浅谈 IR 优化

2021 年编译赛 IR 优化经验总结

何纪宏　李亚美　许文胜　指导教师：王锋

湖南大学
信息科学与工程学院

2022.7

浅谈 IR 优化

1. IR 设计
   - 基本块参数
   - Sea of Nodes
   - LLVM IR
   - SSA IR 的生成
2. IR 优化及相关技巧
   - 常用优化
   - 一个具体优化的例子
   - 优化技巧
3. 赛后回顾

IR 设计

All we need is...
SSA

1. **基本块参数**（Basic Block Arguments
2. Sea of Nodes IR
3. ...

基本块参数——源程序

```
float
average(const float *array, size_t count)
{
    double sum = 0;
    for (size_t i = 0; i < count; i++)
        sum += array[i];
    return sum / count;
}
```

基本块参数——传统 SSA 形式

```
function %average(i32, i32) -> f32 {
    ss0 = alloca f64 ; allocate a stack slot for sum

block1:
    v2 = f64const 0x0.0
    store v2, ss0
    brz v1, block5 ; Handle count == 0.
    jump block2

block2:
    v3 = iconst.i32 0
    jump block3

block3:
    v4 = phi([v3, block2], [v11, block3])
    ; ...omitted multiplication add accumulation to sum...
    v11 = iadd_imm v4, 1
    v12 = icmp ult v11, v1
    br v12, block3, block4

block4:
    ; ...omitted sum / count...
    v16 = "sum / count"
    return v16

block5:
    return +NaN
}
```

基本块参数——基本块参数形式

```
function %average(i32, i32) -> f32 {
    ss0 = alloca f64 ; allocate a stack slot for sum

block1(v0: i32, v1: i32):
    v2 = f64const 0x0.0
    store v2, ss0
    brz v1, block5 ; Handle count == 0.
    jump block2

block2:
    v3 = iconst.i32 0
    jump block3(v3)

block3(v4: i32):
    ; ...omitted multiplication add accumulation to sum...
    v11 = iadd_imm v4, 1
    v12 = icmp ult v11, v1
    br v12, block3(v11), block4

block4:
    ; ...omitted sum / count...
    v16 = "sum / count"
    return v16

block5:
    return +NaN
}
```

优点

1. 不需要特殊处理入口基本块
2. 不需要特殊处理块内 Phi 结点

Sea of Nodes

1. Cliff Click 的 PhD 论文
2. 分析与优化相结合，甚至一个 pass 就能搞定大多数优化（GVN/GCM）
3. 模糊了基本块的边界
4. 可以在 IR 上做指令调度

Today: Still LLVM IR

1. 设计简单，容易理解
2. 不管是什么形式的 SSA，其静态单赋值的核心思想是不变的
3. 参考实现较多，方便大家对照学习（Sourcegraph）
4. 可以使用 LLVM 辅助验证前端
5. LLVM 的 dot-cfg pass 可以将函数的控制流图输出到 dot 文件：opt -dot-cfg -disable-output

1. Efficiently computing static single assignment form and the control dependence graph
2. Simple and Efficient SSA Construction

IR 优化及相关技巧

死代码消除

- 能消除未被使用过的值
- 能消除被使用过，但不会对结果产生影响的值，例如

```
int the_answer_to_life_the_universe_and_everything(int x) {
    int sum = 0;
    while (x >= 0) {
        sum += 100;
        x -= 1;
    }
    return 42;
}
```

- 常量传播/折叠
- 函数内联

```
// by itself, the function is doing useful work,
// so we can't optimize out the loop.
int the_answer_to_life_the_universe_and_everything(int x) {
    int sum = 42;
    while (x >= 0) {
        sum += 100;
        x -= 1;
    }
    return sum; // NOW 'sum' is useful
}

int main() {
    // but the call site pass a -1 to the function,
    // making the loop useless.
    putint(the_answer_to_life_the_universe_and_everything(-1));
}
```

循环展开

- 能消除未被使用过的值
- 能消除被使用过，但不会对结果产生影响的值，例如

```
int a[16] = {1};
int i = 1;
while (i < 16) {
    a[i] = a[i-1] * 2;
    i += 1;
}

a[1] = 1;
a[2] = 2;
a[3] = 4;
// ...
```

- 公共子表达式消除
- 循环不变量外提
- 浮点优化

高级优化

- MemorySSA：将内存切片也视为一个值，
- ArraySSA：提供精确到数组元素的数据流信息
- 向量化：利用 ARM NEON 的 SIMD 指令，一次性处理多个数据
- 多线程：利用 Linux 提供的线程 API 来并行化程序（一般使用 clone 系统调用）

常量传播与除常数优化

```
const int base = 16;
int getNumPos(int num, int pos){
    int tmp = 1;
    int i = 0;
    while (i < pos){
        num = num / base;
        i = i + 1;
    }
    return num % base;
}
```

常量传播与除常数优化

```
const int base = 16;
int getNumPos(int num, int pos){
    int tmp = 1;
    int i = 0;
    while (i < pos){
        num = num / 16;
        i = i + 1;
    }
    return num % 16;
}
```

常量传播与除常数优化

```
sdiv num, #16
@ -----> 优化后
asr r1, num, #31
add r1, num, r1, lsr #28
bic r1, r1, #15
sub result, num, r1
```

除法变为基本算数操作，尽量避免高延迟的 sdiv。被除数不是 2 的幂也可变成乘法实现。

消除重复函数计算

```
while (f(v, b) != i){
    int t = v;
    v = A[B[f(t, b)]];
    A[B[f(t, b)]] = t;
    B[f(t, b)] = B[f(t, b)] + 1;
}
```

消除重复函数计算

```
while (f(v, b) != i){
    int t = v;
    int x = f(t, b);
    v = A[B[x]];
    A[B[x]] = t;
    B[x] = B[x] + 1;
}
```

MemorySSA ——消除重复内存读取

```
int x = A[0];
A[0] = 1;
int y = A[0];
```

内存操作的根本，就是要保证读出来的值是对的。怎么知道两次 A[0] 读出来的值一不一样呢？
引入内存版本的概念，如果两个 load 从同一个内存版本中读取，数据就是一样的：

```
int x = A[0 | 内存版本0];
A[0] = 1; // 污染（clobber）了内存版本0，产生新版本1
int y = A[0 | 内存版本1];
```

要知道两个 load 值是否相同，只需比较其基址，偏移，以及依赖的内存版本。

GVN+MemorySSA

```
while(...) {
    c[f(a[i])] = c[f(a[i])] + 1;
    ....
}
```

发现两次 a[i] 读出的值一样，利用 GVN 公共子表达式消除，合并成一次：

```
while(...) {
    int x = a[i];
    int y = f(x);
    c[y] = c[y] + 1;
    ....
}
```

循环展开 + 内存复制传播

```
int i = 1;
while (i < 16){
    a[i] = b[i - 1];
    b[i] = a[i] + c[i];
    i = i + 1;
}
```

展开后

```
int i = 1;

a[i] = b[i - 1];
b[i] = a[i] + c[i];
i = i + 1;

a[i] = b[i - 1];
b[i] = a[i] + c[i];
i = i + 1;
....
```

循环展开 + 内存复制传播

```
int i = 1;

a[i] = b[i - 1];
b[i] = a[i] + c[i];
i = i + 1;

a[i] = b[i - 1];
b[i] = a[i] + c[i];
i = i + 1;
....
```

经过一轮常量折叠，变成

```
a[1] = b[0]; // b[0]的值给a[1]
// 这里的a[1]就不用再去读内存了，直接变成a[1] = b[0] + c[1]
b[1] = a[1] + c[1];

a[2] = b[1]; // 这个b[1]同理
b[2] = a[2] + c[2];

....
```

经过总体四遍优化，程序的运行速度提高了 160%

IR 的验证遍（Verify Pass）

将一些对 IR 的验证写成 Pass

1. Phi 一定要在基本块最前面
2. 所有指令的操作数不能是空指针
3. Phi 结点的操作数不能指向一个已经被优化掉的基本块
4. 每个 Value 的 Value->bb 一定真的是它所在的块
5. Value 的 User 不能为空
6. ...

IR 的验证遍（Verify Pass）

将一些对 IR 的验证写成 Pass

1. 帮助解决卡了我们两周的 bug
2. 保证 IR 至少具有一定形式正确性
3. 可以与 CI 相结合，尽早发现 pass 内或不同 pass 之间互相影响造成的 bug

优化遍的顺序安排（Compiler Phase-Ordering）

优化遍本身重要，优化遍顺序安排也尤其重要，这里介绍一种特别简单的顺序安排策略：拓扑排序

1. 函数内联 → 常量传播/折叠 → 死代码消除
2. 循环展开 → 公共子表达式消除 → 常量传播/折叠 → 死代码消除
3. 函数内联 → 循环展开

拓扑排序之后：函数内联 → 循环展开 → 公共子表达式消除 → 常量传播/折叠 → 死代码消除

缺点：需要自己分析每个样例，列出优化序列，可以参考更高级的算法

赛后回顾

赛后回顾

1. 团队分工
2. 完善 IR 的文档
3. 功能第一，性能第二
4. 重视测试，CI

Thank you!
祝大家比赛取得好成绩