Enhancing The Clang Compiler with OpenMP Parallelism: A Novel Approach using ClangIR

Walter J. Troiani Vargas
Director: José Miguel Rivero Almeida
Co-Supervisor: Roger Ferrer Ibáñez

Department of Computer Science (CS)
Barcelona Supercomputing Center (BSC)

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Table of Contents

Introduction

Are compilers important in Computer Science?

Introduction

Are compilers important in Computer Science?

- Do you love fast, highly parallel, correct, and optimized code?

Introduction

Are compilers important in Computer Science?

- Do you love fast, highly parallel, correct, and optimized code?
- Do you appreciate debugging errors and warnings?

Introduction

Are compilers important in Computer Science?

- Do you love fast, highly parallel, correct, and optimized code?
- Do you appreciate debugging errors and warnings?
- Do you enjoy programming in high-level languages? (C/C++/Rust...)

Introduction

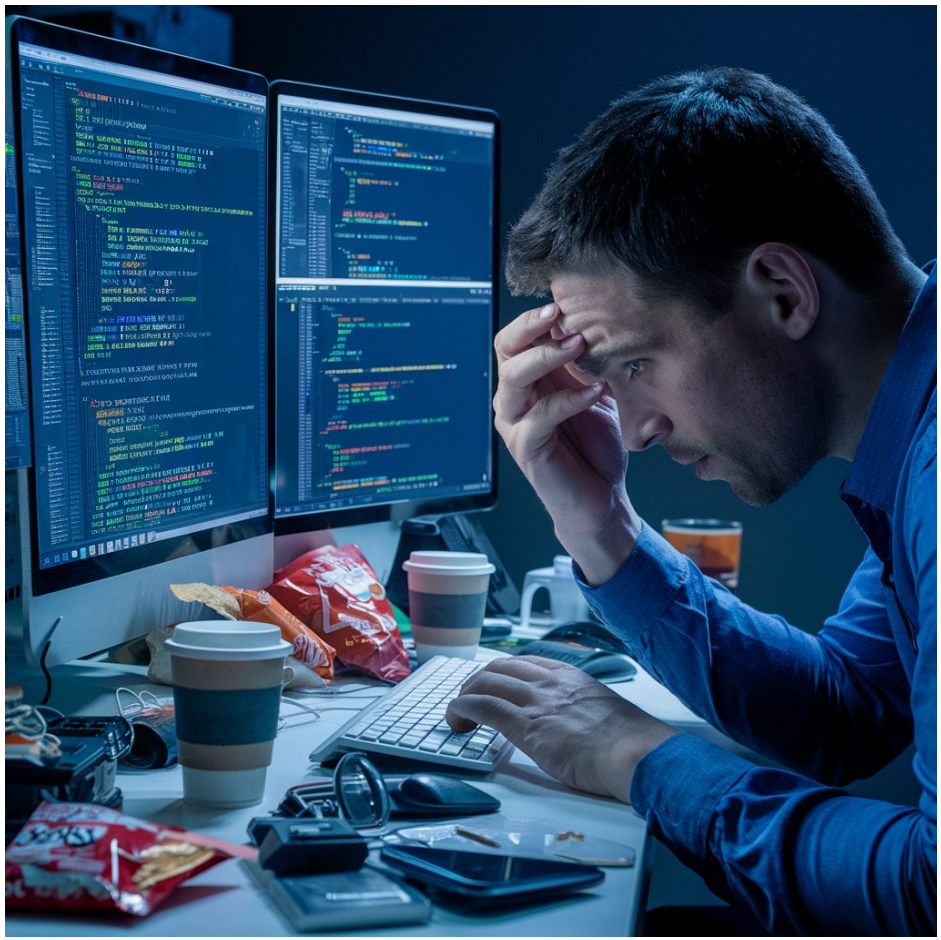
Are compilers important in Computer Science?

- Do you love fast, highly parallel, correct, and optimized code?
- Do you appreciate debugging errors and warnings?
- Do you enjoy programming in high-level languages? (C/C++/Rust...)

If so, then you already have an appreciation for the role of compilers!

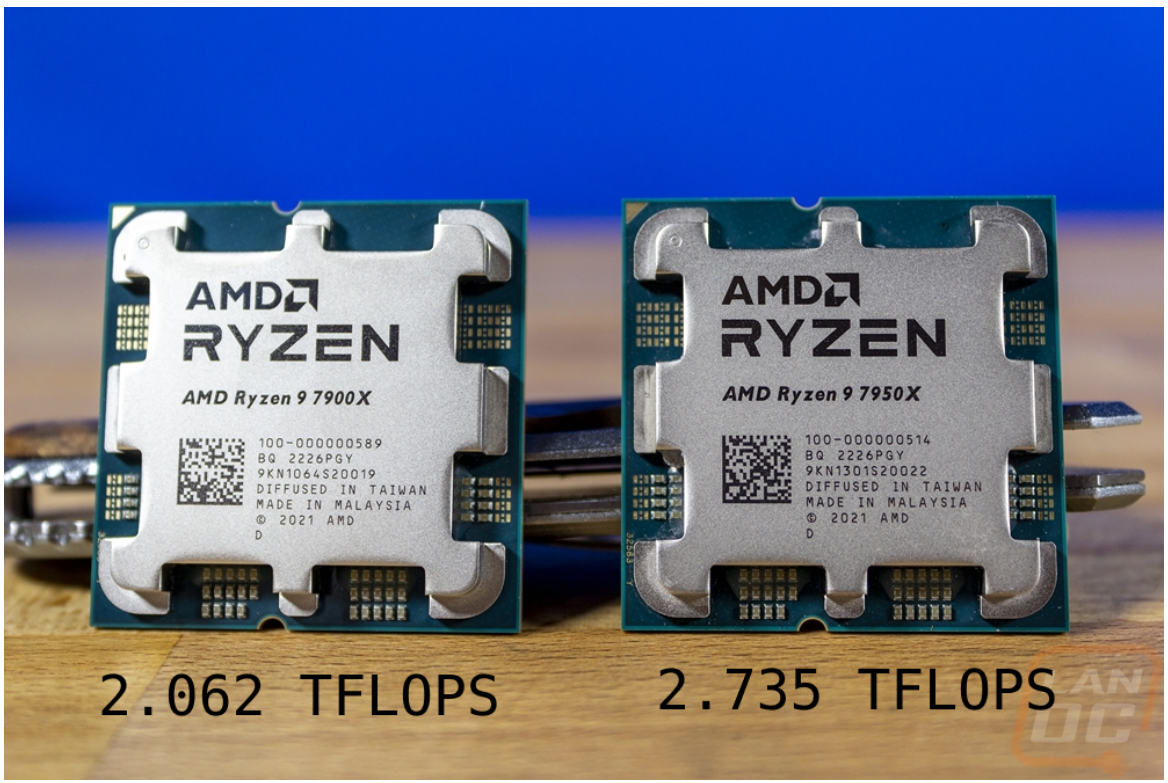
Introduction

Unless you're a fan of raw x86 programming (like me), these tools are indispensable.



The great bottleneck

Taking a quick look at modern hardware, it's clear that its immense capacity is rarely used to its fullest potential. There's a need to enhance the software to mitigate this bottleneck

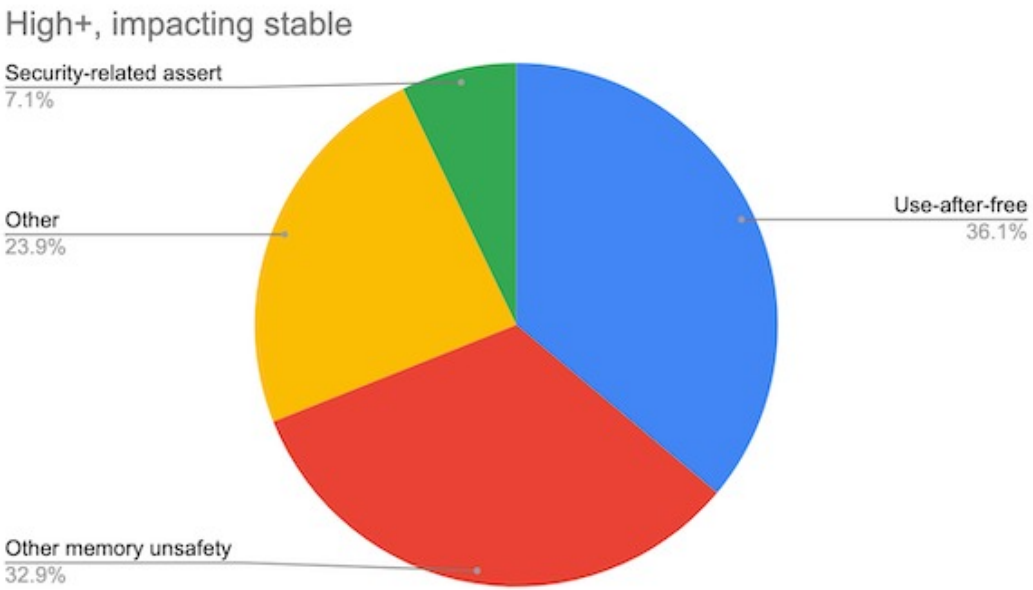


Modern compilers limitations

```
int *may_explode() {
  int *p = nullptr;
  {
    int x = 0;
    p = &x;
    *p = 42;
  }
  *p = 42; // oops...
}
```

Modern compilers limitations

Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.

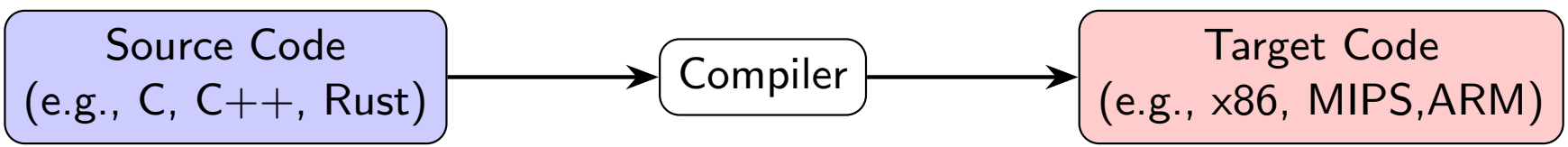


The Chromium project report

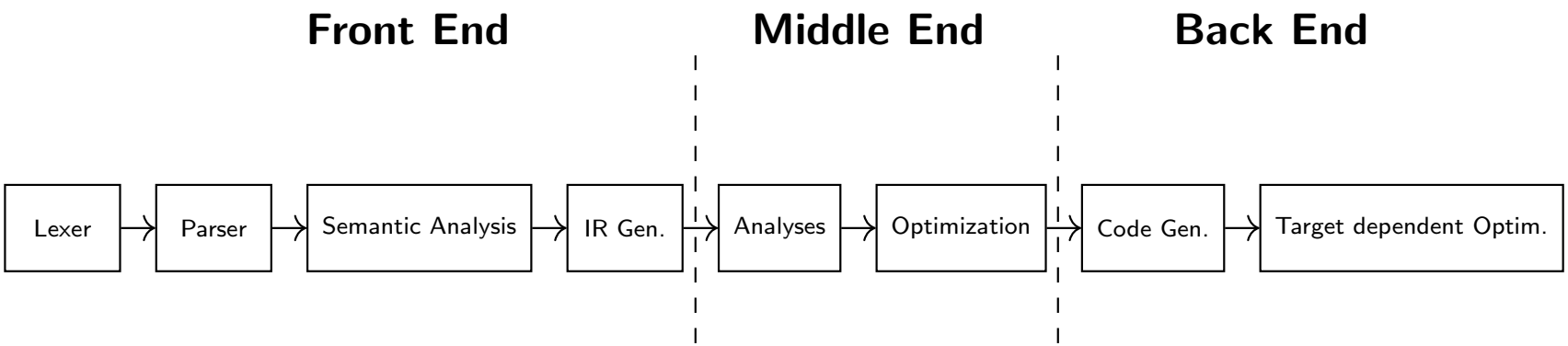
What is a compiler?

Definition

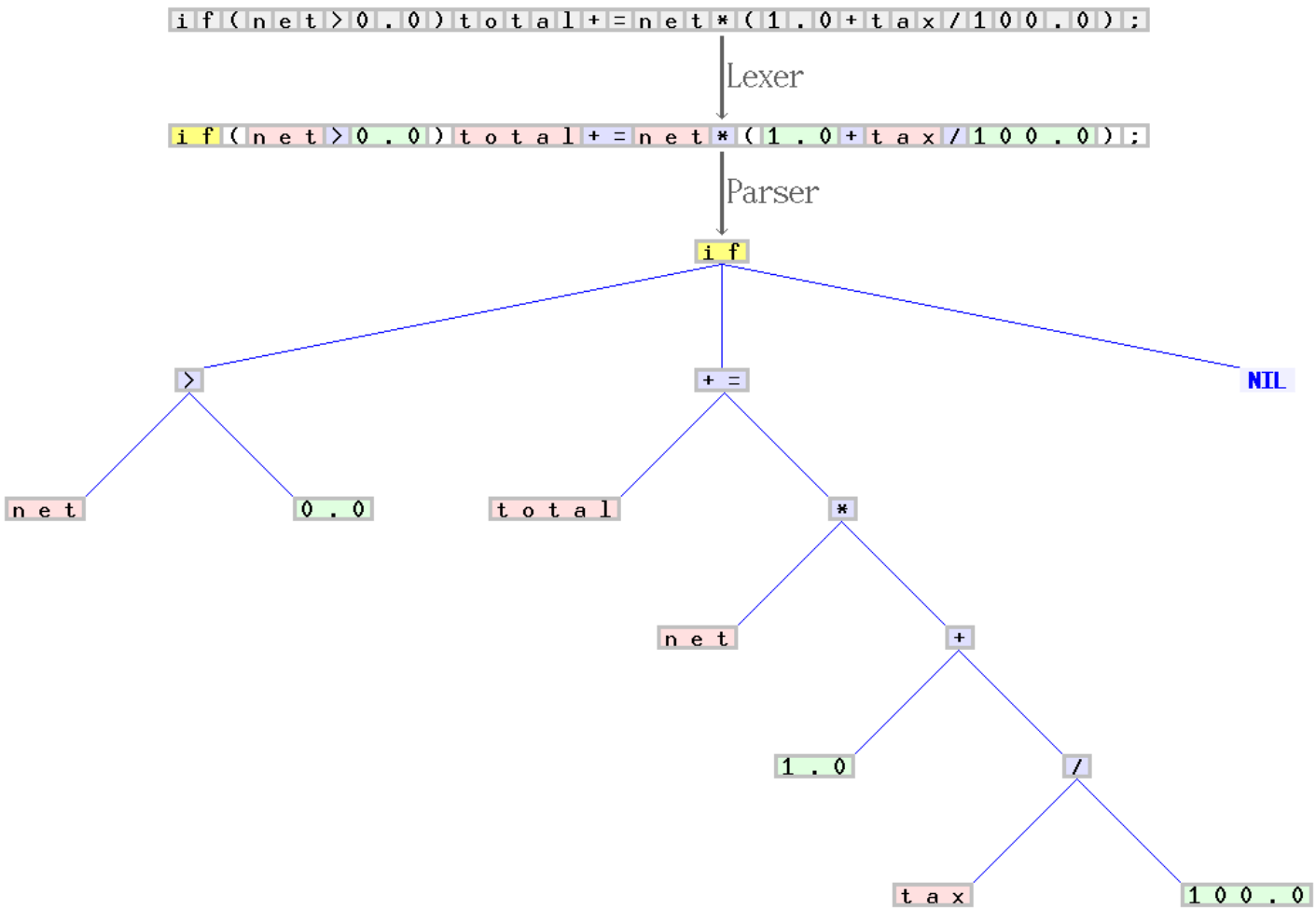
A compiler is a software that translates code from one programming language (Source) into another language (Target). During this process, it may also perform transformations and optimizations.



Typical Compilation Flow

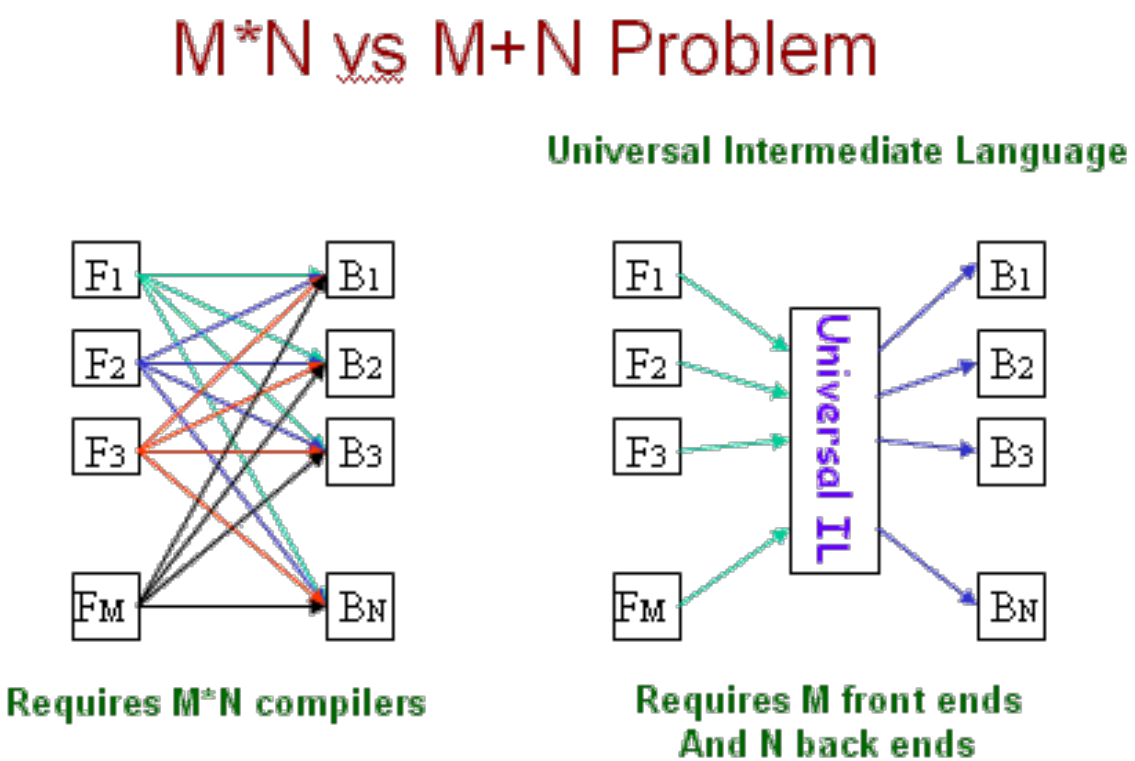


Front-end


if (net > 0.0) total += net * (1.0 + tax / 100.0) ;
Lexer
if (net > 0.0) total += net * (1.0 + tax / 100.0) ;
Parser
if
>
+=
NIL
net
0.0
total
*
net
+
1.0
/
tax
100.0

Middle-end: IR

Intermediate representations (IRs) are a crucial abstraction for providing a unified back-end infrastructure and avoiding redundancy.



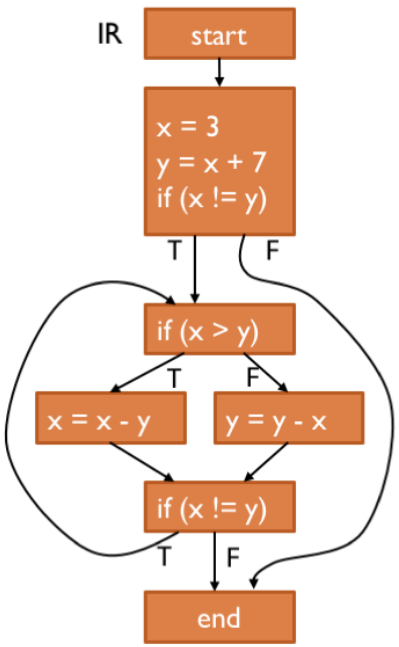
Middle-end: Optimizations

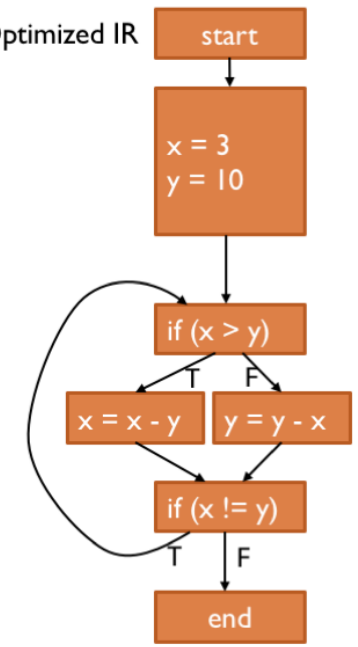
1. Programmers are lazy (often write sub-optimal code)
2. IR introduces extra instructions
3. Prevents engineering redundancy in the back-end

Putting It All Together: GCD

Source code

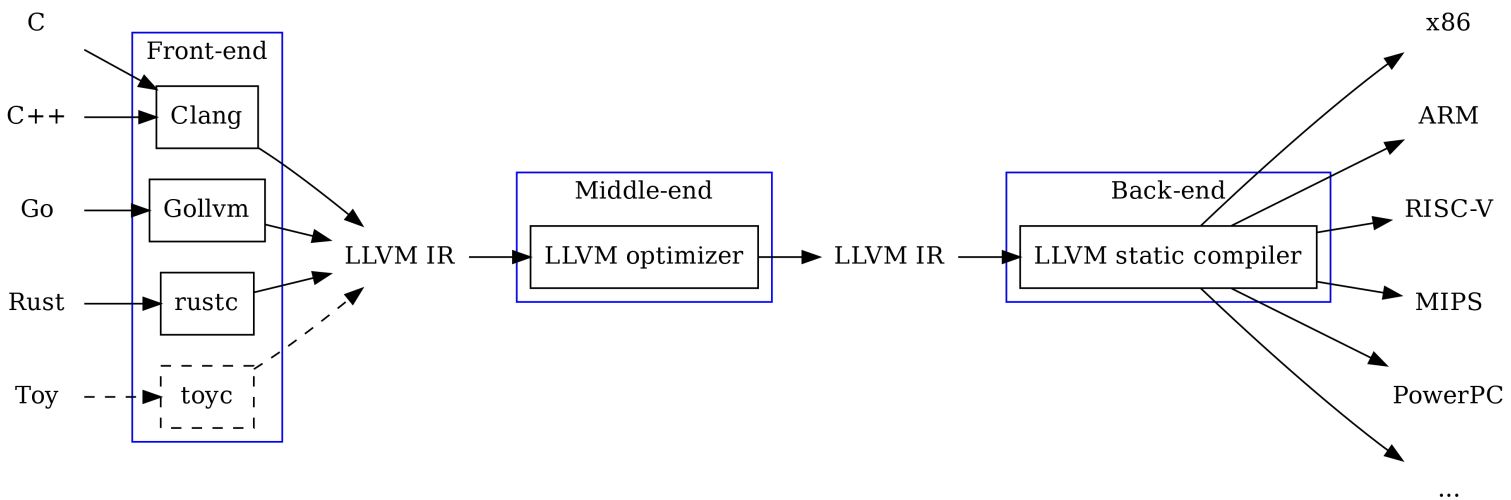
```
int x = 3;
int y = x + 7;
while (x != y) {
  if (x > y) {
    x = x - y;
  } else {
    y = y - x;
  }
}
```





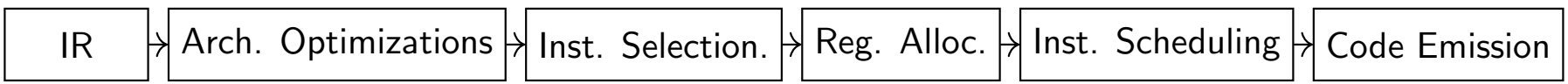
Back-end: Motivation

IRs are great, but processors can't directly execute them!



Back-end: Pipeline

Back End



Objectives

Goal A

Provide a comprehensive resource for understanding the modern compiler ecosystem, including Clang, LLVM, OpenMP, MLIR, and ClangIR.

Objectives

Goal A

Provide a comprehensive resource for understanding the modern compiler ecosystem, including Clang, LLVM, OpenMP, MLIR, and ClangIR.

Goal B

Implement OpenMP constructs (Initially non-existent) in the novel ClangIR intermediate representation to accelerate its development and release.

The LLVM Project

Umbrella project for compiler infrastructure

- Front-end: Clang and Flang

The LLVM Project

Umbrella project for compiler infrastructure

- Front-end: Clang and Flang
- Debugger: LLDB

The LLVM Project

Umbrella project for compiler infrastructure

- Front-end: Clang and Flang
- Debugger: LLDB
- Linker: LLD

The LLVM Project

Umbrella project for compiler infrastructure

- Front-end: Clang and Flang
- Debugger: LLDB
- Linker: LLD
- IR: LLVM IR, MLIR...

The LLVM Project

Umbrella project for compiler infrastructure

- Front-end: Clang and Flang
- Debugger: LLDB
- Linker: LLD
- IR: LLVM IR, MLIR...
- Parallel programming: OpenMP Runtime

The LLVM Project

Umbrella project for compiler infrastructure

- Front-end: Clang and Flang
- Debugger: LLDB
- Linker: LLD
- IR: LLVM IR, MLIR...
- Parallel programming: OpenMP Runtime
- Heterogeneous programming: OpenCL library
- ...

Open Multi-Processing (OpenMP)

- API for shared memory multiprocessing programming.
- Out-of-the-box Heterogeneous Parallelism (CPU, GPU, TPU...) via simple directives

```
#pragma omp parallel for reduction(+:sum)
for (int i = 1; i <= n; ++i) {
    sum += X[i];
}
```

The Clang front-end

Compiler front-end for the C-based language families: C, C++, Objective-C, Objective-C++, Swift, CUDA, OpenCL...

LLVM Intermediate Representation (LLVM-IR)

- Universal low-level IR for the LLVM project, acting as a bridge for back-ends, optimizations, OpenMP, Clang...
- Typed, assembly-like language that generalizes assembly languages.

```
int main() {
    float x = 3.14159f;
    float y = 420.0f;
    float result = add(x, y);
    return (int)result;
}
```

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %x = alloca float, align 4
  %y = alloca float, align 4
  store i32 0, i32* %1, align 4
  store float 0x40091EB860000000, float* %x, align 4
  store float 4.200000e+02, float* %y, align 4
  %x2 = load float, float* %x, align 4
  %y2 = load float, float* %y, align 4
  %result = call float @add(float %x2, float %y2)
  %ret = fptosi float %result to i32
  ret i32 %ret
}
```

LLVM IR Drawbacks

1. **Loss of high-level semantics**: Classes, templates, and exceptions are not represented.

LLVM IR Drawbacks

1. **Loss of high-level semantics**: Classes, templates, and exceptions are not represented.
2. **Too abstract and generic**: Loses touch with the original language (like Fortran or C++), making language-specific optimizations difficult to implement.

LLVM IR Drawbacks

1. **Loss of high-level semantics**: Classes, templates, and exceptions are not represented.
2. **Too abstract and generic**: Loses touch with the original language (like Fortran or C++), making language-specific optimizations difficult to implement.
3. **Flattened control flow**: Original program's control flow context is lost or distiled.

LLVM IR Drawbacks

1. **Loss of high-level semantics**: Classes, templates, and exceptions are not represented.
2. **Too abstract and generic**: Loses touch with the original language (like Fortran or C++), making language-specific optimizations difficult to implement.
3. **Flattened control flow**: Original program's control flow context is lost or distiled.

```
define dso_local i32 @main() #0 {
  ...
  %3 = alloca float, align 4
  store i32 0, i32* %1, align 4
  store float 0x40091EB860000000, float* %2, align 4
  store float 4.200000e+02, float* %3, align 4
  ...
}
```

LLVM IR IS TOO LOW LEVEL

Multi-Level Intermediate Representation (MLIR)

Overview

Abstraction that manages multiple high-level IRs (dialects). Each dialect has its own set of types, attributes and operations. Through progressive "Lowering" of IR's information loss is mitigated.

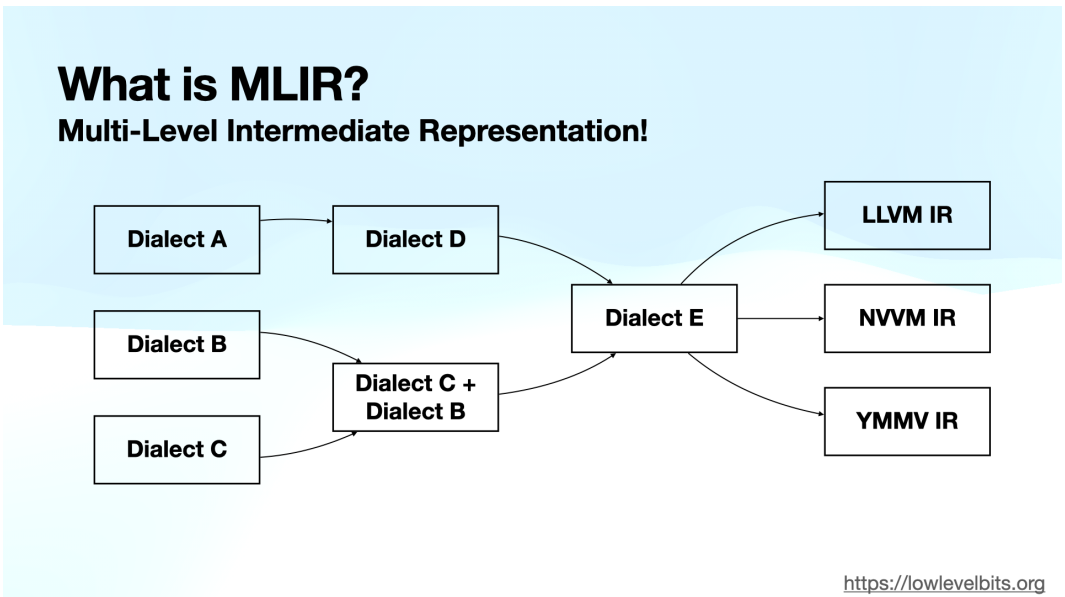
Multi-Level Intermediate Representation (MLIR)

Overview

Abstraction that manages multiple high-level IRs (dialects). Each dialect has its own set of types, attributes and operations. Through progressive "Lowering" of IR's information loss is mitigated.

Examples

arith, memref, tensor, linalg, omp, nvgpu, amdgpu, x86_vector...



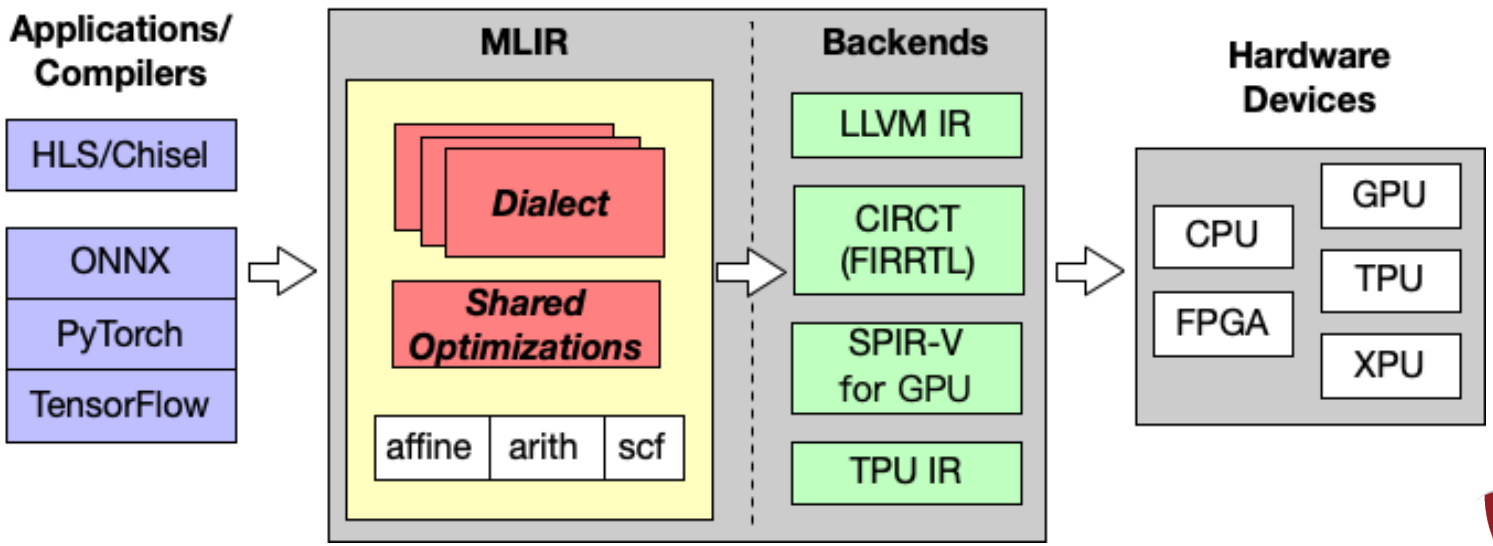
MLIR: Example

```
module {
  func @main() -> i32 {
    %c0 = arith.constant 0 : i32
    %mem = memref.alloca() : memref<1xf32>

    %val1 = arith.constant 3.141592653589793 : f32
    %val2 = arith.constant 420.0 : f32
    %result = arith.addf %val1, %val2 : f32
    memref.store %result, %mem[%c0] : memref<1xf32>

    %loaded_val = memref.load %mem[%c0] : memref<1xf32>
    %res_i32 = arith.fptosi %loaded_val : f32 to i32
    return %res_i32 : i32
  }
}
```

MLIR: Purpose

Purpose

Designed for extensibility and reusability, MLIR supports the creation of IR's and Domain Specific Languages (DSLs) and improves compiler development infrastructure, particularly for AI/ML and heterogeneous hardware.



MLIR: Language specific IR's

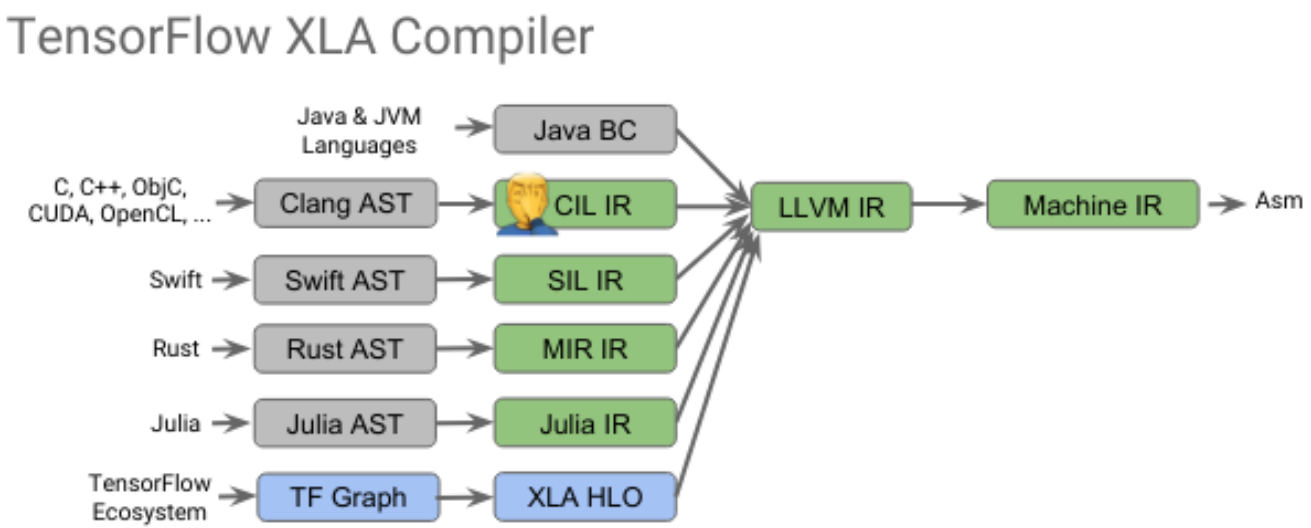
Domain specific IR

MLIR has been used to build high-level IR's to tackle all the issues of using the low-level LLVM-IR at the cost of adding more layers of abstraction.

MLIR: Language specific IR's

Domain specific IR

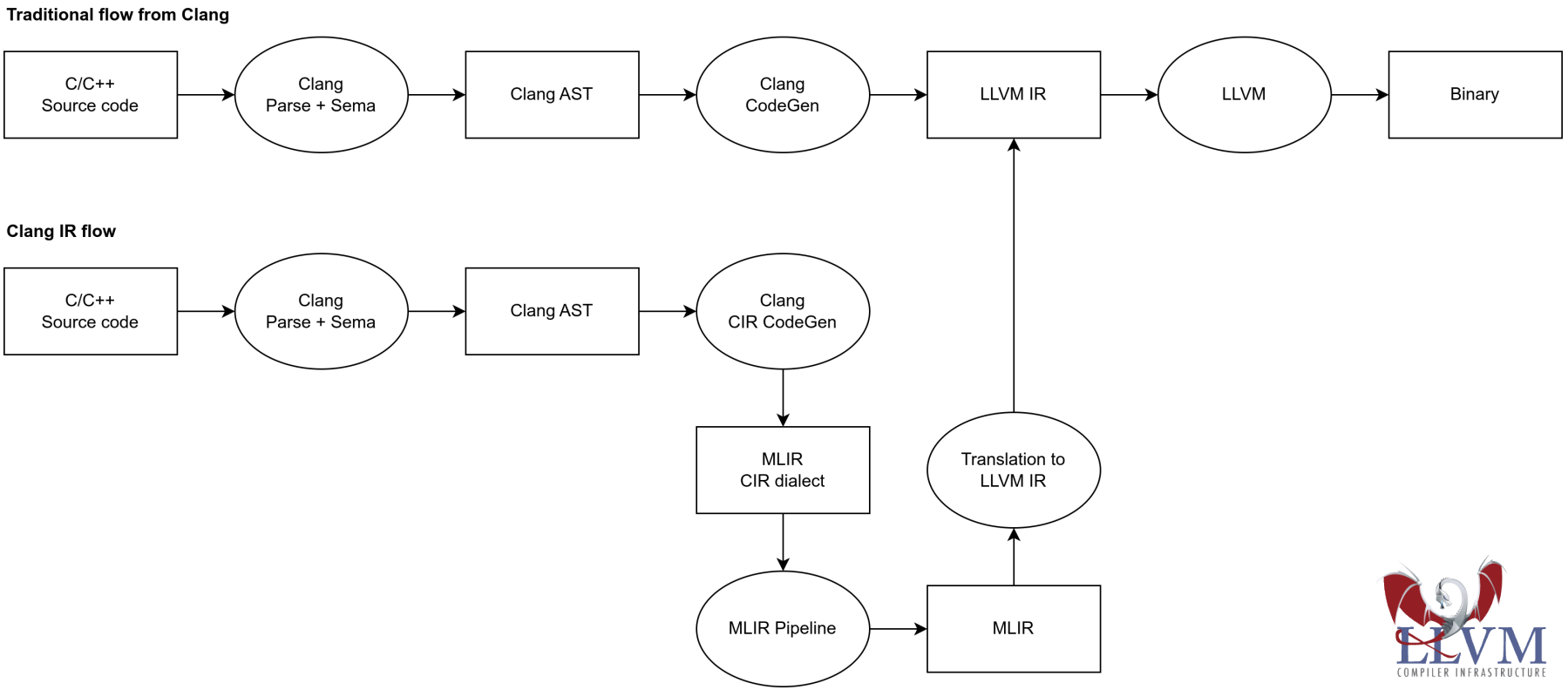
MLIR has been used to build high-level IR's to tackle all the issues of using the low-level LLVM-IR at the cost of adding more layers of abstraction.



ClangIR: Overview

ClangIR

A high-level, Clang-specific intermediate representation (IR) built using MLIR.
Initiated in 2022, ClangIR is the way of the future for C-based compiler technology.



ClangIR: Charter

Potential Benefits

Enhanced detection of dangling pointers, elimination of unnecessary and costly C++ copies, improved control flow structures and loops, enforced const-ness, better diagnostics on data access (Privacy concerns), and C-specific optimizations.

```
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
    %8 = cir.cast(float_to_int, %7 : !cir.float), !cir.int<32>
    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)
}
```

ClangIR: Example of dangling pointer prevention

```
int *may_explode() {
  int *p = nullptr;
  {
    int x = 0;
    p = &x;
    *p = 42;
  }
  *p = 42;
  ...
}
```

```
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>
}
```

Objectives

Goal B

Implement OpenMP constructs (Initially non-existent) in the novel ClangIR intermediate representation to accelerate its development and release.

OpenMP Implementation Coverage

Tasking Constructs

- **Task** ✓
  - if ✓, final ✓, untied ✓
  - default X, mergeable ✓
  - private X, firstprivate X, shared ✓
  - depend X, priority ✓
- **Taskgroup** ✓
- **Taskwait** ✓
- **Taskyield** ✓

Synchronization Constructs

- **Barrier** ✓
- **Critical** ✓
  - name X, hint X
- **Master** ✓
- **Single** ✓
  - private X, firstprivate X, copyprivate X
  - nowait ✓

✓ Implemented
X Not Implemented

OpenMP implementation approaches

Compilation flow change

By diverging from the classic LLVM-IR generation flow to the novel CIR code-generation phase, new methods for handling OpenMP-related AST nodes are required.

OpenMP implementation approaches

Compilation flow change

By diverging from the classic LLVM-IR generation flow to the novel CIR code-generation phase, new methods for handling OpenMP-related AST nodes are required.

Approaches

Two main approaches used in Clang for OpenMP LLVM-IR generation:

1. Basic Code Generation using Pre-Built OpenMP Runtime Routines, **libomp.so** **(LEGACY)**
2. Code Generation through the front-end agnostic **OpenMPIRBuilder** Class **(NEW)**

Chosen Approach: MLIR OMP Dialect

Overview

Integrating the MLIR OMP Dialect inside of ClangIR code-generation, which includes the OpenMPIRBuilder class, provides a straightforward interface for handling OpenMP directives.

Chosen Approach: MLIR OMP Dialect

Overview

Integrating the MLIR OMP Dialect inside of ClangIR code-generation, which includes the OpenMPIRBuilder class, provides a straightforward interface for handling OpenMP directives.

```
...
cir.store %3, %2 : !s32i, !cir.ptr<!s32i>
omp.parallel {
  cir.scope {
    omp.single {
      cir.scope {
        cir.scope {
          %6 = cir.alloca !s32i, !cir.ptr<!s32i>...
          ...
```

Listing: Mixture of ClangIR and OMP dialects

Implementation: Simple directives

```
// Declaration of the code generation
      functions in CIRGenFunction.h
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);
...
```

```
// Adding new statements to the
      generic buildStmt in CIRGenStmt.
      cpp
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));
...
```

Implementation: Simple directives

Directives

Taskwait, Taskyield and Barrier

```
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;
}
...
```

Implementation: Simple directives

```
    //CIRGenOpenMPRuntime.cpp
void CIRGenOpenMPRuntime::emitTaskWaitCall(CIRGenBuilderTy &builder,
                                           CIRGenFunction &CGF,
                                           mlir::Location Loc,
                                           const OMPTaskDataTy &Data) {

  if (!CGF.HaveInsertPoint())
    return;

  if (CGF.CGM.getLangOpts().OpenMPIRBuilder && Data.Dependences.empty()) {
    // TODO(cir): This could change in the near future when OpenMP 5.0 gets
    // supported by MLIR
    builder.create<mlir::omp::TaskwaitOp>(Loc);
  } else {
    llvm_unreachable("NYI");
  }
  assert(!MissingFeatures::openMPRegionInfo());
}
...
```

Implementation: Directives with captured statements

Directives

Task, Taskgroup, Critical, Master, Single

Implementation: Directives with captured statements

```
CIRGenFunction::buildOMPSingleDirective(const OMPSingleDirective &S){
  mlir::LogicalResult res = mlir::success();
  auto scopeLoc = getLoc(S.getSourceRange());

  //Clause handling
  mlir::omp::SingleClauseOps clauseOps;
  CIRClauseProcessor cp = CIRClauseProcessor(*this, S);
  cp.processNowait(clauseOps);
  //Generation of taskgroup op
  auto singleOp = builder.create<mlir::omp::SingleOp>(scopeLoc, clauseOps);
  //Getting the captured statement
  const Stmt* capturedStmt = S.getInnermostCapturedStmt()->getCapturedStmt();

  mlir::Block& block = singleOp.getRegion().emplaceBlock();
  mlir::OpBuilder::InsertionGuard guardCase(builder);
  builder.setInsertionPointToEnd(&block);

  //Build an scope for the single region
  builder.create<mlir::cir::ScopeOp>(
      scopeLoc, /*scopeBuilder=*/
      [&](mlir::OpBuilder &b, mlir::Location loc) {
        LexicalScope lexScope{*this, scopeLoc, builder.getInsertionBlock()};
        //Emit the statement within the single region
        if (buildStmt(capturedStmt, /*useCurrentScope=*/true).failed())
          res = mlir::failure();
      });
  //Add a terminator op to delimit the scope
  builder.create<TerminatorOp>(getLoc(S.getSourceRange().getEnd()));
  return res;
}
```

Implementation: Clause processor

Clauses

If, Final, Priority, Untied, Mergeable, Nowait, Grainsize, NumTasks, Nogroup

Clause handling is repetitive. Therefore, it should be implemented independently of each directive using a dedicated class: **CIRClauseProcessor**.

Implementation: Clause processor

```
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;
}
```

Implementation: Casts and Dialect Conversions

Type Mismatches

One significant challenge encountered during clause implementation was type mismatches between MLIR dialects, given that dialects don't share types.

Example

For instance, in the OMP dialect, booleans are interpreted as 1-bit integers, while ClangIR represents booleans as 8-bit integers (**i8** $\neq$ **i1**).

Implementation: Casts and Dialect Conversions

Type Mismatches

One significant challenge encountered during clause implementation was type mismatches between MLIR dialects, given that dialects don't share types.

Example

For instance, in the OMP dialect, booleans are interpreted as 1-bit integers, while ClangIR represents booleans as 8-bit integers (**i8** $\neq$ **i1**).

Solution

Utilizing MLIR's `builtin unrealized_conversion_cast` + adding a reconciliation pass for unrealized casts, and a new rewrite pattern for this operation to avoid bit-length mismatches.

This avoids introducing new operations to the IR

Implementation: Casts and dialect conversions

Rewrite Patterns

Rewrite patterns enable specifying how operations, such as `builtin.unrealized_conversion_cast`, are treated, similar to using regular expressions, to define actions for specific operations.

```
// 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();
```

Showcase

Unit Testing

Validation

Ensuring the correctness of the implemented features involves incorporating both code generation and lowering unit tests for each new feature.

Unit Testing

Validation

Ensuring the correctness of the implemented features involves incorporating both code generation and lowering unit tests for each new feature.

```
/* 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: 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(...)
// CHECK: call i32 @__kmpc_omp_taskwait(...)
// CHECK: ret void
```

Listing: Taskwait Lowering test

The Experiment

Comparison: ClangIR vs Clang vs GCC

How does the new ClangIR compilation flow compare against the original Clang flow (Directly to LLVM-IR) and GCC?

The Experiment

Comparison: ClangIR vs Clang vs GCC

How does the new ClangIR compilation flow compare against the original Clang flow (Directly to LLVM-IR) and GCC?

Metrics

1. Compilation times
2. Execution times

The Experiment

Comparison: ClangIR vs Clang vs GCC

How does the new ClangIR compilation flow compare against the original Clang flow (Directly to LLVM-IR) and GCC?

Metrics

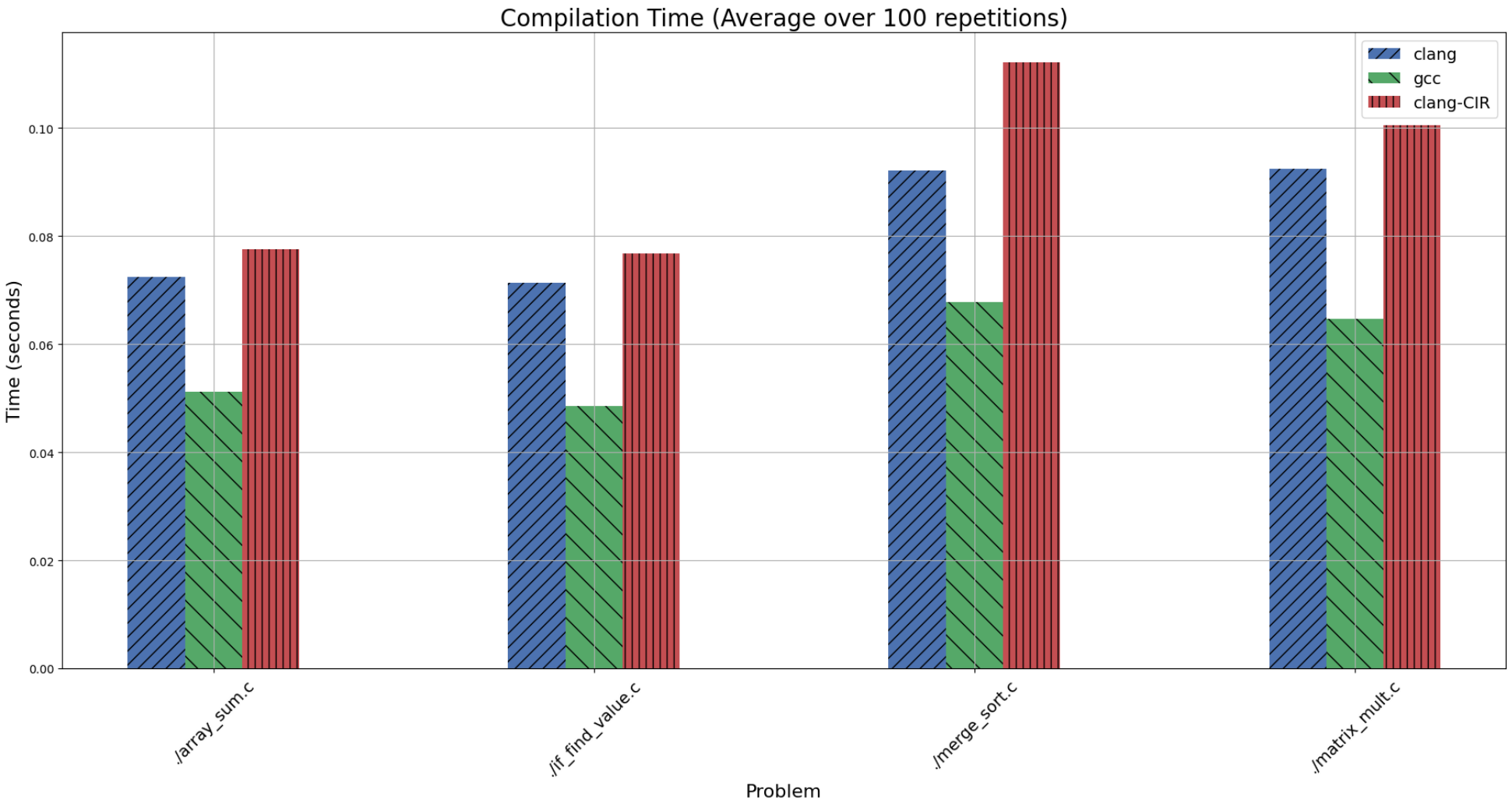
1. Compilation times
2. Execution times

Conditions

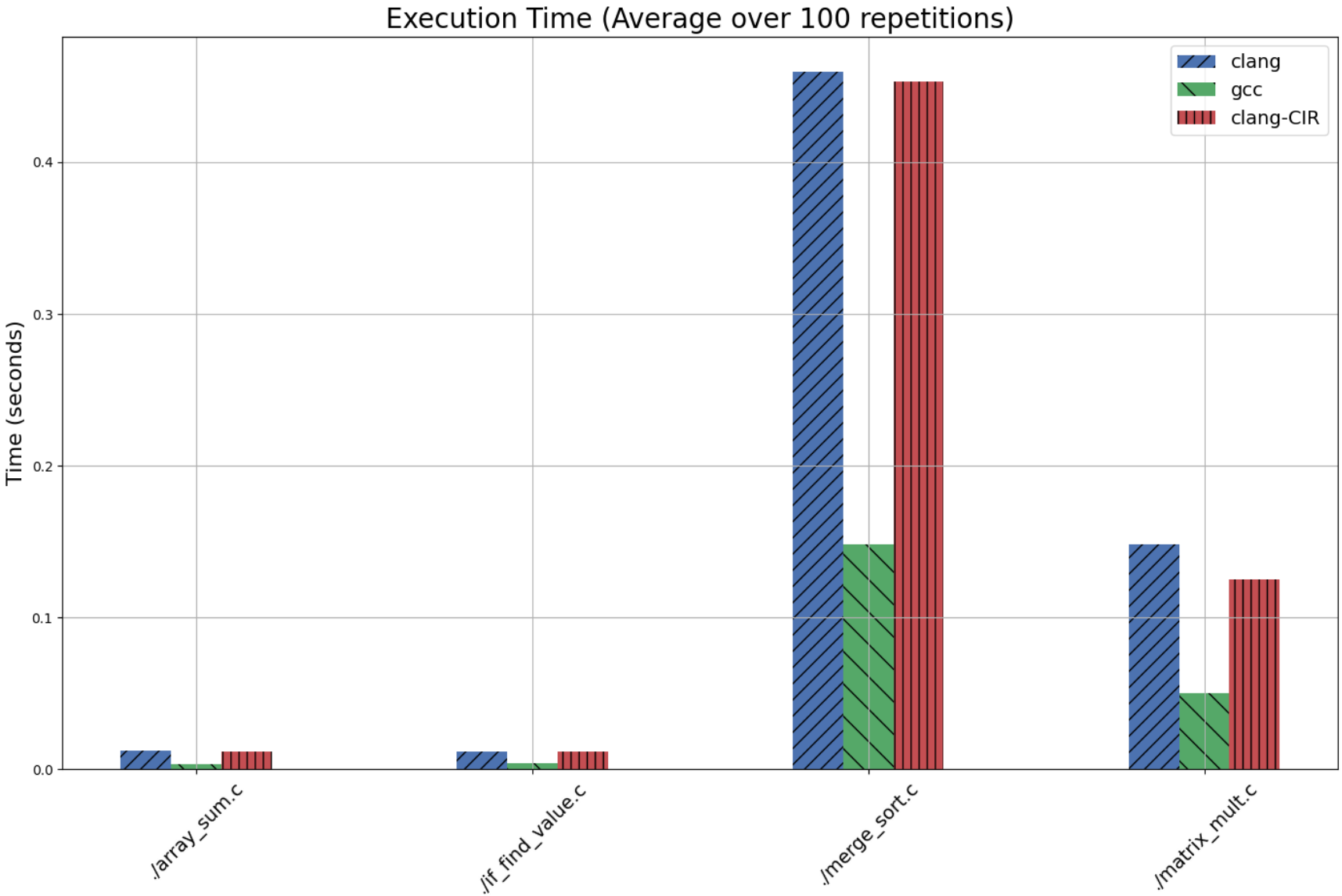
The experiment will be run on the same device, averaging results over 100 iterations using four different parallelized algorithms:

- Linear find value in array
- Array sum
- Matrix multiplication
- Merge sort

The results: compilation times



The results: execution times



Compiler Development

Challenges

- Complex, large codebases
- Difficult debugging/testing
- Steep learning curve

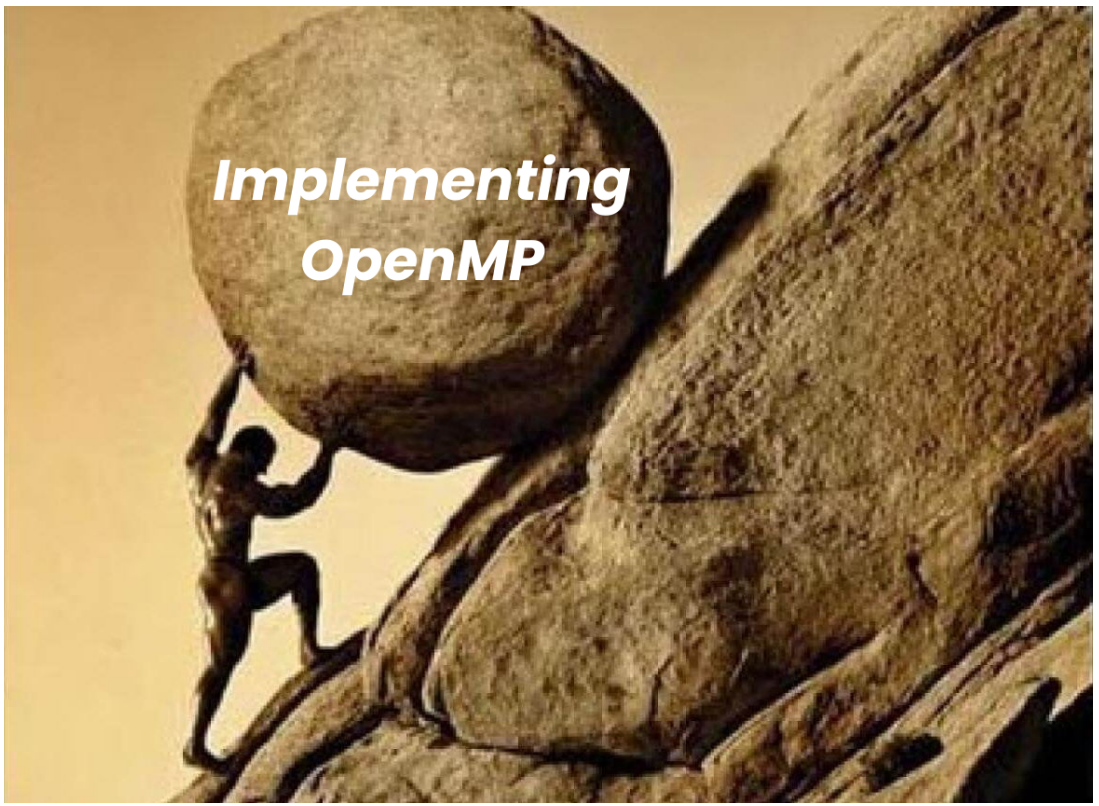
Benefits

- Faster execution times
- Potential improvements in compilation times
- Fewer bugs and security issues
- Reduced engineering efforts and redundancy

LLVM project

Standing on the shoulders of giants

Leveraging the robust MLIR and LLVM infrastructure, even a newcomer can implement 8 directives and 9 clauses in a new compilation flow. Without this foundation, compiler development would be a Sisyphean task.



Achievements

Hall of fame

1. Developed the OpenMP infrastructure of ClangIR from the ground up without prior compiler development experience.

Achievements

Hall of fame

1. Developed the OpenMP infrastructure of ClangIR from the ground up without prior compiler development experience.
2. Paved the way for further contributions

Achievements

Hall of fame

1. Developed the OpenMP infrastructure of ClangIR from the ground up without prior compiler development experience.
2. Paved the way for further contributions
3. Identifying unexplored paths in ClangIR and highlighting pain points in MLIR for future enhancement.

Achievements

Hall of fame

1. Developed the OpenMP infrastructure of ClangIR from the ground up without prior compiler development experience.
2. Paved the way for further contributions
3. Identifying unexplored paths in ClangIR and highlighting pain points in MLIR for future enhancement.
4. Authored a research document introducing modern compilers.

Achievements

Hall of fame

1. Developed the OpenMP infrastructure of ClangIR from the ground up without prior compiler development experience.
2. Paved the way for further contributions
3. Identifying unexplored paths in ClangIR and highlighting pain points in MLIR for future enhancement.
4. Authored a research document introducing modern compilers.
5. Contributed to a major open-source project (LLVM) used globally in the programming landscape.

Achievements

Hall of fame

1. Developed the OpenMP infrastructure of ClangIR from the ground up without prior compiler development experience.
2. Paved the way for further contributions
3. Identifying unexplored paths in ClangIR and highlighting pain points in MLIR for future enhancement.
4. Authored a research document introducing modern compilers.
5. Contributed to a major open-source project (LLVM) used globally in the programming landscape.
6. Collaborated with PhD candidates and leaders from top tech companies (Meta, ARM, AMD...).

Achievements

Hall of fame

1. Developed the OpenMP infrastructure of ClangIR from the ground up without prior compiler development experience.
2. Paved the way for further contributions
3. Identifying unexplored paths in ClangIR and highlighting pain points in MLIR for future enhancement.
4. Authored a research document introducing modern compilers.
5. Contributed to a major open-source project (LLVM) used globally in the programming landscape.
6. Collaborated with PhD candidates and leaders from top tech companies (Meta, ARM, AMD...).
7. Sparked interest from HPC organizations like BSC and Aachen.

Future improvements: Depend clause

Current state

Code generation is complete. However, there is ongoing discussion among ClangIR project leaders on the implementation of ClangIR pointer interfaces.

Future improvements: private data-sharings

Current state

MLIR interface does not yet support private data sharing. Handling this manually is way too complex for the purpose of this thesis.

Future improvements: Taskloop

Current state

Taskloop was deprioritized due to its high opportunity cost.

Future work

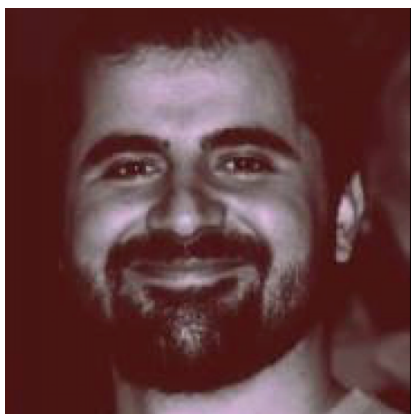
Many directives still need to be implemented: `atomic`, `section`, `sections`, `taskloop`, `simd`, `cancel`, `flush`, `teams`, `target`...

Justification

Challenges

- Community overhead: alignment, discussions, RFCs...
- High opportunity cost
- Immaturity of the MLIR OMP Dialect

LLVM Community



(a) B. Cardoso, Meta

(b) F. Mora, AMD

(c) K. Chandramohan, ARM

Special Thanks

- José Miguel Rivero, for sharing his knowledge and inspiring my interest in compilers.
- Roger Ferrer Ibáñez for his experience, passion and wisdom.
- You, the teachers who have contributed significantly to my intellectual and professional growth.
- The wonderful and collaborative open-source LLVM community, with special thanks to Bruno Cardoso and Fabian Mora.
- Patricia, my supportive and loving partner.
- To the amazing friends I've made during my college journey.

Enhancing The Clang Compiler with OpenMP Parallelism: A Novel Approach using ClangIR

Walter J. Troiani Vargas
Director: José Miguel Rivero Almeida
Co-Supervisor: Roger Ferrer Ibáñez

Department of Computer Science (CS)
Barcelona Supercomputing Center (BSC)

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Insights

Insights

1. GCC consistently shows the fastest compile times.
2. ClangIR's compile times are slightly higher than Clang's, but they remain comparable.
3. Surprisingly, GCC outperforms both Clang and ClangIR in execution times.
4. ClangIR's execution times are nearly identical to those of Clang, indicating similar runtime performance.

Insights

Insights

1. GCC consistently shows the fastest compile times.
2. ClangIR's compile times are slightly higher than Clang's, but they remain comparable.
3. Surprisingly, GCC outperforms both Clang and ClangIR in execution times.
4. ClangIR's execution times are nearly identical to those of Clang, indicating similar runtime performance.

Explanation

1. Clang is significantly larger than GCC.
2. ClangIR's compilation flow extends Clang-LLVMIR (Strict subset).
3. Performance differences stem from variations in OpenMP runtimes (GCC vs. Clang).
4. ClangIR is in its early stages and lacks additional optimizations.

Clang vs. GCC

Choosing the Right Compiler

When choosing between Clang and GCC, consider the following key trade-offs and factors and be aware of your use-case. compilation and execution times are inconclusive and keep evolving over time.

Clang

- Apache 2.0 License
- Highly modular, suitable for custom and industrial compilers
- Less memory usage

GCC

- GPLv3 License
- Smaller disk space usage

Clang vs. GCC

Choosing the Right Compiler

When choosing between Clang and GCC, consider the following key trade-offs and factors and be aware of your use-case. compilation and execution times are inconclusive and keep evolving over time.

Clang

- Apache 2.0 License
- Highly modular, suitable for custom and industrial compilers
- Less memory usage

GCC

- GPLv3 License
- Smaller disk space usage

Healthy Competition

Competition drives improvement. Both Clang and GCC benefit from the rivalry, fostering innovation and preventing stagnation.

The importance of open source

Prime examples

Chrome, Firefox, Git, Vim, VS Code, Docker, Kubernetes, MongoDB, MySQL, GNU, LLVM, Apache, Android, Linux, Ubuntu, Ruby, Python, Rust, Go, JavaScript...

But why?

- **Fosters collaboration** among tech companies, students, and individual contributors.
- Allows everyone to understand and modify large projects, **democratizing knowledge**.
- **Promotes transparency**, preventing consumers from being kept in the dark about critical technologies (e.g., software for autonomous driving).
- **Sets a social expectation** for openness, making proprietary projects less favorable by default.

The Role of FIB-UPC

Positives

Grateful for the knowledge imparted by all the teachers, enabling the completion of this thesis by teaching essential abstraction layers and tools.

Issues in the FIB-UPC Curriculum

- A single course on compilers is insufficient to cover advanced topics like optimizations, back-end development, parallelism, and modern compiler concepts.

The Role of FIB-UPC

Issues in the FIB-UPC Curriculum (cont.)

- Compiler courses are inaccessible to non-computing specialization students due to unnecessary prerequisites.
- Surprisingly, the compilers course is not mandatory for computing specialization students.