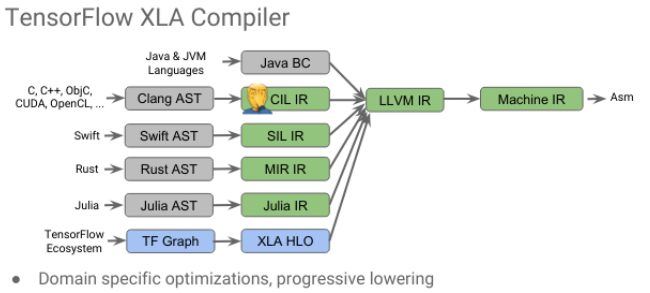

Figure 18: IR landscape in 2019, retrieved from [15]

The goal of CIR is to improve the compilation process providing an extensible, modular and high level IR, therefore allow a more detailed source-level information about the AST to be preserved. This enables better diagnostics for correctness, security and performance, which is a huge pain point about C++, given the freedom of memory management that this language offers, which a lot of times turns into memory unsafety. Around 70% of high severity security bugs at Google chromium are memory unsafety problems, which half of those are related with use-after-free bugs [50]. Not only that but the source location metadata that can be found on every single instruction of CIR, makes debugging easier and more complete. Take the following example of a possible memory management bug that modern compilers cannot catch without special passes, could be easily solved using CIR and its lifetime check pass (retrieved from [49]):

```
1
2  int *may_explode() {
3    int *p = nullptr;
4    {
5      int x = 0;
6      p = &x;
7      *p = 42;
```

```
  }
  *p = 42; // oops... (Explodes)
  ...
}

func @may_explode() -> !cir.ptr<i32> {
  %p_addr = cir.alloca !cir.ptr<i32>, cir.ptr <!cir.ptr<i32>>, ["p", cinit]
  ...
  cir.scope {
    // int x = 0;
    %x_addr = cir.alloca i32, cir.ptr <i32>, ["x", cinit]
    ...
    // p = &x;
    cir.store %x_addr, %p_addr : !cir.ptr<i32>, cir.ptr <!cir.ptr<i32>>
    ...
    // *p = 42
    cir.store %forty_two, %x_addr : i32, cir.ptr <i32>
    %p = cir.load deref %p_addr : cir.ptr <!cir.ptr<i32>>, !cir.ptr<i32>
    ...
  } // 'x' lifetime ends, 'p' is bad.

  // *p = 42
  %forty_two = cir.cst(42 : i32)
  %dead_x_addr = cir.load deref %p_addr : cir.ptr <!cir.ptr<i32>>, !cir.ptr<i32>

  // attempt to store 42 to the dead address
  cir.store %forty_two, %dead_x_addr : i32, cir.ptr <i32>
}
```

Listing 12: Dangling pointer error diagnosed by CIR

Even more importantly, this high level semantics facilitate more sophisticated optimization and analyses passes. For example, unintended and expensive C++ copies could be diagnosed on an analysis pass. Moreover, there is potential to implement cross-translation unit analysis, enabling improving diagnostics and catching bugs when considering multiple source files and not just files in isolation. Finally, enhanced loop optimizations and semantic aware aggressive optimizations are enabled thanks to CIR. However, given the early stage of CIR, the emphasis is not on implementing said optimizations or passes, but it paves the way for the future, or possible compiler researchers that might join the community. Analogously to other sections 13.3.1, a basic simplified example of CIR code is presented:

```
module @"/path/to/tutorial.c" {
  cir.func @add(%a: !cir.float loc("tutorial.c":1:13), %b: !cir.float loc("tutorial.c":1:23)) -> !cir.float
       loc("tutorial.c":1:5) {
    %0 = cir.alloca !cir.float, ["a"] loc("tutorial.c":1:13)
    %1 = cir.alloca !cir.float, ["b"] loc("tutorial.c":1:23)
    cir.store %a, %0 : !cir.float loc("tutorial.c":1:13)
    cir.store %b, %1 : !cir.float loc("tutorial.c":1:23)
    %2 = cir.load %0 : cir.ptr<!cir.float> loc("tutorial.c":2:12)
    %3 = cir.load %1 : cir.ptr<!cir.float> loc("tutorial.c":2:16)
    %4 = cir.binop(add, %2, %3) : !cir.float loc("tutorial.c":2:20)
    cir.return %4 : !cir.float loc("tutorial.c":2:5)
  }

  cir.func @main() -> !cir.int<32> loc("tutorial.c":5:1) {
```

```
    %0 = cir.alloca !cir.int<32>, ["__retval"] loc("tutorial.c":5:1)
    %1 = cir.alloca !cir.float, ["x"] loc("tutorial.c":6:5)
    %2 = cir.alloca !cir.float, ["y"] loc("tutorial.c":7:5)
    %3 = cir.const(#cir.fp<3.14>) : !cir.float loc("tutorial.c":6:15)
    %4 = cir.const(#cir.fp<420.0>) : !cir.float loc("tutorial.c":7:15)
    cir.store %3, %1 : !cir.float loc("tutorial.c":6:15)
    cir.store %4, %2 : !cir.float loc("tutorial.c":7:15)
    %5 = cir.load %1 : cir.ptr<!cir.float> loc("tutorial.c":9:16)
    %6 = cir.load %2 : cir.ptr<!cir.float> loc("tutorial.c":9:18)
    %7 = cir.call @add(%5, %6) : (!cir.float, !cir.float) -> !cir.float loc("tutorial.c":9:12)
    %8 = cir.cast(float_to_int, %7 : !cir.float), !cir.int<32> loc("tutorial.c":9:12)
    cir.store %8, %0 : !cir.int<32> loc("tutorial.c":9:5)
    %9 = cir.load %0 : cir.ptr<!cir.int<32>> loc("tutorial.c":9:5)
    cir.return %9 : !cir.int<32> loc("tutorial.c":9:5)
  }
}
```

Listing 13: Simplified sample of CIR code

It's crucial to grasp how ClangIR alters the compilation workflow (see Figure 5), shifting the code generation to CIR instead of LLVM IR when the flag is enabled. After performing the code generation, CIR specific optimizations, transformations, and analyses can take place. After performing those steps the lowering phase begins, which can follow two possible paths: direct lowering to LLVM IR or lowering to MLIR, where multiple dialects can intermix, including the OpenMP dialect in our case. Ultimately, everything is lowered to LLVM IR, which then assumes responsibility for the back-end.

Despite the current development of ClangIR it already counts with numerous tools like **cir-opt**, and **cir-translate** similar to MLIR, in addition to the support that Clang already provides with various flags (-fclangir -emit-cir...)

More information on how to set up and compile CIR locally, and compile object files afterward using this custom compiler can be found on the appendix A.

14 Practical Work

14.1 OpenMP Specification and coverage

Before introducing any details about the actual implementation, the reader must be at least familiar with the directives and their respective clauses (Which allow the user to make use of all the capabilities of that directive). Even though a brief description was given on the LLVM subsection (1.3.2) as an introduction, part of the specification of OpenMP 4.5 will be presented to understand the implementation details fully (14.1).

OpenMP is a multi-threading API based on the fork-join execution model (19), therefore it must establish data sharing policies to manage how variables and data is used and shared across threads:

- **Shared**: These variables belong to a shared memory region accessible and modifiable by every thread. However, without synchronization mechanisms such as locks, mutexes, or atomic instructions, data race conditions may occur.
- **Private**: Data that belongs to each thread during its lifetime and cannot be accessed or modified by other threads. Typically, private data resides on the stack of each thread.
- **First-Private**: Private data initialized in a thread using the same value as another variable that shares its symbol. For instance, if variable X exists with a value of 4, then a private variable X with a value of 4 will be created inside the new thread.
- **Last-Private**: A private variable that shares its final value with another variable that shares its symbol in an upper scope. This is particularly useful for obtaining the last value of variables after loops, for example.

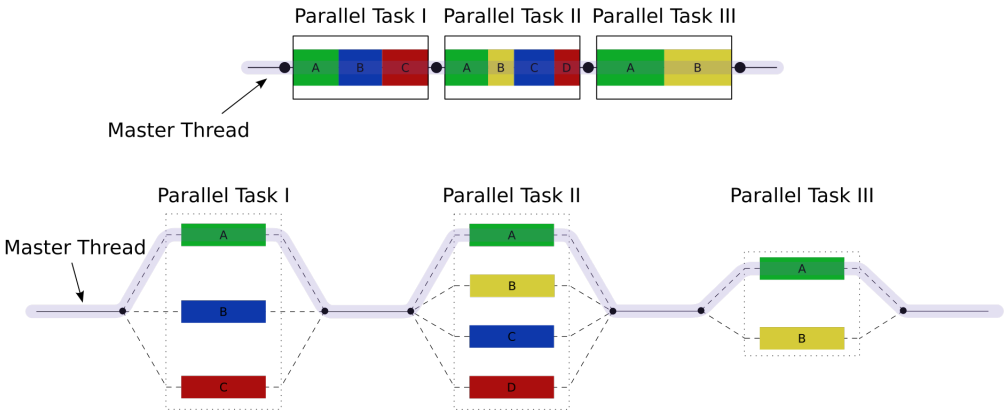

Figure 19: Fork-join model of execution, Retrieved from [51]

- **Task** is a construct which creates regions of code that can be executed concurrently if no restrictions or synchronizations interferes. The execution of tasks can be delayed, this means that it can be executed later on and not necessarily immediately after its creation. Tasks can have a certain priority (Priority clause) over other tasks, its execution can be blocked using the "if" clause and for further control of logic in the control flow, tasks can also be marked as "final" using the final clause.

Default data-sharing can be modified through the default clause (Shared if not specified), and data-sharing are specified using the clauses "private", "shared" and "firstprivate". Also tasks can be marked as untied or mergeable and finally, can enforce synchronization mechanisms making use of the "depend" clause, which can take values such as "in", "out" or "inout" to specify the direction of the dependency. Following, the specification of the directive and its implemented clauses are presented, where check-mark means implemented clause and red cross, not yet implemented:

```
#pragma omp task[clause[ [,] clause] ... ][clause[ [,] clause] ... ]
```

where *clause* is one of the following:

- `if([ task :] scalar-expression)` ✓
- `final(scalar-expression)` ✓
- `untied` ✓
- `default(shared  none)`| X
- `mergeable` ✓
- `private(list)` X
- `firstprivate(list)` X
- `shared(list)` ✓
- `depend(dependence-type : list)` X
- `priority(priority-value)` ✓

- The **Taskgroup** directive creates a region where all tasks are executed as a single task, facilitating task grouping, synchronization, and explicit task hierarchies. When a thread encounters a task group directive, it suspends its execution and proceeds to execute every task within that region, including children tasks and their descendants. The syntax is:

```
#pragma omp taskgroup
structured-block
```

- The **Taskwait** directive provides task synchronization by instructing the current thread to wait until previously created child tasks have completed execution. Unlike TASKGROUP, it only affects the completion of existing child tasks and does not impact future tasks that may be created. The syntax is:

```
#pragma omp taskwait
```

- **Taskyield** is also a synchronization tool that allow the current task to suspend in favor of execution of a different task. The syntax is:

```
#pragma omp taskyield
```

- **Barrier** is one of the most used synchronization directives of OpenMP, given that it allows to suspend execution of various threads until every single of them has reached the barrier directive, making sure that all threads are at the same point before moving forward the execution. The syntax is:

```
#pragma omp barrier
```

- **Critical** ensures that a section of code is executed by only one thread at a time, preventing data races. When a thread encounters a critical section, it will wait until no other thread is executing that section before proceeding. This is useful for protecting shared resources. The syntax is:

```
#pragma omp critical [clause[ [,] clause] ... ]
structured block
```

  where clause can be one of the following

  - `name` ✗
  - `hint(Expr)` ✗

- **Master** specifies a section of code that is to be executed only by the master thread (the thread that encountered the parallel region). This is useful for tasks that should be performed once, like initialization. The other threads will skip this section of code. The syntax is:

```
#pragma omp master
structured block
```

- **Single** ensures that a section of code is executed by only one thread within a team, but unlike the MASTER directive, it is not limited to the master thread. The first thread that reaches this directive will execute the enclosed code, while the other threads will wait at an implicit barrier unless a nowait clause is specified. The syntax is:

```
#pragma omp single [clause[ [,] clause] ... ]
structured block
```

  where cluase can be one of the following:

  - `private(list)` ✗
  - `firstprivate(list)` ✗
  - `copyprivate(list)` ✗
  - `nowait` ✓

It's worth mentioning that although taskloop didn't make it to the final version of the thesis, some of its clauses (Grainsize ✓and NumTasks ✓) have been implemented in the ClauseProcessor class.

14.2 Justification

A brief discussion of the rationale behind this implementation will be given on this section, in particular the reason behind the choice of the specific directives. Firstly, given the early development stage of OpenMP inside of CIR, it was crucial to address its initial limitations, particularly the fact that it could only create shared parallel regions. Secondly, tasks are favored by the author, Roger Ferrer, and BSC in general, being extensively used. Thus, supporting them felt natural. Lastly, it's widely acknowledged that most OpenMP users prefer the most popular and straightforward directives, such as parallel, single, task, and synchronization constructs, which are also the most relevant and useful. Therefore, this implementation was far more functional and practical that say for example, simd, cancellation or teams niche directives.

In addition to implementing the main 3 directives (task, taskwait, taskgroup), the author opted to include the barrier and taskyield directives for their relevance and ease of implementation. This decision ensured a well-rounded solution, enhancing the completeness and usability of the implementation of the OpenMP framework within ClangIR. Later on, the author also had the opportunity to implement the critical, master and single directives, due to the symmetry of implementation.

14.2.1 Evolution over time

This section seeks to answer the most relevant question that may arise to the reader, upon the process of understanding this thesis, which is: **Why do OpenMP needs to be implemented again**? It's a valid inquiry, considering that OpenMP has already been implemented in numerous open-source and commercial compilers. Moreover, the evolution and primary decisions of the implementation are discussed and elaborated.

One can infer that by changing the compilation flow, by adding a new code generation (CIR) phase which diverges with the old classic one (to LLVM-IR), therefore a brand-new code generation to the OpenMP related AST nodes should be implemented. Luckily, there are several ways of doing so, looking at how currently clang is handling OpenMP:

In the context of OpenMP - LLVM IR code generation, Clang currently supports two main approaches after the parsing and semantic analysis of OpenMP nodes [52]:

1. **Basic Code Generation Using Pre-Built OpenMP Runtime Routines:** This method utilizes pre-existing runtime libraries, such as `libomp.so`, which are written in plain LLVM IR. This approach is considered the more traditional and rudimentary method for generating OpenMP code.

2. **Code Generation Through the OpenMPIRBuilder Class:** Introduced in LLVM 10.0 (released on March 24, 2020), the OpenMPIRBuilder is a more modern and standardized approach. This tool was developed by a collaboration of various LLVM sub-projects, including Clang, Flang, MLIR, and OpenMP, to streamline the creation of OpenMP infrastructure. The goal of the OpenMPIRBuilder is to provide a front-end agnostic interface for generating OpenMP LLVM IR routines, reducing redundancy and lowering development time for future compilers that want to integrate OpenMP. This approach is particularly significant for MLIR, which implements all operations in the OpenMP dialect using this builder.

The primary objective of this initiative was to avoid the extensive code churn associated with implementing OpenMP across multiple compiler front-ends and to standardize front-end code-bases, primarily Clang and Flang. Considering the vast scope of the LLVM project, this standardization effort helps mitigate potential divergences and the need for significant refactors over time.

Furthermore, there is another possibility for implementing OpenMP, given the nature of ClangIR and its close relationship with MLIR: blending CIR with the OpenMP dialect, thereby reusing the MLIR and OpenMPIRBuilder infrastructure. Despite how natural this solution appears, it clearly involves complexities such as handling type conversions between dialects and various compatibility issues associated with mixing dialects. It's worth mentioning that the OpenMP dialect is a work in progress. While it covers a significant portion of OpenMP, several clauses, features, and aspects like data-sharing remain to be implemented, thus not yet fully utilizing the capabilities of the underlying OpenMPIRBuilder.

After deliberating and considering all possibilities, the author and Fabian Mora (PhD candidate at the University of Delaware) [53], who contributed to and implemented parts of OpenMP within ClangIR, concluded that the most natural, agile, and modern approach would be to integrate CIR with the OpenMP dialect of MLIR, as Flang had already done. Fabian developed the initial skeleton files for OpenMP within CIR's code generation and a first simple implementation of the **parallel** directive, while the author focused on task and synchronization directives and improving the initial code-base provided by Fabian, as previously mentioned. With the guidance of Bruno Cardoso, one of the main tech leads of Clang responsible for CIR development at Meta [54], and Kiran [55], they were advised through issues and pull requests to replicate the exact control flow within ClangIR's code generation, providing a framework for anyone attempting to implement other approaches.

This approach can be considered the right implementation not only for its exploratory effort and as setting an example of intermixing dialects (which is essential at some point in CIR) but also for the simplicity and maintainability of the developed code. Moreover, relying on the robust MLIR infrastructure, supported by a team of full-time engineers, is a wiser decision than creating extensive code generation functions that depend on two students and may require changes over time. This approach maximizes resilience over time.

14.3 Implementation

Now that the reader has been familiarized and has a great scope with modern compilers and Clang, we can proceed to the actual, particular details of implementation. The first thing one should note is that the vast majority of the implementation relies on following strictly the original Clang's OpenMP code generation, trying to imitate the program execution flow as much as possible. To enhance readability, this section is divided into subsections, each grouped by similarity of implementation or theme. For further information, consider this branch of the CIR fork [56].

14.3.1 Simple directives

Firstly, the 3 simple directives were implemented: taskwait, taskyield and barrier, which are almost symmetrical. The reason was that given that these directives had no body. To introduce these code generation functions, the huge switch inside of **buildStmt** in **CIRGenStmt.cpp** was modified so instead of emitting an error message when trying to build the statements **OMPTaskwaitDirective**, **OMP-**

TaskyieldDirective and **OMPBarrierDirective** it now calls their respective code generation functions: **buildOMPTaskwaitDirective**, **buildOMPTaskyieldDirective** and **buildOMPBarrierDirective**. To avoid redundancy, all the symmetric changes made to CIRGenFunction.h and CIRGenStmt are presented below:

```

//CIRGenFunction.h
//Declaration of the code generation functions
  mlir::LogicalResult buildOMPTaskwaitDirective(const OMPTaskwaitDirective &S);
  mlir::LogicalResult buildOMPTaskyieldDirective(const OMPTaskyieldDirective &S);
  mlir::LogicalResult buildOMPBarrierDirective(const OMPBarrierDirective &S);
  mlir::LogicalResult buildOMPTaskDirective(const OMPTaskDirective &S);
  mlir::LogicalResult buildOMPTaskgroupDirective(const OMPTaskgroupDirective &S);
  mlir::LogicalResult buildOMPCriticalDirective(const OMPCriticalDirective &S);
  mlir::LogicalResult buildOMPMasterDirective(const OMPMasterDirective &S);
  mlir::LogicalResult buildOMPSingleDirective(const OMPSingleDirective &S);

//CIRGenStmt.h
//Adding new statements to the generic buildStmt switch, given a statement S

  case Stmt::OMPTaskwaitDirectiveClass:
    return buildOMPTaskwaitDirective(cast<OMPTaskwaitDirective>(*S));
  case Stmt::OMPTaskyieldDirectiveClass:
    return buildOMPTaskyieldDirective(cast<OMPTaskyieldDirective>(*S));
  case Stmt::OMPBarrierDirectiveClass:
    return buildOMPBarrierDirective(cast<OMPBarrierDirective>(*S));
  case Stmt::OMPTaskDirectiveClass:
    return buildOMPTaskDirective(cast<OMPTaskDirective>(*S));
  case Stmt::OMPTaskgroupDirectiveClass:
    return buildOMPTaskgroupDirective(cast<OMPTaskgroupDirective>(*S));
  case Stmt::OMPMasterDirectiveClass:
    return buildOMPMasterDirective(cast<OMPMasterDirective>(*S));
  case Stmt::OMPSingleDirectiveClass:
    return buildOMPSingleDirective(cast<OMPSingleDirective>(*S));
  case Stmt::OMPCriticalDirectiveClass:
    return buildOMPCriticalDirective(cast<OMPCriticalDirective>(*S));
```

Listing 14: Changes to CIRGenFunction and CIRGenStmt

Furthermore, these functions needed to be declared within CIRGenFunction.h, the file that declares all the AST-to-CIR translation functions. With these modifications in place, the next step is to implement the actual code generation behaviour. This involves the materialization of the build functions, responsible for generating IR or invoking runtime calls. So for these kinds of directives which make use of OpenMPRuntime calls, it's mandatory to define the behaviour of these functions in addition to the OMPTaskDataTy. This object helps to store helpful information and attributes for various task-based directives, like its dependencies, if the task is untied, is a final task, its priority... Finally, it's important to ackownledge that these IR is being created thanks to the polymorphic **builder** object, which creates MLIR instructions such as omp.taskwait, omp.taskyield and omp.barrier leveraging the aforementioned OpenMPIRBuilder, then there is no need to re implement these complex instructions or runtime calls using CIR operations.

```
//CIRGenStmtOpenMP.cpp
static void buildDependences(const OMPExecutableDirective &S,
```

```
                          OMPTaskDataTy &Data) {
  // First look for 'omp_all_memory' and add this first.
  bool OmpAllMemory = false;
  if (llvm::any_of(
          S.getClausesOfKind<OMPDependClause>(), [](const OMPDependClause *C) {
            return C->getDependencyKind() == OMPC_DEPEND_outallmemory ||
                   C->getDependencyKind() == OMPC_DEPEND_inoutallmemory;
          })) {
    OmpAllMemory = true;
    // Since both OMPC_DEPEND_outallmemory and OMPC_DEPEND_inoutallmemory are
    // equivalent to the runtime, always use OMPC_DEPEND_outallmemory to
    // simplify.
    OMPTaskDataTy::DependData &DD =
        Data.Dependences.emplace_back(OMPC_DEPEND_outallmemory,
                                      /*IteratorExpr=*/nullptr);
    // Add a nullptr Expr to simplify the codegen in emitDependData.
    DD.DepExprs.push_back(nullptr);
  }
  // Add remaining dependences skipping any 'out' or 'inout' if they are
  // overridden by 'omp_all_memory'.
  for (const auto *C : S.getClausesOfKind<OMPDependClause>()) {
    OpenMPDependClauseKind Kind = C->getDependencyKind();
    if (Kind == OMPC_DEPEND_outallmemory || Kind == OMPC_DEPEND_inoutallmemory)
      continue;
    if (OmpAllMemory && (Kind == OMPC_DEPEND_out || Kind == OMPC_DEPEND_inout))
      continue;
    OMPTaskDataTy::DependData &DD =
        Data.Dependences.emplace_back(C->getDependencyKind(), C->getModifier());
    DD.DepExprs.append(C->varlist_begin(), C->varlist_end());
  }
}

mlir::LogicalResult
CIRGenFunction::buildOMPTaskwaitDirective(const OMPTaskwaitDirective &S) {
  mlir::LogicalResult res = mlir::success();
  OMPTaskDataTy Data;
  buildDependences(S, Data);
  Data.HasNowaitClause = S.hasClausesOfKind<OMPNowaitClause>();
  CGM.getOpenMPRuntime().emitTaskWaitCall(builder, *this,
                                          getLoc(S.getSourceRange()), Data);
  return res;
}
mlir::LogicalResult
CIRGenFunction::buildOMPTaskyieldDirective(const OMPTaskyieldDirective &S) {
  mlir::LogicalResult res = mlir::success();
  // Creation of an omp.taskyield operation
  CGM.getOpenMPRuntime().emitTaskyieldCall(builder, *this,
                                           getLoc(S.getSourceRange()));
  return res;
}

mlir::LogicalResult
CIRGenFunction::buildOMPBarrierDirective(const OMPBarrierDirective &S) {
  mlir::LogicalResult res = mlir::success();
  // Creation of an omp.barrier operation
  CGM.getOpenMPRuntime().emitBarrierCall(builder, *this,
```

```
59                                        getLoc(S.getSourceRange()));
60   return res;
61 }
62
63 //CIRGenOpenMPRuntime.cpp
64 void CIRGenOpenMPRuntime::emitTaskWaitCall(CIRGenBuilderTy &builder,
65                                            CIRGenFunction &CGF,
66                                            mlir::Location Loc,
67                                            const OMPTaskDataTy &Data) {
68
69   if (!CGF.HaveInsertPoint())
70     return;
71
72   if (CGF.CGM.getLangOpts().OpenMPIRBuilder && Data.Dependences.empty()) {
73     // TODO(cir): This could change in the near future when OpenMP 5.0 gets
74     // supported by MLIR
75     builder.create<mlir::omp::TaskwaitOp>(Loc);
76   } else {
77     llvm_unreachable("NYI");
78   }
79   assert(!MissingFeatures::openMPRegionInfo());
80 }
81
82 void CIRGenOpenMPRuntime::emitBarrierCall(CIRGenBuilderTy &builder,
83                                           CIRGenFunction &CGF,
84                                           mlir::Location Loc) {
85
86   assert(!MissingFeatures::openMPRegionInfo());
87
88   if (CGF.CGM.getLangOpts().OpenMPIRBuilder) {
89     builder.create<mlir::omp::BarrierOp>(Loc);
90     return;
91   }
92
93   if (!CGF.HaveInsertPoint())
94     return;
95
96   llvm_unreachable("NYI");
97 }
98
99 void CIRGenOpenMPRuntime::emitTaskyieldCall(CIRGenBuilderTy &builder,
100                                            CIRGenFunction &CGF,
101                                            mlir::Location Loc) {
102
103   if (!CGF.HaveInsertPoint())
104     return;
105
106   if (CGF.CGM.getLangOpts().OpenMPIRBuilder) {
107     builder.create<mlir::omp::TaskyieldOp>(Loc);
108   } else {
109     llvm_unreachable("NYI");
110   }
111
112   assert(!MissingFeatures::openMPRegionInfo());
113 }
```

Listing 15: Code generation for Taskwait, Taskyield and Barrier

The mlir::LogicalResult is a boolean-like data type used in MLIR to indicate the success or failure of various operations, such as building operations, performing conversions, and applying optimizations. A successful operation returns **mlir::success()**, while a failed one returns **mlir::failure()**. This consistent use of mlir::LogicalResult throughout MLIR helps standardize error handling and makes it easier to compose and chain operations.

Locations, often referred to as Locs, are semantic annotations attached to instructions. These annotations provide valuable debugging information by associating instructions with their source code locations. This helps developers trace errors back to their origin in the source code, making it easier to understand and fix issues, therefore locations are **crucial** in compiler development. In the context of CIR, locations are extensively used to enhance debuggability and maintainability. By embedding source locations directly into the IR, CIR allows for more effective debugging and error reporting.

14.3.2 Directives with captured statements

Following up, an explanation of more complex directives will be given, namely: **Taskgroup**, **Single**, **Master**, **Critical** and **Task**. These directives share one trait in common: They possess some sort of associated/captured/inner statement.

The implementation of all of these directives is almost identical, given that the behaviour is the same: Handling the clauses (Explained in the following section), generating the corresponding OMP dialect operations using the builder object, creating a scope operation (One powerful abstraction of CIR is that scopes are considered operations), recursively building the inner statement inside the scope and finally adding a terminator operation to delimit the end of that scope. The comments on the code snippet below help to give a high-level overview of what these code generation functions are doing (14.3.2).

```
1  mlir::LogicalResult
2  CIRGenFunction::buildOMPTaskgroupDirective(const OMPTaskgroupDirective &S) {
3    mlir::LogicalResult res = mlir::success();
4    auto scopeLoc = getLoc(S.getSourceRange());
5    bool useCurrentScope = true;
6    //Clause handling
7    mlir::omp::TaskgroupClauseOps clauseOps;
8    CIRClauseProcessor cp = CIRClauseProcessor(*this, S);
9    cp.processTODO<clang::OMPTaskReductionClause, clang::OMPAllocateClause>();
10   //Generation of taskgroup op
11   mlir::omp::TaskgroupOp taskgroupOp =
12       builder.create<mlir::omp::TaskgroupOp>(scopeLoc, clauseOps);
13   //Getting the captured statement
14   const Stmt *capturedStmt = S.getInnermostCapturedStmt()->getCapturedStmt();
15   mlir::Block &block = taskgroupOp.getRegion().emplaceBlock();
16   mlir::OpBuilder::InsertionGuard guardCase(builder);
17   builder.setInsertionPointToEnd(&block);
18   // Create a scope for the OpenMP region.
19   builder.create<mlir::cir::ScopeOp>(
20       scopeLoc, /*scopeBuilder=*/
21       [&](mlir::OpBuilder &b, mlir::Location loc) {
```

```
22        LexicalScope lexScope{*this, scopeLoc, builder.getInsertionBlock()};
23        // Emit the body of the region.
24        if (buildStmt(capturedStmt, useCurrentScope).failed())
25          res = mlir::failure();
26      });
27  //Add a terminator op to delimit the scope
28  builder.create<TerminatorOp>(getLoc(S.getSourceRange().getEnd()));
29  return res;
30 }
31 mlir::LogicalResult
32 CIRGenFunction::buildOMPCriticalDirective(const OMPCriticalDirective &S) {
33  mlir::LogicalResult res = mlir::success();
34  auto scopeLoc = getLoc(S.getSourceRange());
35  // WIP: named critical regions still not supported
36  mlir::FlatSymbolRefAttr refAttr;
37  //Generation of critical op
38  mlir::omp::CriticalOp criticalOp =
39      builder.create<mlir::omp::CriticalOp>(scopeLoc, refAttr);
40  //Getting the captured statement
41  const Stmt *capturedStmt = S.getAssociatedStmt();
42  mlir::Block &block = criticalOp.getRegion().emplaceBlock();
43  mlir::OpBuilder::InsertionGuard guardCase(builder);
44  builder.setInsertionPointToEnd(&block);
45  // Build an scope for the critical region
46  builder.create<mlir::cir::ScopeOp>(
47      scopeLoc, /*scopeBuilder=*/
48      [&](mlir::OpBuilder &b, mlir::Location loc) {
49        LexicalScope lexScope{*this, scopeLoc, builder.getInsertionBlock()};
50        // Emit the statement within the critical region
51        if (buildStmt(capturedStmt, /*useCurrentScope=*/true).failed())
52          res = mlir::failure();
53      });
54  //Add a terminator op to delimit the scope
55  builder.create<TerminatorOp>(getLoc(S.getSourceRange().getEnd()));
56  return res;
57 }
58
59 mlir::LogicalResult
60 CIRGenFunction::buildOMPMasterDirective(const OMPMasterDirective &S) {
61  mlir::LogicalResult res = mlir::success();
62  auto scopeLoc = getLoc(S.getSourceRange());
63  //Generation of master op
64  mlir::omp::MasterOp masterOp = builder.create<mlir::omp::MasterOp>(scopeLoc);
65  //Getting the captured statement
66  const Stmt *capturedStmt = S.getAssociatedStmt();
67
68  mlir::Block &block = masterOp.getRegion().emplaceBlock();
69  mlir::OpBuilder::InsertionGuard guardCase(builder);
70  builder.setInsertionPointToEnd(&block);
71  // Build an scope for the master region
72  builder.create<mlir::cir::ScopeOp>(
73      scopeLoc, /*scopeBuilder=*/
74      [&](mlir::OpBuilder &b, mlir::Location loc) {
75        LexicalScope lexScope{*this, scopeLoc, builder.getInsertionBlock()};
76        // Emit the statement within the master region
77        if (buildStmt(capturedStmt, /*useCurrentScope=*/true).failed())
```

```
78          res = mlir::failure();
79        });
80    //Add a terminator op to delimit the scope
81    builder.create<TerminatorOp>(getLoc(S.getSourceRange().getEnd()));
82    return res;
83  }
84
85  mlir::LogicalResult
86  CIRGenFunction::buildOMPSingleDirective(const OMPSingleDirective &S){
87    mlir::LogicalResult res = mlir::success();
88    auto scopeLoc = getLoc(S.getSourceRange());
89
90    //WIP: treatment of single clauses
91    //Clause handling
92    mlir::omp::SingleClauseOps clauseOps;
93    CIRClauseProcessor cp = CIRClauseProcessor(*this, S);
94    cp.processNowait(clauseOps);
95    //Generation of taskgroup op
96    auto singleOp = builder.create<mlir::omp::SingleOp>(scopeLoc, clauseOps);
97    //Getting the captured statement
98    const Stmt* capturedStmt = S.getInnermostCapturedStmt()->getCapturedStmt();
99
100   mlir::Block& block = singleOp.getRegion().emplaceBlock();
101   mlir::OpBuilder::InsertionGuard guardCase(builder);
102   builder.setInsertionPointToEnd(&block);
103
104     // Build an scope for the single region
105   builder.create<mlir::cir::ScopeOp>(
106       scopeLoc, /*scopeBuilder=*/
107       [&](mlir::OpBuilder &b, mlir::Location loc) {
108         LexicalScope lexScope{*this, scopeLoc, builder.getInsertionBlock()};
109         // Emit the statement within the single region
110         if (buildStmt(capturedStmt, /*useCurrentScope=*/true).failed())
111           res = mlir::failure();
112       });
113   //Add a terminator op to delimit the scope
114   builder.create<TerminatorOp>(getLoc(S.getSourceRange().getEnd()));
115   return res;
116 }
117
118 mlir::LogicalResult
119 CIRGenFunction::buildOMPTaskDirective(const OMPTaskDirective &S) {
120   mlir::LogicalResult res = mlir::success();
121   auto scopeLoc = getLoc(S.getSourceRange());
122   OMPTaskDataTy data;
123   data.Tied = not S.getSingleClause<clang::OMPUntiedClause>();
124   clang::OpenMPDirectiveKind DKind = clang::OpenMPDirectiveKind::OMPD_task;
125   bool useCurrentScope = true;
126   // Clause handling
127   mlir::omp::TaskClauseOps clauseOps;
128   CIRClauseProcessor cp = CIRClauseProcessor(*this, S);
129   cp.processUntied(clauseOps);
130   cp.processMergeable(clauseOps);
131   cp.processFinal(clauseOps);
132   cp.processIf(clauseOps);
133   cp.processPriority(clauseOps);
```

```
134   // TODO(cir) Give support to this OpenMP v.5 clauses
135   cp.processTODO<clang::OMPAllocateClause, clang::OMPInReductionClause,clang::OMPAffinityClause, clang::
          OMPDetachClause,clang::OMPDefaultClause, clang::OMPDependClause>();
136
137   // Create a `omp.task` operation
138   mlir::omp::TaskOp taskOp = builder.create<mlir::omp::TaskOp>(scopeLoc, clauseOps);
139   //Getting the captured statement
140   const Stmt* capturedStmt = S.getCapturedStmt(DKind)->getCapturedStmt();
141
142   mlir::Block& block = taskOp.getRegion().emplaceBlock();
143   mlir::OpBuilder::InsertionGuard guardCase(builder);
144   builder.setInsertionPointToEnd(&block);
145
146   // Create a scope for the OpenMP region.
147   builder.create<mlir::cir::ScopeOp>(
148       scopeLoc, /*scopeBuilder=*/
149       [&](mlir::OpBuilder &b, mlir::Location loc) {
150         LexicalScope lexScope{*this, scopeLoc, builder.getInsertionBlock()};
151         // Emit the body of the region.
152         if (buildStmt(capturedStmt, /*useCurrentScope=*/true).failed())
153           res = mlir::failure();
154       });
155   //Add a terminator op to delimit the scope
156   builder.create<TerminatorOp>(getLoc(S.getSourceRange().getEnd()));
157   return res;
158 }
```

Listing 16: Code generation of directives with body

Note that some of these directives are still being uploaded to upstream through pull requests, therefore the runtime calls haven't been yet implemented.

14.3.3 Clause Handling

Following up, the logic of clause handling is presented. When the author started implementing clauses on the `task` directive, the initial idea was to implement the clause handling logic for each single directive. After programming the code generation for a couple of clauses, the author noticed a pattern: clauses exhibited repetitive behavior, and the auxiliary procedures were identical. Following this approach, the code generation functions were becoming large too quickly. Thus, the thought of utilizing the Command Design Pattern emerged. This pattern allowed for the complete decoupling of clause processing and directive code generation, reducing the number of implementations from $O(d \cdot c)$ to $O(c)$.

This clause handling logic was directly mapped inside the new **CIRClauseProcessor** class, which implements functions to facilitate the iteration through clauses within a directive. There is extensive use of templates to handle the polymorphic nature of the Clang-OMP AST nodes. Each clause has its own processing method (e.g., `processFinal`, `processUntied`) that returns true if the clause is found and stores the result in the corresponding `ClauseOps` result passed by reference, or returns false otherwise. The helper functions used to implement the process functions are the following:

```
1 template <typename C>
2 CIRClauseProcessor::ClauseIterator
3 CIRClauseProcessor::findClause(ClauseIterator begin, ClauseIterator end) {
```

```
4   for (ClauseIterator it = begin; it != end; ++it) {
5     const clang::OMPClause *clause = *it;
6     if (llvm::dyn_cast<const C>(clause)) {
7       return it;
8     }
9   }
10  return end;
11 }
12
13 template <typename C> const C *CIRClauseProcessor::findUniqueClause() const {
14   ClauseIterator it = findClause<C>(clauses.begin(), clauses.end());
15   if (it != clauses.end()) {
16     return llvm::dyn_cast<const C>(*it);
17   }
18   return nullptr;
19 }
20
21 template <typename C>
22 bool CIRClauseProcessor::markClauseOccurrence(mlir::UnitAttr &result) const {
23   if (findUniqueClause<C>()) {
24     result = this->CGF.getBuilder().getUnitAttr();
25     return true;
26   }
27   return false;
28 }
29
30 template <typename C>
31 bool CIRClauseProcessor::findRepeatableClause(
32     std::function<void(const C *)> callback) const {
33   bool found = false;
34   ClauseIterator next, end = clauses.end();
35   for (ClauseIterator it = clauses.begin(); it != end; it = next) {
36     next = findClause<C>(it, end);
37
38     if (next != end) {
39       callback(llvm::dyn_cast<const C>(*next));
40       found = true;
41       ++next;
42     }
43   }
44   return found;
45 }
46
47 template <typename... Cs> void CIRClauseProcessor::processTODO() const {
48   auto checkClause = [&](const clang::OMPClause *clause) {
49     if (clause) {
50       std::string error_msg = "The following clause is not yet implemented: " +
51                               getClauseName(clause).str();
52       llvm_unreachable(error_msg.c_str());
53     }
54   };
55
56   for (const clang::OMPClause *clause : clauses) {
57     (checkClause(llvm::dyn_cast<const Cs>(clause)), ...);
58   }
59 }
```

```
#endif
```

Listing 17: Helper functions of CIRClauseProcessor

`ClauseOps` are containers of the result values and attributes of a specific directive, providing a compact and clean way of handling result values. For example, `taskwaitClauseOps` contains `dependClauseOps` and `nowaitClauseOps`, which in turn contain the `mlir::Values`, `mlir::Attributes`, and other relevant clause information needed to build the operation. During the development of clause processing one issue arised, given the mixture of dialects, which can be seen on the code and the need to use UnrealizedConversionCast operations. More on this issue will be covered on the following section (14.3.3). Below we can find the processing of 3 sample clauses (14.3.3).

```
bool CIRClauseProcessor::processNowait(mlir::omp::NowaitClauseOps &result) const {
   return markClauseOccurrence<clang::OMPNowaitClause>(result.nowaitAttr);
}


bool CIRClauseProcessor::processIf(mlir::omp::IfClauseOps &result) const {
  const clang::OMPIfClause *clause = findUniqueClause<clang::OMPIfClause>();
  if (clause) {
    auto scopeLoc = this->CGF.getLoc(this->dirCtx.getSourceRange());
    const clang::Expr *ifExpr = clause->getCondition();
    mlir::Value ifValue = this->CGF.evaluateExprAsBool(ifExpr);
    mlir::ValueRange ifRange(ifValue);
    mlir::Type int1Ty = builder.getI1Type();
    result.ifVar = builder
                       .create<mlir::UnrealizedConversionCastOp>(
                           scopeLoc, /*TypeOut*/ int1Ty, /*Inputs*/ ifRange)
                       .getResult(0);
    return true;
  }
  return false;
}

bool CIRClauseProcessor::processGrainSize(
    mlir::omp::GrainsizeClauseOps &result) const {
  // Check existence of mutally exclusive clause num_tasks
  const clang::OMPNumTasksClause *tasksClause =
      findUniqueClause<clang::OMPNumTasksClause>();
  const clang::OMPGrainsizeClause *grainClause =
      findUniqueClause<clang::OMPGrainsizeClause>();
  if (tasksClause and grainClause) {
    // This should be replaced by a proper error and not llvm_unreachable
    // probably
    llvm_unreachable("error: 'num_tasks' and 'grainsize' clause are mutually "
                     "exclusive and may not appear on the same directive!");
  }
  if (grainClause) {
    auto scopeLoc = this->CGF.getLoc(dirCtx.getSourceRange());
    const clang::Expr *grainExpr = grainClause->getGrainsize();
    mlir::Value grainValue = this->CGF.buildScalarExpr(grainExpr);
    mlir::ValueRange grainRange(grainValue);
    mlir::Type uint32Ty = builder.getI32Type();
    result.grainsizeVar =
        builder
```

```
44            .create<mlir::UnrealizedConversionCastOp>(
45                scopeLoc, /*TypeOut*/ uint32Ty, /*Inputs*/ grainRange)
46            .getResult(0);
47    return true;
48  }
49  return false;
50 }
```

Listing 18: Processing of the clauses Nowait, If and GrainSize

Similar to the coverage report of that was given on the OpenMP specification (14), a list of all the supported clauses is presented (Even clauses that weren't used at all by any of the implemented directives).

1. If ✓
2. Final ✓
3. Priority ✓
4. Untied ✓
5. Mergeable ✓
6. Nowait ✓
7. Grainsize ✓
8. NumTasks✓
9. Nogroup ✓
10. Depend X

In section (17), the reasons behind the non-completion of this clause are discussed. However, the main code generation behavior for the `depend` clause can be summarized as follows: for each `OMPDependClause` node found in the AST, retrieve its variable list through the captured `DeclRefExpr` nodes, obtain a pointer from the `LValue` of the variable, and append them to the result `depend` variable list along with the dependency type (In, Out, or InOut).

```
1  bool CIRClauseProcessor::processDepend(mlir::omp::DependClauseOps &result,
2                                         cir::OMPTaskDataTy &data,
3                                         mlir::Location &location) const {
4   llvm_unreachable("The following clause is not yet implemented: Depend");
5   return findRepeatableClause<clang::OMPDependClause>(
6       // Get an mlir value for the pointer of each variable in the var list
7       [&](const clang::OMPDependClause *clause) {
8         // Get the depend type
9         mlir::omp::ClauseTaskDependAttr dependType =
10            getDependKindAttr(this->builder, clause);
11        auto capturedVarsBegin = clause->varlist_begin();
12        auto capturedVarsEnd = clause->varlist_end();
13        // Get an mlir value for the pointer of each variable in the var list
14        for (auto varIt = capturedVarsBegin; varIt != capturedVarsEnd;
15             ++varIt) {
16          const clang::DeclRefExpr *varRef =
```

```
17              dyn_cast<clang::DeclRefExpr>(*varIt);
18
19          if (varRef) {
20            cir::LValue capturedLvalue = this->CGF.buildLValue(varRef);
21            cir::Address capturedAddress = this->CGF.buildLoadOfReference(capturedLvalue, location);
22            mlir::Value rawPointer = capturedAddress.emitRawPointer();
23
24            result.dependVars.push_back(rawPointer);
25            result.dependTypeAttrs.push_back(dependType);
26        }
27      }});
28  }
```

Listing 19: Implementation of the depend clause

14.3.4 Casts and dialect conversions

Finally, an overview of the casting and dialect conversion logic is provided. Referring back to the main CIR compilation flow (Figure (5)) and the concepts of dialect conversion discussed in Section (13.3.2), a potential issue becomes apparent: the mixture of dialects (CIR, Arith, Built-in, OpenMP...) means that not all operations or types will be immediately compatible. Unfortunately, there were incompatibilities between types of the CIR and OMP dialects, given that CIR uses its custom data types and OMP uses the default MLIR types for clauses arguments such as the boolean value of the final clause or the priority value of the priority clause. To overcome this obstacle, the author opted to use UnrealizedConversionCast operations. This operation allow for temporary intermixing without actually performing conversions. Given that CIR types are aliases of MLIR basic types (For example, cir.i32 and MLIR i32 share the same base type), there's no actual need of performing a real cast, and we would use a UnrealizedConversionCastOp to enable this temporary cast, which would then be actually fixed adding a reconcile-unrealized-casts pass in the lowering process.

There are two primary lowering or compilation flows:

1. **Direct CIR to LLVM IR Dialect Conversion**: This is the main and fastest path, directly converting CIR to LLVM IR.
2. **Progressive Lowerings with Multiple Dialects**: This is a more experimental path that allows for a greater degree of intermediate lowerings, passing through multiple dialects to achieve the final conversion.

Considering that we are using **UnrealizedConversionCastOp** from the **Built-in** dialect, the first compilation flow wasn't possible due to the conversion target of this lowering, which specified that only **LLVMIR** operations and types would be legal after the passes. The only change we have to do is add the **reconcile-unrealized-casts** pass on the lowering module.

After resolving the aforementioned issue, another challenge arose with casts that cannot be handled using UnrealizedConversionCastOps. One example is casting from cir.bool to the 1-bit integer type (i1) used in the OMP dialect. This wasn't feasible due to the bit width mismatch—cir.bool is implemented using an i8 integer. Creating a new dedicated casting operation seemed necessary, but given the complexity and overhead of introducing new operations, the author opted to exploit the Arith dialect.

The approach involved creating a constant 0 and using arith.cmpi to check inequality for the cir.bool value. Since arith.cmpi returns a boolean result (i1), it served the purpose effectively. However, this method required introducing two new operations (arith.constant and arith.cmpi) for each boolean cast. Recognizing that Clang represents true and false values as 0 or 1, arith.truncI (Truncate integer) emerged as a viable alternative. It truncates the 8-bit integer (cir.bool) to a 1-bit representation, reducing the cast to a single operation.

It's important to acknowledge that this workaround may not generalize well to other boolean representations that may vary across different languages. Moreover, a rewrite pattern (see Listing (14.3.4)) was added to automatically insert the truncation instruction whenever an UnrealizedConversionCastOp encounters a bit width mismatch between source and target types. Finally, integration of the arith-to-llvm conversion patterns, already implemented in MLIR, was necessary to complete the conversion pipeline to guarantee that the conversion target was respected.

```
class CIRUnrealizedConversionLowering: public mlir::OpConversionPattern<mlir::UnrealizedConversionCastOp> {
  public:
    using OpConversionPattern<mlir::UnrealizedConversionCastOp>::OpConversionPattern;
    mlir::LogicalResult matchAndRewrite(mlir::UnrealizedConversionCastOp op, OpAdaptor adaptor,
                                        mlir::ConversionPatternRewriter &rewriter) const override {
        // Force one-to-one casts
        assert(op.getOutputs().getTypes().size() == 1 && "expected a single type");

        auto inputType = adaptor.getInputs()[0].getType();
        auto outputType = op.getOutputs()[0].getType();

        // Ensure that the types are integer types
        if (!inputType.isa<mlir::IntegerType>() || !outputType.isa<mlir::IntegerType>()) {
            return mlir::failure();
        }

        auto inputIntType = inputType.cast<mlir::IntegerType>();
        auto outputIntType = outputType.cast<mlir::IntegerType>();

        // Check for bit width mismatch
        if (inputIntType.getWidth() > outputIntType.getWidth()) {
            // Insert TruncIOp to truncate the input to the output type
            // Creating a constant 0 and using arith.cmpi would work, but we will avoid the extra operation
                using truncate
            auto truncatedValue = rewriter.create<mlir::arith::TruncIOp>(op.getLoc(), outputType, adaptor.
                getInputs()[0]);
            rewriter.replaceOp(op, truncatedValue);
            return mlir::success();
        } else if (inputIntType.getWidth() == outputIntType.getWidth()) {
            // Directly replace the operation if types are the same
            rewriter.replaceOp(op, adaptor.getInputs());
            return mlir::success();
        }

        return mlir::failure();
    }
};
```

Listing 20: UnrealizedCastConversion Rewrite pattern

It's worth noting that these modifications are still under discussion on the CIR community, because the addition of a new dialect may introduce unwanted compilation time overhead that could be avoided creating a new redundant cast operation, but it is still being pondered by the project leaders.

14.4 Design decisions

This section aims to give a brief argumentation on the logic behind the proposed implementation, addressing its strong points, to achieve this in an elegant and understandable manner, common design patterns that were originally intended by the author will be mentioned.

Firstly, it is unmistakably clear that most code generation developed provided follows a behavioral pattern known as "Visitor", which tends to be a natural and elegant way of separating the algorithms and polymorphic objects. Considering that we are already working with polymorphic trees (the Abstract Syntax Tree) and that is considered a standard in compiler development, it's indeed a satisfactory solution. An example of this would be buildOMPTaskwaitDirective, which takes an AST node of type OMPTaskwaitDirective, and defines the treatment of this node, which objects should modify and which code should be emitted.

Secondly, despite not being strictly necessary in order for the proposed code generation solutions to work, all the runtime calls that are invoked during code generation visitors maintain the same control flow (referring to the flow of execution through flags and conditionals) as the original Clang code base. CIR from its early inception tries to replicate Clang code as much as possible and just changing what is strictly needed, in order to provide the same identical results in compilation as the original code generation compilation flow. Instances of this can be found in the proposed solution when some paths invoke the **llvm_unreachable** function, for example (This functions acts a more elaborate and richer assertion for typically "not yet implemented" features or actual errors). This wasn't originally intended, but it was done by the request of Bruno Cardoso.

Thirdly, a clause processor class was created to deal with the treatment of directives that may contain clauses. Given the repetitive nature of checking the variables and conditions inside these clauses, the author drew the conclusion this object, that resembled the behavior of the command design pattern, would feel natural. To simplify the complexity of parameter passing, the AST node and the IRBuilder are given as constructor parameters to the processor object.

Lastly, an acknowledgement to the original Clang and the Flang code-base, given that FIR (Fortran Intermediate Representation, an MLIR-based IR for Fortran), covers a great deal of OpenMP. These mature and robust front-ends have set an example for how code generation should be performed in modern compilers.

15 Results

15.1 Verification and testing

Key topics to cover are testing and validation, which are mandatory for verifying the proper functioning of the implementation. These tests have been done throughout the thesis following the unit-test approach using LIT (LLVM Integrated Testing), which are mandatory in pull requests to justify the expected behaviour of the newly introduced features. Given that this work implements code generation of different directives, for each single directive a couple of tests should be implemented:

- **Code Generation**: These tests are aimed into validating that high level language programs (C/C++...) are translated into CIR properly. This is achieved, validating that certain key instructions exist on the CIR-translated program.
- **Lowering**: Similarly to code generation, lowering validation is performed through regex matching of the source code (In this case CIR), checking the existence of certain instructions or constructs on the target code (LLVM IR).

The LIT framework operates on a string-matching basis and plays a critical role in development by ensuring that compiler transformations maintain expected functionality and correctness. It is integral to the testing practices employed throughout the thesis, providing confidence in the implementation's reliability before integrating new features into the main upstream code-base. Examples of such tests can be found on section A.1 and now on the main branch of Clang.

15.2 Experimentation

Although this thesis does not primarily focus on experimentation, it is important to provide some key statistics for context. Given the nature of the work, traditional execution time experiments, common in compiler research, are less relevant here. Instead, the emphasis is on practical code generation rather than optimization. However, two experiments—compilation time and execution time—have been conducted. These tests were performed using three different compilers: GCC, classical Clang, and Clang with CIR. These experiments aim to evaluate the performance of the compilers, particularly assessing whether the CIR pipeline produces programs with execution times comparable to the standard Clang-LLVM compilation pipeline. Achieving comparable execution times would further validate the normal functioning of the CIR pipeline.

It is important to point out that CIR is not yet fully mature and has a lot of missing or not fully developed features, and that its optimization pipeline has not yet been exploited, so one should expect neither speed-ups in respect to the original Clang nor complete coverage of the C/C++ languages, yet.

The experiments have been conducted solely using a compilation script in Python and a set of 4 sample codes, which consist of basic algorithms (search of a value in an unsorted array, sum of an array, merge sort and matrix multiplication) with some OpenMP constructs to implement their parallel counterparts. After individually compiling (using -O3 optimization flag) and sequentially executing the sample codes for N iterations, averages are computed to minimize potential inaccuracies stemming from variations in computing resources while compiling or executing. The following findings are reported below:

1. **Compilation time:**

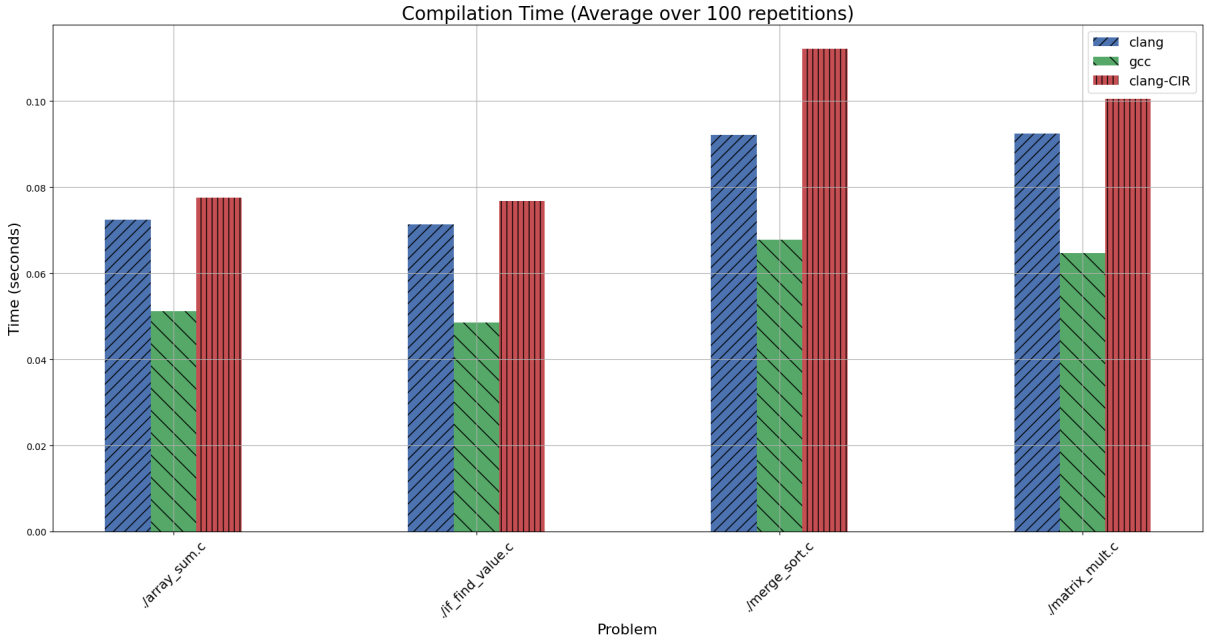

Figure 20: Comparison of compilation times, self-elaborated

2. **Execution time:**

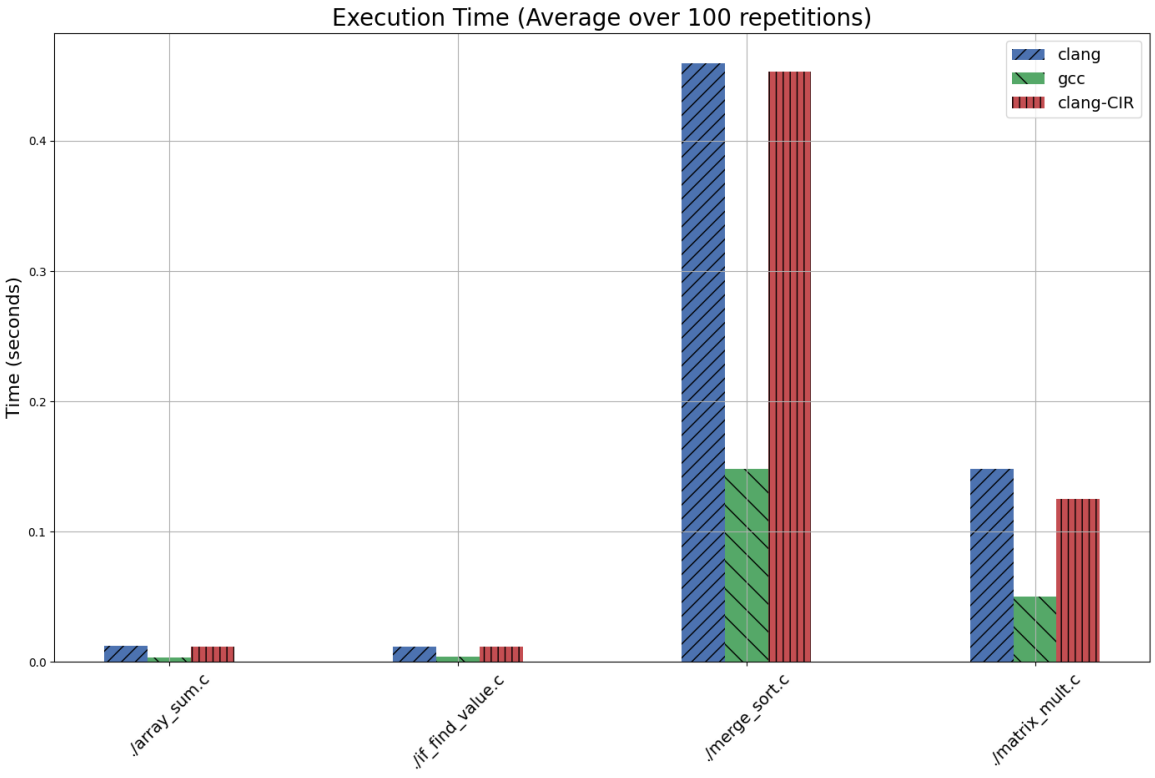

Figure 21: Comparison of execution times, self-elaborated

Given these plots of the compilation/execution data, one can infer several key insights:

1. GCC compile times are the lowest.
2. The compile time using CIR as an intermediate representation of Clang is similar or higher to the regular Clang compilation flow (Lower bound).
3. Execution times greatly vary depending on the kind of program, but generally Clang and Clang-CIR, counterintuitively perform way worse than GCC (Almost 2 times).
4. The classical Clang compilation flow produces almost identical execution times compared to the CIR compilation flow.

However, It's essential to acknowledge that there are underlying factors that the reader may not be aware: GCC and Clang have different OpenMP runtimes that perform different computations and utilize distinct synchronization mechanisms, this could explain for instance insight number 3. Moreover, the reason behind insight 1 can be attributed to the lightweight nature of GCC. Originating in the 1980s, GCC was designed to be compact enough for old computers and even modern embedded systems. Examining its compact code-base, which includes extensive inlining and global variable handling, highlights the significant contrast between GCC and Clang. The latter prioritizes extensibility and modularity through larger abstractions and a more expansive code-base, that is the main advantage of using GCC.

Despite the lightweight nature of GCC, its execution times are typically higher than Clang's due to Clang's more efficient optimization infrastructure and better utilization of main memory. However, the plots in our analysis show the opposite trend. This discrepancy is likely due to differences in OpenMP runtimes, particularly in how Clang handles task synchronization and the low parallelism of the executions, rather than any inherent advantage of GCC.

Moreover, the second insight is readily apparent when considering the differences between the two compilation flows (see Figure 5). The flow that exclusively uses LLVM as its IR represents a strict subset of the operations in the ClangIR-based flow. Therefore, in the best-case scenario, the CIR compilation flow could only be as fast as the classic LLVM-based one. This comparison establishes a lower bound for performance expectations. However, it's unrealistic to expect the compile times to be equal due to the additional overhead introduced by the CIR compilation flow.

Finally, the similarity between execution times of Clang and Clang-CIR find its explanation in the current lack of CIR-specific optimizations of this pipeline. Therefore, both compilations flows apply the same set of optimizations, the general LLVM-IR middle-end optimization passes followed by the back-end specific optimizations, which are based on **x86_64** due to the architecture of the author's CPU.

Note that the execution script and the sample codes that have been mentioned, can be found on the fourth subsection A.3.1 of the appendix.

16 Conclusions

16.1 Reflection

This dissertation has fulfilled its duty as an introductory reference to compilers and the modern relevant frameworks that are currently being used by the LLVM community, the most substantial compiler ecosystem in our contemporary world. Although the project has always been surrounded by a great deal of uncertainty and has suffered from changes in direction and adjustments in priorities, it has successfully accomplished its core purpose, which was contributing to the LLVM ecosystem and introducing OpenMP to this novel and this promising intermediate representation that is ClangIR, through the efforts of the author and Fabian Mora. Eight new core OpenMP directives were introduced and this will surely speed up the set in production of ClangIR, which will eventually be the main compilation flow of the Clang compiler.

To conclude, this thesis makes a modest but valuable contribution to Clang. With the support of its growing community and the open-source initiatives driven by top-tier companies like ARM, Meta, and Intel, CIR will undoubtedly be a success. This will benefit the global community of engineers and developers who use this tool daily, enhancing the quality of their work, their user experience and leading to more optimal, secure and bug-free code.

16.2 Clang vs GCC

The age-old debate about Clang or GCC inevitably arises whenever modern compilers are mentioned. Despite its popularity and the fast rate of improvement by both compilers, not many meticulous experiments on compilation and execution times have been made. Generally, users find that Clang produces marginally faster executable code but has slower compilation times compared to GCC. However, this basic thesis experiment suggests otherwise, potentially introducing bias to the reader. Nonetheless, this comparison is hugely dependant on the type of program being executed and the different optimization pipelines that both compilers offer. For a greater understanding refer to [57] and [58].

The emergence of Clang nearly 20 years ago, directed by Apple's compiler team under Chris Lattner, ignited a competition for C compiler supremacy. This rivalry has forced the GNU community to continuously enhance their compiler, in danger of the potential obsolescence of GCC. As a result, significant advancements in compiler infrastructure have occurred, ending the stagnation that surrounded the GCC project for decades. It is vital to maintain this competitive spirit to ensure ongoing improvements. Consequently, society benefits more from this healthy competition than from being divided into rigid camps favoring either Clang or GCC. Therefore, the author encourages you to appreciate the constructive nature of competition rather than taking a simplistic view of this important topic and simply choosing one side.

When choosing between Clang and GCC, consider the following key trade-offs and factors:

- **License**: For industrial products, the GPLv3 license of GCC requires the software to be free and open-source. In contrast, Clang, with its Apache 2.0 LLVM license, permits modifications and private usage, making it preferable for either proprietary applications or free software at the same time.
- **Time Tradeoff**: Historically, GCC has offered faster compilation times, while Clang has often achieved better execution times. However, there have been instances where Clang excels in both

aspects, as well as GCC. This trade-off should be carefully considered based on the specific needs of your project and the compiler version being used.

- **Space**: If disk space is a significant concern (For example, embedded systems that make use of compilers), GCC is generally the better choice due to its smaller footprint.

- **Extensibility**: For building custom compilers, LLVM/Clang is superior due to its modular design and extensibility. Unlike GCC rigid middle-end, LLVM features an universal and easy-to-use intermediate representation (IR) specifically designed to support a wide range of backends, optimizations and has a complete API library, making it more practical and efficient for custom compiler development.

16.3 Compiler development

Furthermore, another conclusion that one can infer is that LLVM compiler development is not simple, due to the intensive use of computing resources (Which takes prolonged compilation times), large and intricate code-bases but more importantly, a great steep learning curve for newcomers. This is due to the fact that the frameworks involved take a considerable amount of time to master, and one must learn the work-flow and code standards of the project meticulously, the tools of the ecosystem and most importantly, get proficient in reading and understanding LLVM IR / MLIR code for debugging purposes, which similar to assembly languages, requires dedicated practice and familiarity over time.

Compilers are often underrated or overlooked because to outsiders, compilers seem like an already solved problem, and while that statement is partially true, compilers still represent a vast and ongoing engineering challenge. Their relevance to our technological society is immense, given that all professionals, scholars, and users rely on these tools at the end of the day. Almost all the gigantic projects that are used every single day by the industry, such as Python, PyTorch, TensorFlow, and Go, depend or have depended on compilers.

Operating systems like Linux, Unix, and Windows, as well as databases like MySQL, Oracle, and PostgreSQL, heavily rely on compilers. Essential tools and applications like Git (version control system), Apache (web server), OpenGL (graphics rendering), Vulkan (graphics API), and Vim (text editor) are also written in C. Given the significant importance of C (which some experts consider more a protocol than a simple programming language), the role of C-based compilers like GCC or Clang is monumental. These compilers form the backbone of technology, programming, and the tech industry, shouldering their immense responsibility akin to Atlas.

Thus, improving the extensibility, modularity, efficiency, and ease of use of compilers significantly facilitates the creation of new programming languages and the enhancement of existing ones. This leads to substantial global savings in resources, including money, electricity, and human time, highlighting the immense importance of this tool. Often people don't think of the huge effort that creating a programming language is, and how often new programming languages or extensions of the most popular ones are created. Therefore, the creation of this greatly reusable LLVM infrastructure has saved thousands if not millions of hours of development in state-of-the-art languages like Rust, Julia, Swift...

Lately, there has been a trend in computer science towards using high-level programming languages and black-box technologies, often to the point where users are unaware of the tools they are using. We have layered so many levels of abstraction that we have lost touch with the underlying machinery. While this has increased efficiency, it has come at the cost of a deeper understanding. As users of these technologies, we need to empower ourselves by becoming familiar with these tools, stepping out of our small bubbles to explore their full capabilities. If everyone in the tech industry had a comprehensive understanding of computer science, the overall efficiency of programmers would improve. A deeper level of abstraction directly correlates with enhanced problem-solving abilities, ultimately leading to more effective and innovative solutions. Not only that, but we as developers have the need and the obligation of choosing the next generation of tools that will shape our technology and, therefore, the world of tomorrow.

16.4 The importance of Open Source

One key term that has appeared over and over during the course of this dissertation is **Open Source**, thus one can infer that it's quite relevant. From its early stages, the author possessed little knowledge about what open source is and the impact it has on our modern day technological society.

Open source refers to software that is freely available to anyone to interact, modify and enhance its source code. Its philosophy stands for collaboration, transparency, community, and freedom. This philosophy has gained traction thanks to early initiatives like the GNU project and the Free Software Foundation, and has evolved to become the standard in modern software development.

The impact of open source is clear, developers from around the world are welcomed to contribute and give their opinions, accelerating the pace of development and the quality of the work. Several vital projects such as Linux, Apache, Kubernetes, GCC and LLVM are prime examples of how open source reduces the costs for both companies and individuals, enabling smaller companies and individuals to make use of such complex tools freely and launch products and software without paying any license fees.

Furthermore, its collaborative environment promotes contributions, knowledge sharing and continuous learning which may be greatly beneficial to individuals who want to improve their development toolbox, hand to hand with great engineers from top companies. Therefore, its educational value for new developers that may get to know real world software development and best practices, it's undeniable.

Major corporations like Google, Meta, Microsoft, IBM, Intel, AMD, Nvidia... are increasingly embracing the open source paradigm, given the win-win scenario it provides in terms of development time and engineering effort. However, these tech companies are often still in competition and tend to monopolize these projects, leading to tensions between them and diminishing the voice of independent developers.

It is imperative to note the privatization of all software poses significant dangers. When software is proprietary, control is monopolized in the hands of a few companies, limiting user's ability to understand and improve the tools they depend on. This centralization can extinguish innovation at all, as developers are forced to work within the constraints of closed systems and pay for licenses that can be prohibitively expensive for individual users. Furthermore, privatized software often comes with restrictive terms of service, limiting how this product that the user has already paid can be used. Such restrictions

erase educational opportunities, as users lose control of their tools. In a world where software is increasingly relevant to all aspects of life, from how human interact with each other to how cars are driven autonomously, the monopolization of software control threatens to undermine both individual freedoms, and the collective progress and health of society. Therefore, it's crucial to the development of our future technological society that open source software is prioritized above private software and expected and even enforced from the tech giants.

However, it is important to note that as already mentioned 16.2, the cost of entrance to these projects tends to be significant and the standards in pull requests are high. So contributing isn't as straightforward as it may seem from the outside, and communication overheads should also be considered.

16.5 Acknowledging the role of FIB-UPC and BSC

An expression of gratitude is made from the author to its faculty FIB, from Universitat Politècnica de Catalunya for the invaluable academic framework provided, through Theory of Computation, Programming Languages and Compilers subjects specially. Without all the knowledge that has been passed down to the author about compilers and computer science in general and the problem-solving tool set that has been honed through the course of the degree, this thesis wouldn't have been possible in this short timeline.

Last but not least, thanks to the Barcelona Supercomputing Center, which has a tight relationship with FIB, for mentoring me and introducing me to these complex topics that shape modern compiler development, through Roger Ferrer Ibàñez.

However, it's important to acknowledge that while this project relies on novel frameworks that are not typically covered in standard college curricula, the challenge extends beyond the lack of familiarity with these frameworks. The educational approach at UPC tends to emphasize theory over practical application and framework utilization. Consequently, the obstacle to successfully completing the thesis lies not only in the absence of framework knowledge but also in the limited depth of theoretical understanding. The teachers make efforts to impart as much knowledge about compilers as possible within the compilers course curriculum. However, given the inherent complexity of compiler development and the small time-frame, this coverage primarily addresses the front-end aspects and serves as an introduction to the middle-end and real code generation, providing a tiny glimpse into LLVM.

It may be beneficial for the faculty to reconsider the curriculum structure and prioritize compilers as a mandatory subject over other courses. This subject is not even mandatory for students majoring in the computing specialization, and is highly discouraged to take by students out of the computing major due to **Theory of Computation** being a prerequisite of this subject, which points out possible flaws in the actual curriculum structure. Additionally, reintroducing a second compilers course, as was done in the past, could provide students with a deeper understanding of middle-end and back-end compiler concepts. Compilers are fundamental to computer science and should be treated as such, akin to subjects like databases, operating systems, or data structures and algorithms.

17 Future improvements and unimplemented features

This final last section aims to shed light into the issues that were encountered during the thesis which prevented the full implementation of some OpenMP features, which were greatly explored but discarded for the purpose of the thesis. The root causes of the 3 unimplemented features remain the same:

1. **Community overheads**: Given that this project is not an individual project, requests for comments and GitHub issues must be opened to get the approval of project leaders and the community in general for deciding how to tackle the resolution of delicate issues that may have multiple solutions. As a result, this alignment with the CIR community caused unforeseen delays and, in extreme cases, even led to the abandonment of the task, such as data-sharings and taskloop.

2. **Opportunity Cost**: When tasks were highly complex and required intensive approval from the CIR community, there was a tradeoff between addressing these intricate details and continuing to implement simpler directives. Opting for simpler directives would allow for a more comprehensive CIR dialect. Essentially, it was a tradeoff between depth of implementation and breadth of coverage. At a certain point, the author chose the latter, aiming to create a more complete and usable dialect, even if it meant sacrificing some depth in individual implementations. Additionally, it's important to note that for the purposes of this thesis, having working directives in the upstream repository is far more valuable than having a complex solution that only functions locally.

3. **Customizing MLIR**: Some issues arose from the lack of support of some features by MLIR. At the beginning of the thesis, this tool was considered mature and flexible enough to cover our needs. However, some features above prove otherwise, and that would mean that the author should either find a hacky local way to modify MLIR, which would be immensely time-consuming due to getting used to a new code-base or getting involved with the MLIR community and actually modifying an upstream version through pull requests.

This work serves as a great starting point for OpenMP inside of CIR, however there is still a long way to go before this reaches an end. The implemented directives don't have a 100% clause coverage, therefore some of these clauses could be a great first issue by a newcomer that wants to start contributing to CIR. Not only that, but there's still a great number of directives that are yet to be implemented, but again, future contributors may benefit from this work as a reference, using it to kickoff their development. Lastly, the author has raised awareness to MLIR and OpenMP communities of the lack of support that is being given to task data-sharings and other features, which will eventually be supported, and added into CIR afterward.

17.1 Data sharings

Data sharings were always a top priority in the author's original planning, given that this enables a full exploitation of the OpenMP framework, and not constrained to only using shared variables. This option was meticulously explored by the author and the co-supervisor of the thesis. After exploring the FIR code-base, a conclusion was drawn, MLIR doesn't offer yet support for data sharings among task-based directives. So this leaves us only with the possibility of implementing classical and redundant way of dealing with private data sharings, through adding and tracking a new set of variables in the code generation to emulate this private variables, which is a good approach but given the complexity and churn that adds to the MLIR code generation, and due to the request of some reviewers of the CIR community,

this assignment has been temporarily discarded until official MLIR support is given to this feature.

This decision was taken by the author despite the importance of private data-sharings mostly due to the cost of opportunity and to avoid the complexity that the latter solution would give to the ClangIR code-base.On a positive note, the author's efforts have led to the priorization of this feature in the near future by the MLIR team.

17.2 Taskloop

The taskloop directive, while initially considered, ultimately took a back seat due to various factors: Firstly, MLIR provides no native support for task-based data sharings. Secondly, the complexity of implementing this directive was another key consideration. Unlike some simpler directives, such as barrier or taskyield, the taskloop directive required a more extensive time investment for development and testing, and much generous time window than initially expected.

Given these challenges and constraints, the decision was made to deprioritize the taskloop directive in favor of other directives that could be implemented more efficiently within the available time frame. While the taskloop directive remains a potential area for future exploration and development, the focus was redirected towards implementing directives that could provide immediate benefits and address pressing needs within the compiler framework. Finally, the decision to forego the implementation of taskloop allowed sufficient time to focus on other critical tasks. As a result, the vast majority of directives were successfully integrated into the upstream open-source project. Had this not been the case, these directives would have remained confined to the author local version as a draft or proposal, limiting their impact and accessibility.

17.3 Depend Clause

The depend clause, though not extensively used by the community, was the author's favorite and was given some priority. Significant efforts were aimed at implementing this directive, and it was almost completed. The main obstacle was the lack of pointer-like interfaces in CIR, which the OpenMP dialect needs to correctly implement these depend clauses. Notably, there is still an ongoing discussion on how these pointer interfaces will be implemented by the project leaders. It is hoped that this discussion will conclude shortly after the thesis is finalized, allowing the depend clause to become fully functional and upstream.

A Appendix

A.1 Environment and building Clang

This section aims to provide a concise guide for setting up the environment and installing a local version of Clang. Whether you're experimenting with the latest Clang features or developing on LLVM-related projects, this process can be daunting without clear instructions or guidance.

In order to compile a local version of Clang, several tools must be installed before, such as the Cross-platform build system generator **CMake**, the build system **Ninja** (A modern version of **Make**), a stable compiler such as **Clang** or GCC, a linker such as **LLD** and **git**, to download the whole LLVM project where Clang resides. Assuming that the source operating system is a Debian distribution, such as Ubuntu 20.04/22.04, one can install those tools using the following commands:

```
# Update package index
sudo apt-get update

# Install CMake
sudo apt-get install cmake

# Install Ninja
sudo apt-get install ninja-build

# Install a Compiler
sudo apt-get install clang

# Install a Linker
sudo apt-get install lld

# Install Git
sudo apt-get install git
```

Listing 21: Pre-build setup

Now, create a working directory where the source code, build and install directories will be created in order to download, compile and install Clang in the local machine. For example:

```
/work-dir
├── clangir-source
├── clangir-build
└── clangir-install
```

Now download the LLVM repository through git inside the `clangir-source` directory and create a path variable `$CIR_INSTALL` that will come in handy. Then, run the CMake command inside the `clangir-source` directory to generate the ninja build files, which will be used to compile the source code. There's a huge number of compilation options in the LLVM project, but we will focus on enabling debug mode (useful to have greater error handling when developing), enabling OpenMP runtime, selecting the build of only "Clang" and "MLIR" out of all the other projects LLVM has, and enabling the build of ClangIR. After the completion of the CMake command, the build and installation can be done easily through ninja:

```
#Download LLVM project
git clone https://github.com/llvm/clangir.git clangir-source

#Create an install Path variable
export CIR_INSTALL = $(pwd)/clangir-install

#Creating the build files
cd clangir-build

cmake ../clangir-source/llvm -GNinja \ #Select a build system
-DCMAKE_BUILD_TYPE=Debug \ #Compile using debug mode
-DCMAKE_INSTALL_PREFIX=$CLANGIR_INSTALLDIR \ #Installation prefix
-DLLVM_ENABLE_ASSERTIONS=ON \
-DLLVM_TARGETS_TO_BUILD="host" \
-DLLVM_ENABLE_PROJECTS="clang;mlir" \ #Sub Projects to build
-DCLANG_ENABLE_CIR=ON \ #Enabling ClangIR
-DCMAKE_CXX_COMPILER=$(which clang++) \ #Select the C++ compiler
-DCMAKE_C_COMPILER=$(which clang) \ #Select the C compiler
-DLLVM_ENABLE_RUNTIMES="openmp" \ #Building OpenMP runtime
-DLLVM_BUILD_EXAMPLES=ON \
-DLLVM_USE_LINKER=lld \ #select LLD as the linker
-DLLVM_USE_SPLIT_DWARF=ON

#Compilation of the source code using X threads
ninja -j X

#Installation of the binary files
sudo ninja install
```

Listing 22: Build of ClangIR

Building ClangIR with OpenMP, consisting of approximately 7000 large files, may demand as much as **130 GB** of storage space, particularly when opting for debug mode (with about 20GB less required without enabling examples). Given the intensive CPU and memory usage, the build process can be time-consuming, typically taking around 1 hour on the author laptop. Insufficient memory may cause the machine to idle for minutes, emphasizing the need for adequate resources—ideally, a minimum of 2GB per core to avoid reliance on Swap memory. With 16 GB of available memory, it's advisable not to exceed 8 threads to maintain optimal performance, for example. Additionally, since builds are somewhat incremental, transitioning between git branches necessitates recompiling the entire project from scratch, which can significantly impact productivity. Therefore, developers are encouraged to switch between branches sparingly to minimize disruptions.

A.2 Compiling files and testing

With the environment now set up, one can make use of the full capabilities of Clang using CIR and also enables the reader to develop and contribute to Clang.

Recalling figure 5 4, there were 2 possible compilation flows, the traditional clang to LLVM IR code generation (1) or the new ClangIR code generation which outputs the MLIR CIR dialect representation of that same source code (2), which then can be lowered to LLVM IR (3). Taking for instance that the source file is called source.c, the flags and options to achieve such compilation with our locally compiled

Clang are:

```
#1. Traditional Clang -> LLVM IR code generation
$CIR_INSTALL/bin/clang-19 -S -emit-llvm source.c -o source.ll

#2. ClangIR code generation (use clang-cpp instead of 19!)
$CIR_INSTALL/bin/clang-cpp -fclangir -Xclang -emit-cir source.c -o source.cir

#3. ClangIR code generation + lowering LLVM IR
$CIR_INSTALL/bin/clang-19 -fclangir -S -emit-llvm source.c -o source.ll

#EXTRA: Visualize the AST of the source file
$CIR_INSTALL/bin/clang-19 -Xclang -ast-dump source.c
```

Listing 23: Compilation using Clang

Note that this is subject to changes in the future, not only the clang last version (19) but also the early stage on which CIR finds itself into. For example, the flag to enable CIR compilation flow was originally called **-fclangir-enable** but on one of the latest upstreaming was renamed **-fclangir**. To enable OpenMP the flags **-fopenmp** / **-fopenmp-enable-irbuilder** must be set.

Another crucial aspect of contributing to Clang is the testing part, which can be easily located inside of clang/test/CIR directory and contains tests for code generation, lowerings, transformations... These unit tests are written in on C/C++/ll/cir files and verified thanks to the LIT (LLVM Integrated Tester) tool. To compile and execute the aforementioned, the following build command can be used:

```
ninja check-clang-cir
```

An extensive guide on how to use the LIT tool can be found at the LLVM LIT guide [59], but it boils down to simple textual matching using regular expressions, given that these files are strings at the end of the day. The developer must explicitly write the relevant strings that should be found after performing a code generation or a lowering:

Taking a simple code generation taskwait, our aim of this test is to check that after emitting CIR, effectively a function has been created that contains a omp.taskwait operation inside of it.

```
// RUN: %clang_cc1 -std=c++20 -triple x86_64-unknown-linux-gnu -fopenmp -fclangir -fopenmp-enable-irbuilder
     -emit-cir %s -o %t.cir
// RUN: FileCheck --input-file=%t.cir %s

// CHECK: cir.func
void omp_taskwait_1(){
// CHECK: omp.taskwait
  #pragma omp taskwait
}
```

Listing 24: Taskwait CodeGen test

```
// RUN: cir-translate %s -cir-to-llvmir | FileCheck %s


module {
```

```
  cir.func @omp_taskwait_1() {
    omp.taskwait
    cir.return
  }
}

// CHECK: define void @omp_taskwait_1()
// CHECK: call i32 @__kmpc_global_thread_num(ptr {{.*}})
// CHECK: call i32 @__kmpc_omp_taskwait(ptr {{.*}}, i32 {{.*}})
// CHECK: ret void
```

Listing 25: Taskwait Lowering test

A.3 Contribution Git work-flow

Contributing to a big open source project such as LLVM requires knowledge of the pull request and rebase workflows of Git and also being highly aware of the LLVM contribution guidelines [60], that state that every single patch must obey:

- Contains unit tests
- Conform the LLVM coding standards (clang-format tool)
- Upload atomic and isolated changes
- Uploading unrelated changes is forbidden
- Commits must be up-to-date with the remote origin/main branch and should not contain merges.

Pull requests are the de-facto workflow for open source which involve 2 figures, the developer, and the reviewers. A Developer creates a branch with patches to review, opening a pull request to fix some certain issue or to add some new feature and that sparks an iterative process between the reviewers, which request changes to the patch and the developer, who uploads code updates until the developer gives up or the pull request is approved and then merged into the main branch (In this case onto origin/main of ClangIR).

Rebasing is a crucial aspect of this Git workflow. It involves taking a series of commits, often an entire branch, and relocating them onto a new base, typically the main branch. The following illustration demonstrates this process:

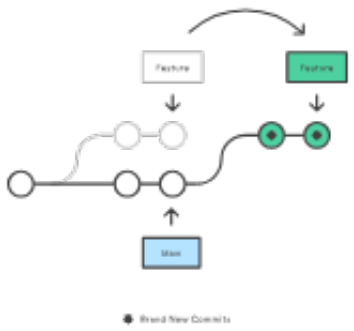

Figure 22: Illustration of a Git rebase, Retrieved from [61]

Moreover, it's important to note that ClangIR is periodically rebased against its upstream, the main LLVM repository. This highlights the necessity of harnessing rebases to prevent branches from becoming outdated. While rebasing may introduce some overhead, particularly when performed during a pull request, it's essential for contributing effectively to the project.

A.3.1 LLVM API static example

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include "llvm/Transforms/Utils.h"
#include "llvm/Transforms/Scalar.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/Support/TargetSelect.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/FileSystem.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/MC/TargetRegistry.h"
#include <optional>

int main() {
    // Initialize the LLVM context which holds global data used by LLVM
    llvm::LLVMContext Context;

    // Create a new LLVM module, which is a container for functions and global variables (Basically a
        program/file)
    std::unique_ptr<llvm::Module> Module = std::make_unique<llvm::Module>("my_module", Context);

    // Create an IRBuilder, which is a helper object to create LLVM instructions
    llvm::IRBuilder<> Builder(Context);

    // Define the 'add' function which takes two float arguments and returns a float
    llvm::FunctionType *AddFuncType = llvm::FunctionType::get(Builder.getFloatTy(), {Builder.getFloatTy(),
        Builder.getFloatTy()}, false);
    llvm::Function *AddFunction = llvm::Function::Create(AddFuncType, llvm::Function::ExternalLinkage, "add
        ", Module.get());

    // Create a single basic block in the 'add' function
    llvm::BasicBlock *AddBB = llvm::BasicBlock::Create(Context, "entry", AddFunction);
    Builder.SetInsertPoint(AddBB);

    // Get the function arguments "a" and "b"
    auto Args = AddFunction->args().begin();
    llvm::Value *A = Args++;
    A->setName("a");
    llvm::Value *B = Args++;
    B->setName("b");

    // Create the addition instruction
    llvm::Value *Sum = Builder.CreateFAdd(A, B, "sum");
    //Create a return instruction
    Builder.CreateRet(Sum);

```

```
46  // Define the 'main' function which returns an int and takes no arguments
47  llvm::FunctionType *MainFuncType = llvm::FunctionType::get(Builder.getInt32Ty(), false);
48  llvm::Function *MainFunction = llvm::Function::Create(MainFuncType, llvm::Function::ExternalLinkage, "
        main", Module.get());
49
50  // Create a single basic block in the 'main' function
51  llvm::BasicBlock *MainBB = llvm::BasicBlock::Create(Context, "entry", MainFunction);
52  Builder.SetInsertPoint(MainBB);
53
54  // Create constants instructions for the float values 3.14 and 420.0
55  llvm::Value *X = llvm::ConstantFP::get(Context, llvm::APFloat(3.14f));
56  llvm::Value *Y = llvm::ConstantFP::get(Context, llvm::APFloat(420.0f));
57
58  // Call the 'add' function through call instruction with X and Y as arguments and store the result
59  llvm::Value *AddCall = Builder.CreateCall(AddFunction, {X, Y}, "call_add");
60
61  // Return 0 from the 'main' function
62  Builder.CreateRet(Builder.getInt32(0));
63
64  // Verify the module to ensure it's well-formed
65  if (llvm::verifyModule(*Module, &llvm::errs())) {
66      llvm::errs() << "Error constructing function!\n";
67      return 1;
68  }
69
70  // Optimize the module using various optimization passes
71  // More info on https://llvm.org/docs/Passes.html
72  llvm::legacy::PassManager PM;
73  PM.add(llvm::createPromoteMemoryToRegisterPass());
74  PM.add(llvm::createReassociatePass());
75  PM.add(llvm::createCFGSimplificationPass());
76  PM.run(*Module);
77
78  // Initialize the native target to generate machine code for the host machine
79  llvm::InitializeNativeTarget();
80  llvm::InitializeNativeTargetAsmPrinter();
81  llvm::InitializeNativeTargetAsmParser();
82
83  // Set the target triple which specifies the target architecture, vendor, and OS
84  //The triple here is hardcoded just to showcase it. Use the line below instead.
85  std::string Error;
86  auto TargetTriple = "x86_64-pc-linux-gnu";
87  //auto TargetTriple = llvm::sys::getDefaultTargetTriple();
88  Module->setTargetTriple(TargetTriple);
89
90  // Lookup the target based on the target triple
91  auto Target = llvm::TargetRegistry::lookupTarget(TargetTriple, Error);
92  if (!Target) {
93      llvm::errs() << Error;
94      return 1;
95  }
96
97  // Create a target machine which generates code for the specified target
98  // This target machine object encapsulates all the complexities of the target language
99  auto CPU = "generic";
100 auto Features = "";
```

```
101     llvm::TargetOptions opt;
102     auto RM = llvm::Optional<llvm::Reloc::Model>();
103     auto TargetMachine = Target->createTargetMachine(TargetTriple, CPU, Features, opt, RM);
104 
105     //Set the data layout on the module
106     Module->setDataLayout(TargetMachine->createDataLayout());
107 
108     // Open a file to write the generated object file
109     std::string Filename = "output.o";
110     std::error_code EC;
111     llvm::raw_fd_ostream dest(Filename, EC, llvm::sys::fs::OF_None); // Use llvm::sys::fs::OF_None
112     if (EC) {
113         llvm::errs() << "Could not open file: " << EC.message();
114         return 1;
115     }
116 
117     // Create a pass manager to emit the object file
118     llvm::legacy::PassManager pass;
119     auto FileType = llvm::CGFT_ObjectFile;
120 
121     if (TargetMachine->addPassesToEmitFile(pass, dest, nullptr, FileType)) {
122         llvm::errs() << "The TargetMachine can't emit a file of this type";
123         return 1;
124     }
125 
126     // Run the pass manager to generate the object file
127     pass.run(*Module);
128     dest.flush();
129     return 0;
130 }
```

Listing 26: Basic usage of LLVM API, self-elaborated

A.4 Experimentation

As a reference for the reproducibility of the proposed experiment of the thesis, the scripts, and code samples can be found on the following sections:

A.4.1 Script

```python
import subprocess
import os
import time
import matplotlib.pyplot as plt
from typing import List, Tuple
import argparse
import numpy as np
import tempfile

def plot_times(file_paths: List[str], compiler_names: List[str], compilation_times: List[List[float]],
     execution_times: List[List[float]], num_reps: int) -> None:
    num_compilers = len(compiler_names)
    x = np.arange(len(file_paths))

    colors = ['#4c72b0', '#55a868', '#c44e52']
    hatch_patterns = ['//', '\\', '||']

    fig, ax = plt.subplots(figsize=(14, 10))
    bar_width = 0.15
    for i in range(num_compilers):
        ax.bar(x + bar_width * i, [ct[i] for ct in compilation_times], bar_width, label=compiler_names[i],
             color=colors[i % len(colors)], hatch=hatch_patterns[i % len(hatch_patterns)])

    ax.set_xlabel("Problem", fontsize=16)
    ax.set_ylabel("Time (seconds)", fontsize=16)
    ax.set_title(f"Compilation Time (Average over {num_reps} repetitions)", fontsize=20)
    ax.set_xticks(x + bar_width * (num_compilers - 1) / 2)
    ax.set_xticklabels(file_paths, fontsize=14)
    ax.legend(fontsize=14)
    ax.grid(True)
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()

    # Plotting Execution Times
    fig, ax = plt.subplots(figsize=(14, 10))
    for i in range(num_compilers):
        ax.bar(x + bar_width * i, [et[i] for et in execution_times], bar_width, label=compiler_names[i],
             color=colors[i % len(colors)], hatch=hatch_patterns[i % len(hatch_patterns)])

    ax.set_xlabel("Problem", fontsize=16)
    ax.set_ylabel("Time (seconds)", fontsize=16)
    ax.set_title(f"Execution Time (Average over {num_reps} repetitions)", fontsize=20)
    ax.set_xticks(x + bar_width * (num_compilers - 1) / 2)
    ax.set_xticklabels(file_paths, fontsize=14)
    ax.legend(fontsize=14)
    ax.grid(True)
    plt.xticks(rotation=45)
    plt.tight_layout()
```

```
47      plt.show()
48
49  class Compilation:
50      def __init__(self, compiler_path: str, file_path: str, flags: List[str], preprocess: bool = False) ->
            None:
51          self.compiler = os.path.expandvars(compiler_path)
52          self.file_path = file_path
53          self.flags = flags
54          self.preprocess = preprocess
55
56  def preprocess_file(original_file_path: str) -> str:
57      preprocessed_file_fd, preprocessed_file_path = tempfile.mkstemp(suffix=".c")
58      with open(original_file_path, 'r') as original_file:
59          lines = original_file.readlines()[1:] # Skip the first line
60      with os.fdopen(preprocessed_file_fd, 'w') as preprocessed_file:
61          preprocessed_file.writelines(lines)
62      return preprocessed_file_path
63
64  def measure_time(func) -> Tuple[any, float]:
65      def wrapper(*args, **kwargs):
66          start_time = time.time()
67          result = func(*args, **kwargs)
68          end_time = time.time()
69          return result, end_time - start_time
70      return wrapper
71
72  # Returns both compilation and execution times
73  def compute_times(compilation: Compilation, num_reps: int) -> Tuple[float, float]:
74      if compilation.preprocess:
75          file_path_to_compile = preprocess_file(compilation.file_path)
76      else:
77          file_path_to_compile = compilation.file_path
78
79      compile_cmd = [compilation.compiler] + compilation.flags + [file_path_to_compile, "-o", os.path.
            splitext(compilation.file_path)[0]]
80      execution_cmd = [os.path.splitext(compilation.file_path)[0]]
81
82      compile_times = []
83      execution_times = []
84
85      for i in range(num_reps):
86          start_cmp = time.time()
87          subprocess.run(compile_cmd)
88          end_cmp = time.time()
89          compile_times.append(end_cmp - start_cmp)
90
91          start_exec = time.time()
92          subprocess.run(execution_cmd)
93          end_exec = time.time()
94          execution_times.append(end_exec - start_exec)
95
96      if compilation.preprocess:
97          os.remove(file_path_to_compile)
98
99      avg_compile_time = sum(compile_times) / float(num_reps)
100     avg_exec_time = sum(execution_times) / float(num_reps)
```

```
101     return avg_compile_time, avg_exec_time
102 
103 def main() -> None:
104     parser = argparse.ArgumentParser("Compile and run source files for multiple compilers")
105     parser.add_argument("--src", help="Paths to the source files", nargs='+', required=True)
106     parser.add_argument("--n", help="Number of repetitions of the experiment (the resulting plot is an
            average)", required=False, default=10)
107     args = parser.parse_args()
108     file_paths = args.src
109     num_reps = int(args.n)
110 
111     compilations = []
112     for file_path in file_paths:
113         compilations.append([
114             Compilation("$CLANGIR_INSTALLDIR/bin/clang-19", file_path=file_path, flags=["-fopenmp", "-O2"]),
115             Compilation("gcc", file_path=file_path, flags=["-fopenmp", "-O2"]),
116             Compilation("$CLANGIR_INSTALLDIR/bin/clang-19", file_path=file_path, flags=["-fclangir", "-
                fopenmp-enable-irbuilder", "-fopenmp", "-O2"], preprocess=True)
117         ])
118 
119     compiler_names = ["clang", "gcc", "clang-CIR"]
120     all_compilation_times = []
121     all_execution_times = []
122 
123     for file_compilations in compilations:
124         file_compilation_times = []
125         file_execution_times = []
126         for compilation in file_compilations:
127             avg_compile_time, avg_exec_time = compute_times(compilation, num_reps)
128             file_compilation_times.append(avg_compile_time)
129             file_execution_times.append(avg_exec_time)
130         all_compilation_times.append(file_compilation_times)
131         all_execution_times.append(file_execution_times)
132 
133     plot_times(file_paths, compiler_names, all_compilation_times, all_execution_times, num_reps)
134 
135 if __name__ == "__main__":
136     main()
```

Listing 27: Compilation script

A.4.2 Sample code

The experimentation involved four OpenMP parallelized versions of fundamental algorithms: matrix multiplication, merge sort, searching for a value in an unsorted array, and computing the sum of array elements. Despite not optimizing all parameters or achieving maximal parallelism, these implementations provide a basis for comparing execution and compilation times across three different compilers.

These algorithms, which operate on basic one-dimensional arrays, employ a technique known as 'Chunking'—dividing arrays into sub-arrays. Notably, the merge sort implementation synchronizes all threads at each level of the recursion tree using taskwait, introducing overhead from creating and synchronizing numerous small tasks. It's important to recognize that these implementations may not be optimal due to this overhead.

However, the primary goal of these sample codes is to demonstrate the correct implementation of various constructs discussed in the thesis, rather than achieving peak performance.

```
#include <omp.h>
#define SIZE 100000
#define CHUNK_SIZE 1000

int main() {
    int arr[SIZE];
    for (int i = 0; i < SIZE; i+=1)
        arr[i] = i + 1;

    int sum = 0;
    int num_chunks = (SIZE + CHUNK_SIZE - 1) / CHUNK_SIZE;
    int chunk_indices[num_chunks];
    int current_chunk = 0;

    // Initialize chunk indices
    for (int i = 0; i < num_chunks; i+=1) {
        chunk_indices[i] = i * CHUNK_SIZE;
    }

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < num_chunks; i+=1) {
                #pragma omp task
                {
                    int local_sum = 0;
                    int chunk_start, chunk_end;

                    // Get the next chunk to process
                    int local_current_chunk;
                    #pragma omp critical
                    {
                        local_current_chunk = current_chunk;
                        current_chunk+=1;
                    }

                    // Calculate chunk bounds
                    chunk_start = chunk_indices[local_current_chunk];
                    chunk_end = chunk_start + CHUNK_SIZE;
                    if (chunk_end > SIZE) chunk_end = SIZE;

                    // Process the chunk and compute local sum
                    for (int j = chunk_start; j < chunk_end; j+=1) {
                        local_sum += arr[j];
                    }

                    #pragma omp critical
                    {
                        sum += local_sum;
                    }
                }
            }
```

```
54              #pragma omp taskwait
55          }
56      }
57
58      return sum;
59  }
```

Listing 28: Parallel summation of an array

```
#include <omp.h>
#define SIZE 100000
#define CHUNK_SIZE 1000
#define TARGET 93129

int main() {
    int arr[SIZE];
    for (int i = 0; i < SIZE; i+=1)
        arr[i] = i + 1;

    int found_index = -1;
    int num_chunks = (SIZE + CHUNK_SIZE - 1) / CHUNK_SIZE;
    int chunk_indices[num_chunks];
    int current_chunk = 0;

    // Initialize chunk indices
    for (int i = 0; i < num_chunks; i+=1) {
        chunk_indices[i] = i * CHUNK_SIZE;
    }

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < num_chunks; i+=1) {
                #pragma omp task
                {
                    int chunk_start, chunk_end;
                    int local_found_index = -1;

                    // Get the next chunk to process
                    int local_current_chunk;
                    #pragma omp critical
                    {
                        local_current_chunk = current_chunk;
                        current_chunk+=1;
                    }

                    // Calculate chunk bounds
                    chunk_start = chunk_indices[local_current_chunk];
                    chunk_end = chunk_start + CHUNK_SIZE;
                    if (chunk_end > SIZE) chunk_end = SIZE;

                    // Process the chunk
                    for (int j = chunk_start; j < chunk_end; j+=1) {
                        if (arr[j] == TARGET) {
                            local_found_index = j;
                            break; // Exit the loop as soon as the target is found
                        }
                    }

                    // Update global found_index if a target is found
                    if(local_found_index != -1){
                        #pragma omp critical
                        {
                            if (found_index == -1) {
```

```
                            found_index = local_found_index;
                        }
                    }
                }
            }
            #pragma omp taskwait
        }
    }
    return found_index;
}
```

Listing 29: Parallel search of element in an unsorted array

```
#include <omp.h>
#define N 500
#define M 500
#define P 500
#define CHUNK_SIZE 50

int main() {
    int A[N][M], B[M][P], C[N][P];

    for (int i = 0; i < N; i+=1)
        for (int j = 0; j < M; j+=1)
            A[i][j] = i + j;

    for (int i = 0; i < M; i+=1)
        for (int j = 0; j < P; j+=1)
            B[i][j] = i - j;

    for (int i = 0; i < N; i+=1)
        for (int j = 0; j < P; j+=1)
            C[i][j] = 0;

    // Calculate chunk indices for rows of C
    int num_chunks_i = (N + CHUNK_SIZE - 1) / CHUNK_SIZE;
    int chunk_indices_i[num_chunks_i];
    for (int i = 0; i < num_chunks_i; i+=1) {
        chunk_indices_i[i] = i * CHUNK_SIZE;
    }

    // Calculate chunk indices for columns of C
    int num_chunks_j = (P + CHUNK_SIZE - 1) / CHUNK_SIZE;
    int chunk_indices_j[num_chunks_j];
    for (int j = 0; j < num_chunks_j; j+=1) {
        chunk_indices_j[j] = j * CHUNK_SIZE;
    }

    #pragma omp parallel
    {
        #pragma omp single
        {
            // Loop over chunk indices for rows of C
            for (int ci = 0; ci < num_chunks_i; ci+=1) {
                int start_i = chunk_indices_i[ci];
                int end_i = start_i + CHUNK_SIZE;
```

```
44              if (end_i > N) end_i = N;
45
46              // Loop over chunk indices for columns of C
47              for (int cj = 0; cj < num_chunks_j; cj+=1) {
48                  int start_j = chunk_indices_j[cj];
49                  int end_j = start_j + CHUNK_SIZE;
50                  if (end_j > P) end_j = P;
51
52                  // Task to compute elements of C for the current chunk
53                  #pragma omp task
54                  {
55                      for (int i = start_i; i < end_i; i+=1) {
56                          for (int j = start_j; j < end_j; j+=1) {
57                              int local_sum = 0;
58                              for (int k = 0; k < M; k+=1) {
59                                  local_sum += A[i][k] * B[k][j];
60                              }
61                              // Critical section to update C[i][j]
62                              #pragma omp critical
63                              {
64                                  C[i][j] += local_sum;
65                              }
66                          }
67                      }
68                  }
69              }
70          }
71          // Wait for all tasks to complete
72          #pragma omp taskwait
73      }
74  }
75
76  int sum = 0;
77  for (int i = 0; i < N; i+=1) {
78      for (int j = 0; j < P; j+=1) {
79          sum += C[i][j];
80      }
81  }
82  return sum;
83 }
```

Listing 30: Parallel matrix multiplication

```
#include <omp.h>
#define ARRAY_SIZE 100000

void merge(int *array, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    int L[n1], R[n2];

    for (int i = 0; i < n1; i+=1) {
        L[i] = array[left + i];
    }
    for (int j = 0; j < n2; j+=1) {
        R[j] = array[mid + 1 + j];
    }

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            #pragma omp critical
            {
                array[k] = L[i];
                i+=1;
            }
        } else {
            #pragma omp critical
            {
                array[k] = R[j];
                j+=1;
            }
        }
        k+=1;
    }

    while (i < n1) {
        #pragma omp critical
        {
            array[k] = L[i];
            i+=1;
            k+=1;
        }
    }

    while (j < n2) {
        #pragma omp critical
        {
            array[k] = R[j];
            j+=1;
            k+=1;
        }
    }
}

void parallel_merge_sort(int *array, int left, int right) {
    if (left < right) {
```

```
57          int mid = left + (right - left) / 2;
58
59          #pragma omp taskgroup
60          {
61              #pragma omp task
62              parallel_merge_sort(array, left, mid);
63              #pragma omp task
64              parallel_merge_sort(array, mid + 1, right);
65
66              // Wait for all tasks to complete
67              #pragma omp taskwait
68
69              merge(array, left, mid, right);
70          }
71      }
72  }
73
74  int is_sorted(int *array, int size) {
75      for (int i = 0; i < size - 1; i+=1) {
76          if (array[i] > array[i + 1]) {
77              return 0;
78          }
79      }
80      return 1;
81  }
82
83  int main() {
84      int array[ARRAY_SIZE];
85
86      for (int i = 0; i < ARRAY_SIZE; i+=1) {
87          array[i] = ARRAY_SIZE - i;
88      }
89
90      #pragma omp parallel
91      {
92          #pragma omp single
93          parallel_merge_sort(array, 0, ARRAY_SIZE - 1);
94      }
95
96      int sorted = is_sorted(array, ARRAY_SIZE);
97      return sorted;
98  }
```

Listing 31: Parallel merge sort

References

[1] A. H. Bravo, *An experimental guided approach to the metric dimension on different graph families*, `https://upcommons.upc.edu/bitstream/handle/2117/407759/183222.pdf?sequence=2&isAllowed=y`, Trabajo final de grado. Accessed on: 17/05/2024, 2024.

[2] *Gcc gnu project homepage*, `https://gcc.gnu.org/`, Accessed: 23/02/2024.

[3] *Llvm project homepage*, `https://llvm.org/`, Accessed: 23/02/2024.

[4] *Api definition*, https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces, Accessed: 23/02/2024.

[5] *Clangir documentation*, https://llvm.github.io/clangir/, Accessed: 23/02/2024.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd. Boston: Addison-Wesley, 2006.

[7] *The syntax of the c language in backus-naur form*, https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/ThesyntaxofCinBackus-Naurform.htm, Accessed: 23/02/2024.

[8] *Jochen burghardt representation of front-end*, https://en.wikipedia.org/wiki/Compiler, Accessed: 23/02/2024.

[9] *Mojo programming language*, https://www.modular.com/max/mojo, Accessed on: 20/06/2024.

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd. Addison-Wesley, 2006.

[11] *Intermediate Representation - 2*, https://archive.nptel.ac.in/content/storage2/courses/106104072/chapter_2/2_29.htm, Accessed: 23/02/2024.

[12] S. Sarda, *LLVM Essentials*, 1st. Packt Publishing, 2015.

[13] R. Eklind, *Llvm ir explanation and go*, https://blog.gopheracademy.com/advent-2018/llvm-ir-and-go/, Accessed on: 27/02/2024.

[14] W. Moses, *Custom MLIR compilation flow*, https://www.researchgate.net/figure/The-Polygeist-compilation-flow-consists-of-4-stages-The-frontend-traverses-Clang-AST-to_fig1_355432602, Accessed on: 27/02/2024.

[15] C. Lattner and J. Pienaar, *Mlir primer: A compiler infrastructure for the end of moore's law*, https://pdfs.semanticscholar.org/d43c/f123628324291328ddd0b3966dfa6f338e25.pdf, Accessed on: 27/02/2024.

[16] F. Mora, *Parallel openmp implementation on clangir*, https://github.com/llvm/clangir/commit/d129b7656fdeae74e45cdc95bc22154219511c77, Accessed: 25/02/2024.

[17] LLVM, *LLVM Project*, https://github.com/llvm/llvm-project, Accessed on: 26/02/2024.

[18] LLVM, *ClangIR Project*, https://github.com/llvm/clangir, Accessed on: 26/02/2024.

[19] *Overleaf editor*, https://www.overleaf.com/project, Accessed on: 29/2/2024.

[20] *Cmake documentation*, https://cmake.org/, Accessed on: 29/2/2024.

[21] *Ninja build manual*, https://ninja-build.org/manual.html, Accessed on: 29/2/2024.

[22] *Kaleidoscope tutorial*, https://llvm.org/docs/tutorial/, Accessed on: 01/03/2024.

[23] F. M. Q. P. et al., *Llvm introduction course by compilers lab*, https://www.youtube.com/playlist?list=PLDSTpI7ZVmVnvqtebWnnI8YeB8bJoGOyv, Accessed on: 01/03/2024.

[24] *Mlir toy tutorial*, https://mlir.llvm.org/docs/Tutorials/Toy/, Accessed on: 01/03/2024.

[25] L. Zhang, *Lei.chat*, https://www.lei.chat/, Accessed on: 01/03/2024.

[26] E. Meardon, *Gantt chart: A guide to agile project management*, https://www.atlassian.com/es/agile/project-management/gantt-chart, Accessed on: 01/03/2024.

[27] *Mlir documentation*, https://mlir.llvm.org/docs/, Accessed on: 01/03/2024.

[28] *Openmp documentation*, https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf, Accessed on: 01/03/2024.

[29] B. C. Lopes, *Rfc: Upstreaming clangir*, https://discourse.llvm.org/t/rfc-upstreaming-clangir/76587/25, Accessed on: 18/03/2024.

[30] *Userbenchmark - asus zenbook ux425ua-um425ua*, https://www.userbenchmark.com/System/Asus-ZenBook-UX425UA-UM425UA/224205, Accessed on: 01/03/2024.

[31] *Glassdoor*, https://www.glassdoor.es/Empleo/index.htm, Accessed on: 01/03/2024.

[32] *Glassdoor - spain software engineer salary*, https://www.glassdoor.com/Salaries/spain-software-engineer-salary-SRCH_IL.0,5_IN219_KO6,23.htm, Accessed on: 01/03/2024.

[33] *Glassdoor - compiler engineer salary*, https://www.glassdoor.com/Salaries/compiler-engineer-salary-SRCH_KO0,17.htm, Accessed on: 01/03/2024.

[34] *Glassdoor - barcelona supercomputing center compiler developer salaries*, https://www.glassdoor.es/Sueldo/Barcelona-Supercomputing-Center-Compiler-Developer-Sueldos-E382342_D_KO32,50.htm, Accessed on: 01/03/2024.

[35] *Glassdoor - spain project manager salary*, https://www.glassdoor.com/Salaries/spain-project-manager-salary-SRCH_IL.0,5_IN219_KO6,21.htm, Accessed on: 01/03/2024.

[36] *Asus zenbook 14 um425ua*, https://www.asus.com/es/laptops/for-home/zenbook/zenbook-14-um425-ua/, Accessed on: 03/03/2024.

[37] *Tarifa luz hora*, https://tarifaluzhora.es/, Accessed on: 03/03/2024.

[38] Gobierno de España, *Instrucciones para el cálculo de las emisiones de gases de efecto invernadero de la administración general del estado y sus organismos públicos*, https://www.miteco.gob.es/content/dam/miteco/es/cambio-climatico/temas/mitigacion-politicas-y-medidas/instruccionescalculadorahc_tcm30-485627.pdf, Accessed on: 04/03/2024.

[39] Foretica, *Informe: Mitos y Verdades sobre Plásticos*, https://foretica.org/wp-content/uploads/2020/06/Informe_Mitos_y_Verdades_Plasticos_Foretica_2020.pdf, Accessed on: 17/03/2024, 2020.

[40] *Environmental paper network - global paper calculator*, https://c.environmentalpaper.org/target2, Accessed on: 18/03/2024.

[41] Boletín Oficial del Estado (BOE), *Ley 14/2011, de 1 de junio, de la Ciencia, la Tecnología y la Innovación*, https://www.boe.es/diario_boe/txt.php?id=BOE-A-2011-9617, Accessed on: 12/05/2024.

[42] T. A. I. C. Team, *Clang - comparison*, https://opensource.apple.com/source/clang/clang-23/clang/tools/clang/www/comparison.html, Accessed on: 18/05/2024.

[43] T. C. Team, *Introduction to the clang ast*, https://clang.llvm.org/docs/IntroductionToTheClangAST.html, LLVM Project. Accessed on: 18/05/2024.

[44] C. Dixit, S. Chattopadhyay, and A. Sanyal, *Ir2vec: A flow analysis based scalable infrastructure for program encodings*, Accessed on: 16/05/2024, IEEE, 2019.

[45] Q. Yi, *Cs 5363: Advanced compiler construction*, http://www.cs.uccs.edu/~qyi/UTSA-classes/cs5363/index.htm, Accessed on: 16/05/2024.

[46] M. Rathi, *Llvm ir tutorial: The what, why, and how*, https://mukulrathi.com/create-your-own-programming-language/llvm-ir-cpp-api-tutorial/, Accessed on: 16/05/2024.

[47] Y. Shan, *Mlir*, `http://lastweek.io/notes/MLIR/`, Yizhou Shan's Home Page. Accessed on: 2024/05/17.

[48] M. Community, *Creating a custom dialect in mlir*, `https://mlir.llvm.org/docs/Tutorials/CreatingADialect/`, MLIR Documentation. Accessed on: 17/05/2024.

[49] B. C. Lopes, *An mlir-based clang ir (cir)*, `https://discourse.llvm.org/t/rfc-an-mlir-based-clang-ir-cir/63319`, Accessed on: 17/05/2024.

[50] T. C. Project, *Memory safety - chromium security*, `https://www.chromium.org/Home/chromium-security/memory-safety/`, Accessed on: 17/05/2024.

[51] N. M. Ali and A. M. Rahma, "An improved aes encryption of audio wave files," Ph.D. dissertation, Apr. 2015.

[52] J. Doerfert, *A compiler's view of openmp*, `https://www.openmp.org/wp-content/uploads/A-compilers-view-of-OpenMP.pdf`, LLVM Developers' Meeting 2020. Acessed on: 18/05/2024.

[53] F. Mora, *Phd thesis university of delaware*, Accessed: 17/06/2024. [Online]. Available: `https://fabianmcg.github.io/`.

[54] B. Cardoso, *Bruno's website*, Accessed: 17/06/2024. [Online]. Available: `http://brunocardoso.cc/`.

[55] K. Chandramohan, *Google scholar profile*, Accessed: 17/06/2024. [Online]. Available: `https://scholar.google.com.cu/citations?user=gDy3jSMAAAAJ&hl=es`.

[56] W. T. Vargas, *Clangir tfg fork*, Accessed: 17/06/2024. [Online]. Available: `https://github.com/eZWALT/clangir/tree/tfg_branch`.

[57] K. Raj, *Gcc/clang optimizations for embedded linux - khem raj, comcast rdk*, YouTube, Accessed on: 20/06/2024, 2019. [Online]. Available: `https://www.youtube.com/watch?v=jVYnT_onb70&ab_channel=TheLinuxFoundation`.

[58] A. Tech, *Gcc vs clang/llvm: An in-depth comparison of c/c++ compilers*, `https://alibabatech.medium.com/gcc-vs-clang-llvm-an-in-depth-comparison-of-c-c-compilers-899ede2be378`, Accessed on: 20/06/2024, Aug. 2019.

[59] *LLVM lit*, `https://llvm.org/docs/CommandGuide/lit.html`, Accessed on: 10/05/2024.

[60] *Llvm contribution guidelines*, `https://llvm.org/docs/Contributing.html`, Accessed on: 10/05/2024.

[61] Atlassian, *Git Rebase*, `https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase`, Accessed on: 10/05/2024.