КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

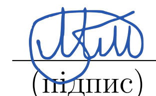
Кваліфікаційна робота

на здобуття ступеня бакалавра

За спеціальністю
121 Інженерія програмного забезпечення: програмна інженерія
на тему:

ОПТИМІЗАЦІЯ LLVM IR

Виконав студент 4-го курсу
Остап МИКИТЮК


(підпис)

Науковий керівник:
асистент, кандидат фізико-математичних наук
Костянтин ЖЕРЕБ

____
(підпис)

Засвідчую, що в цій роботі
немає запозичень з праць інших авторів
без відповідних посилань

Студент
(підпис)

Роботу розглянуто й допущено до захисту
на засіданні кафедри інтелекутальних
програмних систем

25 травня 2022 р.

Протокол № 10

Завідувач кафедри
Олександр ПРОВОТАР ________
(підпис)

Київ — 2022

РЕФЕРАТ

Обсяг роботи 21 сторінка, 1 ілюстрація, 10 джерел посилань.

LLVM IR, ОПТИМІЗАЦІЯ, БАЙТ-КОД, ПРОМІЖНЕ ПРЕДСТАВЛЕННЯ, ПРОХІД, ФРОНТЕНД КОМПІЛЯТОРА, C++, RUST

Об'єктом роботи є оптимізація неоптимального машинного коду для частовживаних конструкцій в різних мовах програмування та написання проходу (pass) по проміжному представленню, який оптимізує знайдені ділянки.

Метою роботи є пошук неоптимального машинного коду, який генерується інфраструктурою `LLVM` для мов програмування, які її використовують та подальша оптимізація.

Інструменти розробки: текстовий редактор `VSCode`, мова програмування `C++`, фреймворк для тестування `Google Benchmark` та набір бібліотек і програм `LLVM`.

Результати роботи: створено прохід по IR, який успішно шукає потрібний патерн коду та перетворює на ефективніший байт-код. Покращення швидкості виконання складає близько 3% для архітектури `x86`.

Розроблена оптимізація безпосередньо впливає на швидкодію машинного коду, який генерується компіляторами сучасних мов програмування, яі використовують `LLVM`: Rust, Swift, Clang, Zig, etc.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ 4

ВСТУП 5

РОЗДІЛ 1. ЩО ТАКЕ LLVM ТА IR? 7
1.1 Детальніше про IR 8

РОЗДІЛ 2. ПОШУК ТА АНАЛІЗ ОПТИМІЗАЦІЇ 10

РОЗДІЛ 3. ІМПЛЕМЕНТАЦІЯ ПРОХОДУ 13

РОЗДІЛ 4. БЕНЧМАРК 17

ВИСНОВКИ 19

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 20

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

CFG — Control-flow Graph, множина всіх можливих шляхів виконання програми (найчастіше окремої функції), представлених у вигляді графа;

Сніпет — snippet, невеликий фрагмент коду;

AST — Abstract Syntax Tree, представлення абстрактної структури вихідного коду у вигляді дерева;

IR — Intermediate Representation, проміжне представлення програми перед фазою генерації коду;

Pass — прохід, цикл проходу оптимізатора по коду та його оптимізація;

ВСТУП

Оцінка сучасного стану об'єкта розробки. Останнім часом спостерігається тенденція, як показує, що закон Мура [1] перестає працювати [2], тобто зменшується темп збільшення вичислювальних ресурсів. В умовах глобальної діджиталізації потреба у масштабуванні обчислювальних ресурсів збільшується неймовірними темпами.

Оптимізацій машинного коду дуже не вистачає (особливо використання новомодних SIMD/AVX), оскільки існує ще велика кількість конструкцій, оптимізація яких може значно підвищити швидкодію програми та розмір результуючого бінарного файлу.

Актуальність роботи. Зараз відбувається тотальна цифровізація всього, стрімко зростає потреба у вичислювальних ресурсах. Часто недостатньо тільки горизонтального масштабування, потрібно оптимізовувати роботу з наявним процесором та його архітектурою; оптимізовувати машинний код. Дуже важливо вміти шукати вузькі місця та покращувати їхню швидкодію.

Від інженерів-програмістів останнім часом все частіше вимагають вміння працювати з профайлерами пам'яті та швидкодії, оскільки дуже часто потрібно вміти знайти вузькі місця у вже існуючих програмаха та рішеннях, щоб скоротити витрати на залізо. Технологічні гіганти на кшталт Google або Facebook мають багато власних датацентрів. Виграш по перфомансу в декілька відсотків на таких масштабах економлять таким компаніям мільйони доларів, а також, що не менш важливо, зменшує нагрівання атмосфери [3].

Окрім класичних мов програмування, на радарах з'явились технології децентралізації, блокчейн та смарт-контракти. Для написання останніх часто створюються свої доменно-специфічні мови та активно використовується `LLVM` [4]. Враховуючи кількість транзакцій, дуже важливо вміти оптимізовувати швидкодію, що веде до збільшення пропускної здатності.

Метою роботи є пошук неоптимального машинного коду, який генерує-

ться інфраструктурою LLVM для мов програмування, які її використовують, та подальша оптимізація.

Інструментами розробки є текстовий редактор VS Code через свою простоту та гнучкість, libLLVM як основа для написання проходу та мова програмування C++, оскільки це мова проєкту LLVM.

РОЗДІЛ 1. ЩО ТАКЕ LLVM ТА IR?

`LLVM` — це набір бібліотек та програм для розробки компіляторів. Абревіатура розшифровується як Low Level Virtual Machine, тобто низкорівнева віртуальна машина. Використовуючи цю інфраструктуру можна написати фронтенд для мови програмування та правила перетворення її AST в проміжне представлення — IR. Принцип роботи показаний на рисунку 1.1.

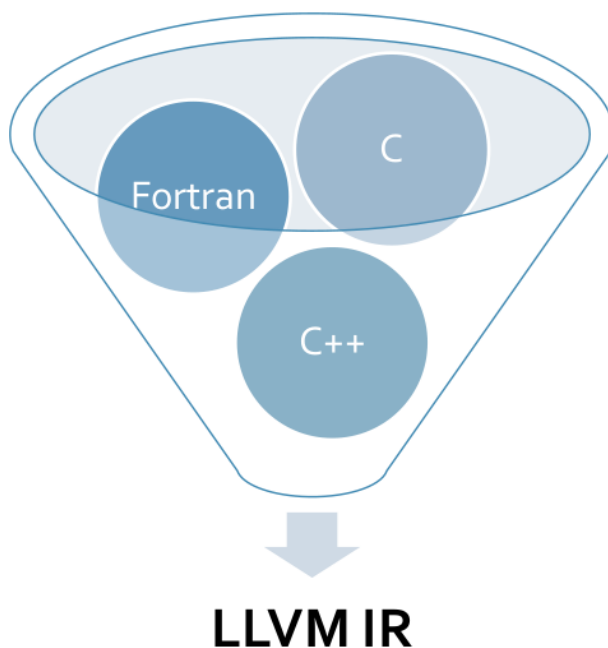


Рисунок 1.1 Перетворення AST в IR

Решту LLVM бере на себе, а саме перетворення проміжного представлення в машинний код для конкретної платформи. Очевидно, що це набагато кращий підхід, ніж написання трансляторів AST в машинний код кожною мовою програмування окремо, оскільки архітектур процесорів та наборів машинних команд є дуже велика кількість.

Найвідоміші мови програмування та їхні компілятори, які використовують LLVM:

- C/C++ (clang);
- Rust (rustc);

— Swift (swiftc);

— Zig (zig);

`LLVM` написаний на `C++` та має власний компілятор `Clang`, який використовує `LLVM` за основу. Також `LLVM` має власний лінкер `lld` та дебагер `lldb`. До всього перечисленого є багато інших утиліт, які роблять роботу з `LLVM` дуже комфортною.

Один з найбільших мінусів `LLVM` — це скупа документація, але є декілька авторських книжок, які детально пояснюють роботу з ним. Одна з таких книжок, LLVM Techniques, Tips and Best Practices [5], була використана для написання цієї роботи. Ця книжка покриває останню мажорну версію `LLVM 14`.

1.1 Детальніше про IR

IR — це зручний байт-код, який може читати людина. Він має купу переваг у порівнянні з іншими проміжними представленнями (наприклад, GIMPLE [6] компілятора gcc):

— RISC-подібний набір віртуальних інструкцій, які підходять для всіх мов та машин. Ці віртуальні інструкції потім трансулюються в машинні інструкції для конкретної платформи;

— представлення коду у вигляді SSA [7];

— сильна типізація. У кожної зміної є фіксований тип, кожна функція повертає значення конкретного типу;

— модульність;

Найкраще показати, як працює LLVM IR, на невеликому прикладі — функції додавання двох чисел на мові програмування `C++`.

```
int add(int x, int y) {
    return x + y;
}
```

Сніпет 1.1 Проста функція додавання двох чисел

Враховуючи вищеописані властивості IR, розуміння IR стає досить інтуїтивним. Нижче в сніпеті 1.2 показане проміжне представлення для простенької функції додавання двох чисел із сніпету 1.1. В сніпеті

```
define i32 @add(i32 %0, i32 %1) {
  %3 = add nsw i32 %1, %0
  ret i32 %3
}
```

Сніпет 1.2 Додавання двох цілих чисел в LLVM IR

В першому рядку відбувається оголошення глобальної функції `add`, яка приймає два 32-бітних цілих знакових числа та вертає значення такого ж типу. В другому рядку створюємо нову змінну та присвоюємо їй результат додавання двох чисел, результуюча змінна має тип `i32`. В команди `add` є неочевидний параметр `nsw`, але насправді все стає набагато зрозумілішим, якщо заглянути в довідку по мові IR [8]. І, нарешті, останній рядок — вернути значення змінної, в якій записаний результат додавання двох чисел.

Для повноти картини нижче наведений асемблерний "вихлоп"для архітектури `x86`.

```
add(int, int):
        lea     eax, [rdi + rsi]
        ret
```

Сніпет 1.3 Функція додавання двох цілих чисел на `x86`

РОЗДІЛ 2. ПОШУК ТА АНАЛІЗ ОПТИМІЗАЦІЇ

Зазвичай, коли хочуть написати оптимізацію, інженери мають реальний кейс і його проблемні місця.

Змоделюємо ситуацію: на адронному колайдері пучки протонів розганяють до швидкості світла і починають ці пучки зіштовхувати один з одним. За долю секунди з'являються сотні мільярдів нових частинок, які потрібно якось аналізувати та обробляти. Цілком ймовірно може виникнути питання, чи у двох подібних частинок однаковий заряд. В коді це виражається функцією `samesign_naive` (мова `C++`).

```
bool samesign_naive(int x, int y) {
    return (x >= 0 && y >= 0) || (x <= 0 && y <= 0);
}
```

Сніпет 2.1 Наївна перевірка на однаковий знак

Коли частинок величезна кількість потрібно оптимізовувати фільтри й перевіряти, наскільки ефективно використовується процесорний час та пам'ять.

Найсвіжіша версія `clang`'а (15.0.0) видає наступний код для платформи `x86`.

```
samesign_naive(int, int):
        mov     al, 1
        mov     ecx, esi
        or      ecx, edi
        js      .LBB1_1
        ret
.LBB1_1:
        test    edi, edi
        setle   cl
        test    esi, esi
        setle   al
        and     al, cl
        ret
```

Сніпет 2.2 Асемблерне представлення перевірки на однаковий знакоптим

Тут є умовні переходи і, в цілому, багато зайвого коду. По великому рахунку нам потрібно перевірити, чи знакові біти двох чисел однакові. Це можна

зробити в рази простішим способом — зробити побітове "виключне або"(xor) двох чисел та перевірити, чи результат не буде від'ємним.

Результат буде невідємним тільки тоді, коли знакові біти обох чисел однакові, оскільки $1 \oplus 1 = 0$ та $0 \oplus 0 = 0$, в решті випадків знаковий біт буде виставлений в 1.

Вигляд описаної концепції в проміжному представленні LLVM можна побачити в сніпеті 2.3 та його трансляцію в асемблерний код в сніпеті 2.4.

```
define i1 @samesign(i32 %0, i32 %1) {
  %3 = xor i32 %1, %0
  %4 = icmp sgt i32 %3, -1
  ret i1 %4
}
```

Сніпет 2.3 Оптимізована перевірка в проміжному представленні

```
samesign(int, int):
        xor     edi, esi
        setns   al
        ret
```

Сніпет 2.4 Оптимізована перевірка для x86

Проміжне представлення наївного способу визначення того, чи два цілих числа мають однаковий знак, наведено в сніпеті 2.6. Його можна отримати завдяки команді зі сніпету 2.5.

```
$ clang -O3 -emit-llvm samesign.cpp
```

Сніпет 2.5 Команда отримання проміжного представлення

```
define i1 @samesign_naive(i32 %0, i32 %1) {
  %3 = or i32 %1, %0
  %4 = icmp sgt i32 %3, -1
  br i1 %4, label %9, label %5

5: ; preds = %2
  %6 = icmp slt i32 %0, 1
  %7 = icmp slt i32 %1, 1
  %8 = and i1 %6, %7
  br label %9

9: ; preds = %2, %5
  %10 = phi i1 [ %8, %5 ], [ true, %2 ]
  ret i1 %10
}
```

Сніпет 2.6 Проміжне представлення samesign

На цьому прикладі можна побачити головну перевагу SSA-підходу: кожній змінній значення може бути присвоїне тільки один раз. Це значно спрощує написання проходів для оптимізації.

В цілому, код є досить інтуїтивним, складається з трьох базових блоків (basic block) [9], семантично помітно, що це наївна left-to-right перевірка умов.

РОЗДІЛ 3. ІМПЛЕМЕНТАЦІЯ ПРОХОДУ

Ця оптимізація є незвичною, оскільки вона опирається на цілих три базових блоки з умовними переходами. Подібного роду оптимізації називаються агресивними, оскільки вони комбінують складні конструкції у більш прості. В LLVM вже є реалізація декількох дуже цікавих агресивних оптимізацій, їх можна включити ключем компілятора `aggressive-instcombine`.

Новий прохід по IR потрібно реєструвати в класі `PassManager`, надаючи всі необхідні метадані. Оскільки ми оптимізовуємо функцію, то і наслідуємось ми від `FunctionPass` та перевантажувати функцію, яка викликатиметься для кожної функції в програмі.

```
class AggressiveInstCombinerPass : public FunctionPass {
public:
  static char ID; // Pass identification, replacement for typeid

  AggressiveInstCombinerLegacyPass() : FunctionPass(ID) {
    initializeAggressiveInstCombinerLegacyPassPass(
        *PassRegistry::getPassRegistry());
  }

  void getAnalysisUsage(AnalysisUsage &AU) const override;

  /// Run all expression pattern optimizations
  /// on the given /p F function.
  ///
  /// \param F function to optimize.
  /// \returns true if the IR is changed.
  bool runOnFunction(Function &F) override;
};
```

Сніпет 3.1 Декларація класу проходу

Тепер в імплементації функції потрібно зробити семантичний матч функції наївної перевірки та замінити на базовий блок без умовних переходів та зі xor'ом. Фактично, реалізація сходиться до наступних перевірок:

— функція приймає два аргументи однакового знакового цілочисельного типу;

— функція має три базових блоки: в першому перевіряється умова `icmp sl 1`, а у другому icmp sg -1, третій блок — це повернення одного із значень з перших двох блоків;

— перевірити, що функція повертає значення типу `i1`, тобто булеан;

Це все робиться за допомогою допоміжних функцій, які метчать не конкретні інструкції, а семантичну поведінку. Приклад матчінгу інструкції додавання двох чисел наведено в сніпеті 3.2.

```
Value *AddOp;
if (match(AddOp, m_Add(I.getOperand(0), I.getOperand(1))) {
    // Process...
}
```

Сніпет 3.2

Для `C++` реалізована оптимізація працює без проблем, оскільки ми на її основі робили припущення. Але для верифікації того, що оптимізація працює не тільки для однієї мови програмування напишемо код на Rust (сніпет 3.3), глянемо його поточне проміжне представлення (сніпет 3.4) та порівняємо з проміжним представленням після застосування оптимізації.

```
pub fn samesign(x: i32, y: i32) -> bool {
    x >= 0 && y >= 0 || x <= 0 && y <= 0
}
```

Сніпет 3.3 Перевірка на однаковий знак двох чисел на мові Rust

```
define i1 @samesign(i32 %x, i32 %y) {
start:
  %_4 = icmp sgt i32 %x, -1
  %_6 = icmp sgt i32 %y, -1
  %_3.0 = select i1 %_4, i1 %_6, i1 false
  br i1 %_3.0, label %bb3, label %bb2

bb2: ; preds = %start
  %_9 = icmp slt i32 %x, 1
  %_11 = icmp slt i32 %y, 1
  %_8.0 = select i1 %_9, i1 %_11, i1 false
  br label %bb3, !dbg !10
```

```
bb3: ; preds = %start, %bb2
  %.0 = phi i1 [ %_8.0, %bb2 ], [ true, %start ]
  ret i1 %.0
}
```

Сніпет 3.4 Проміжне представлення від фронтенду Rust'а для наївної перевірки

Проміжне представлення дещо відрізняється від еквівалентної функції для C++, але семантично воно робить те саме. Також, тут більш явно показані базові блоки, а аргументи функції зберигли свої імена, замість порядкових номерів. Це вже відноситься до особливостей того, як фронтенд Rust'а транслює синтаксис в проміжне представлення.

Наступною командою можна підвантажити компілятору Rust'а динамічну бібліотеку з імплементованою оптимізацією.

```
$ rustc --emit=llvm-ir \
        -C opt-level=3 \
        -C load=lib/LLVMAggressiveInstCombine.so \
        samesign.rs
```

Сніпет 3.5 Команда підвантажування динамічної бібліотеки компілятором Rust'а з подальшим виходом IR

Можна побачити, що оптимізований IR Rust-функції в сніпеті 3.6 еквівалентний до раніше наведеного сніпета 2.6.

```
define i1 @samesign(i32 %x, i32 %y) {
start:
  %_3 = xor i32 %y, %x
  %0 = icmp sgt i32 %_3, -1
  ret i1 %0
}
```

Сніпет 3.6 Оптимізований IR для функції, написаної мовою програмування Rust

Проходів по IR є дуже багато, потрібно вміти відслідковувати час виконання проходу, оскільки на великих кодобазах компіляція може зайняти багато часо. В LLVM це все передбачено, тому можна передати прапорець `-report` компілятору та отримати детальний репорт виконання.

Розроблена оптимізація у великих кодобазах рідко зустрічається, тому на тестових даних така перевірка не буде репрезентативною, але корисно знати про таку можливість.

Також часто грає роль послідовність, в яких проходи застосовуються, оскільки одні оптимізації можуть покладатись на результат інших. В розробленій оптимізації використовується семантичний аналіз коду, тому не важливо, перед якими або після яких була застосована дана оптимізація.

РОЗДІЛ 4. БЕНЧМАРК

Залишається зробити останній пункт — бенчмарк. Потрібно порівняти код оригінальної функції та оптимізованої на предмет швидкодії. В сніпеті 4.1 показано код, який був використаний для тестування. Для замірів була використана бібліотека Google Benchmark [10], яка дозволяє зручно тестувати код на мові C++.

```
#include <benchmark/benchmark.h>
#include <random>

int rand32() {
    static std::mt19937 rng(std::random_device{}());
    static std::uniform_int_distribution<> dist{-10'000, 10'000};
    return dist(rng);
}

void BM_samesign(benchmark::State& state) {
    for (auto _ : state) {
        state.PauseTiming();
        const auto x = rand32();
        const auto y = rand32();
        state.ResumeTiming();

        benchmark::DoNotOptimize(samesign(x, y));
    }
}

BENCHMARK(BM_samesign);
BENCHMARK_MAIN();
```

Сніпет 4.1 Код бенчмарку

Генеруємо два цілих числа заміряємо швидкість функції `samesign` зі сніпета 2.1. Використовуємо спеціальну функцію `benchmark::DoNotOptimize`, щоб компілятор не оптимізував виклик функції як мертвий код, тому явно вказуємо, що значення, яке повертає функція, використовується.

Спочатку запускаємо оригінальним компілятором, а потім модифікований з імплементованою оптимізацією.

Швидкість може відрізнятись на різних прцоесорах з різними архітектурами. Цей бенчмарк запускався лише на процесорі `Intel Core i7-8550U` (ар-

хітектура `x86`) та `16 Gb` оперативної памяті.

Загальний приріст швидкості, у порівнянні з неоптимізованою версією, на випадково згенерованих даних склав 3%, що є чудовим результатом для такої оптимізації [11].

ВИСНОВКИ

Результатом роботи стала реалізація проходу, який знаходить патерн перевірки двох цілих знакових чисел на одинаковий знак наївним методом та перетворює відповідний `IR` на оптимальніший.

Реалізований бенчмарк, який показує, що оптимізація дає 3% виграшу в швидкості там зменшує розмір бінарного файлу.

Описані перспективи оптимізації байт-коду та процес реалізації на прикладі `LLVM IR`.

Детально продемостроване перевага інфраструктури `LLVM` для написання фронтенду своєї мови програмування. Показано, як написання оптимізації для `IR` дає виграш всім мовам, які використовують `LLVM`.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

[1] *What is Moore's Law?* [Electronic Resource] / Mike Gianfagna // Synopsys [web-site]. – Access Mode: https://www.synopsys.com/glossary/what-is-moores-law.html

[2] *Moore's Law Is Dead. Now What?* [Electronic Resource] / Przemek Chojecki // Built In [web-site]. – Access Mode: https://builtin.com/hardware/moores-law

[3] *Data Centers & The Environment* [Electronic Resource] // Super Micro [web-site]. – Access Mode: https://www.supermicro.com/wekeepitgreen/Data-Centers-and-the-Environment-Dec2018-Final.pdf

[4] *LLVM x Blockchain* [Electronic Resource] / Robin Zhong // LLVM Developers Meeting 2018 [web-site]. – Access Mode: https://llvm.org/devmtg/2018-04/slides/Zhong-LLVM%20x%20Blockchain.pdf

[5] *LLVM Techniques, Tips and Best Practices* [Electronic Resource] / Min-Yih Hsu // Packt Publishing [web-site]. – Access Mode: https://www.packtpub.com/product/llvm-techniques-tips-and-best-practices-clang-and-middle-end-libraries/9781838824952

[6] *Exploring the Intermediate Representation of GCC* [Electronic Resource] / Min-Yih Hsu // Blog @ AESTE [web-site]. – Access Mode: https://blog.aeste.my/2015/07/27/exploring-the-intermediate-representation-of-gcc

[7] *LLVM Memory and SSA* [Electronic Resource] // Opengenus [web-site]. – Access Mode: https://iq.opengenus.org/llvm-memory/

[8] *LLVM Language Reference Manual* [Electronic Resource] // LLVM [web-site]. – Access Mode: https://llvm.org/docs/LangRef.html

[9] *LLVM: Bacic Block* [Electronic Resource] // LLVM [web-site]. – Access Mode: https://llvm.org/doxygen/group__LLVMCCoreValueBasicBlock.html

[10] *Google Benchmark Repository* [Electronic Resource] // GitHub [web-site]. – Access Mode: https://github.com/google/benchmark

[11] *Reconciling High-Level Optimizations and Low-Level Code in LLVM* [Electronic Resource] // Utah University [web-site]. – Access Mode: https://www.cs.utah.edu/ regehr/oopsla18.pdf