



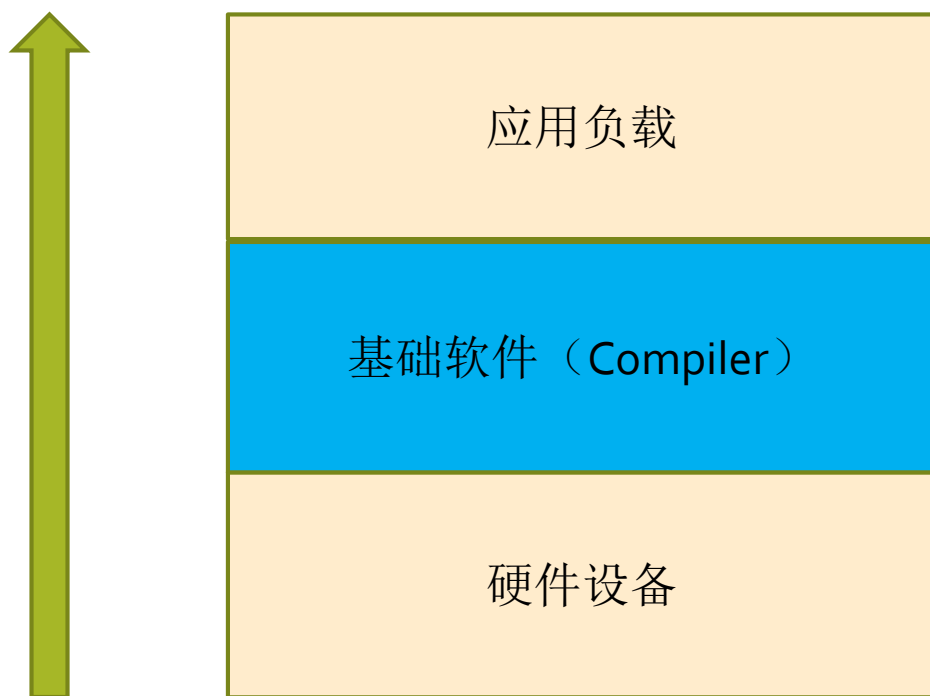
编译器的优化能力

黄波

bhuang@dase.ecnu.edu.cn

本次课的关注点

Scale up
全栈思维

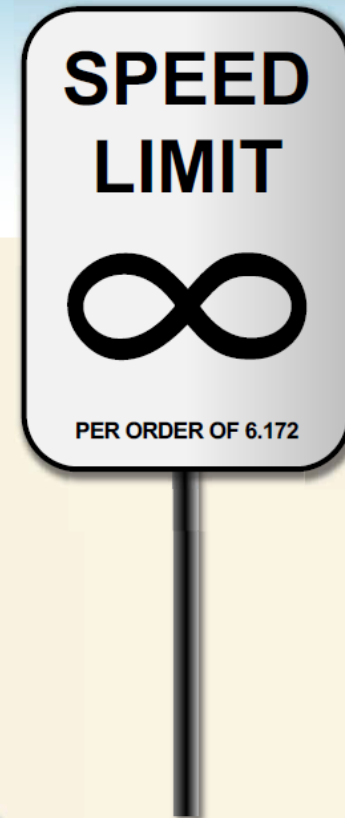


优化：没有最好，只有更好！

- 编译优化的本质是程序变换
- 编译优化的两大原则
 - ✓ 正确：必须保证语义等价
 - ✓ 保守：如果优化有风险，那就不优化
- 编译器可以做很多非常高级的优化
- 也有一些优化编译器无法完成

Q: 碰到编译器无法完成期待的优化时我们该怎么办？

6.172
Performance
Engineering
of Software
Systems

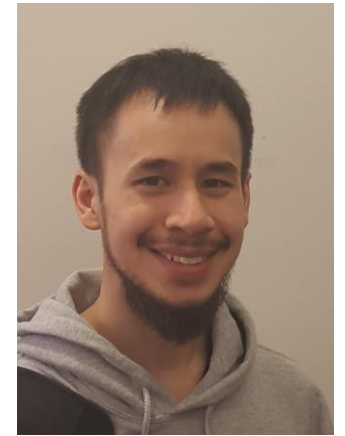


LECTURE 9
What Compilers Can and
Cannot Do

Tao B. Schardl

© 2008–2018 by the MIT 6.172 Lecturers

1

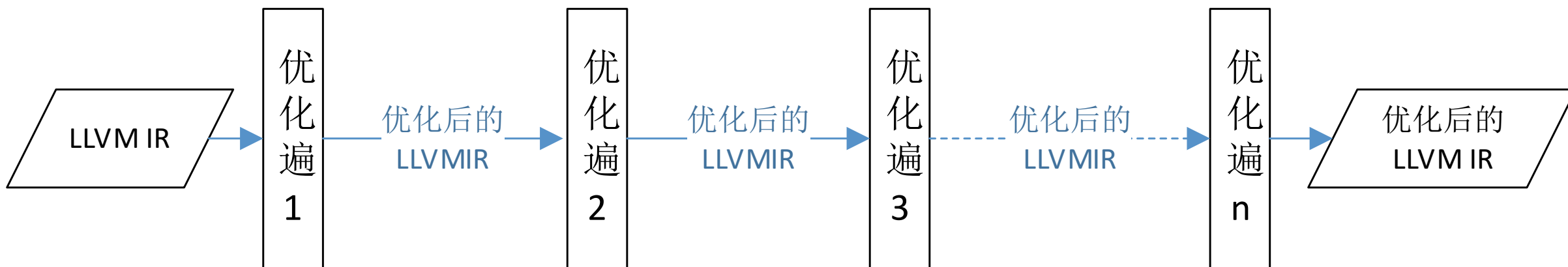


https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/lecture-slides/MIT6_172F18_lec9.pdf

为什么要了解编译器的优化机理？

- 编译器对软件的性能起了非常重要的影响
- 利用编译器来进行程序优化可以极大地提升程序性能调优的效率
- 用编译器可以在达到优化效果的同时，很好地保证程序代码的简洁性、可读性和可维护性
- 可以通过研究编译器编译过程的报告、源代码和编译器生成的中间代码（或者汇编代码）的差异，更加深入地理解编译器的优化效果，从而更加有效地用好编译器
- 编译器也可能会有bug！

LLVM 编译器的“优化遍”



- 编译优化其实由多个“优化遍”构成
- 每一个优化都是基于程序中间表示的程序分析及程序变换
- 有些“优化遍”可能在编译优化的过程中被多次执行
- “优化遍”之间的执行顺序会影响最终的优化（程序变换）效果

编译器的分析/优化报告

LLVM编译器对于内部实现的很多“优化遍”都能产生详细的优化报告，而且这些优化报告的生成可以通过编译器选项来控制：

- *-Rpass=<string>*：报告跟<string>限定的字符串合法匹配并且成功执行的“优化遍”
 - *-Rpass-missing=<string>*：报告跟<string>限定的字符串合法匹配但没有成功执行的“优化遍”
 - *-Rpass-analysis=<string>*：报告跟<string>限定的字符串合法匹配的“优化遍”所执行过的程序分析
- <string>是一个正则表达式，比如说用“.”可以匹配所有的“优化遍”

LLVM编译器的优化报告示例

```
bhuang@LAPTOP-BHUANG:~/Courses/fib$ clang -O3 fib.c -S -emit-llvm -Rpass-analysis=.* -Rpass-missed=.*
fib.c:3:9: remark: Simplify the CFG: IR instruction count changed from 20 to 19; Delta: -1 [-Rpass-analysis=
size-info]
int64_t fib(int64_t n) {
    ^
fib.c:3:9: remark: Simplify the CFG: Function: fib: IR instruction count changed from 20 to 19; Delta: -1 [-Rpass-analysis=
size-info]
fib.c:3:9: remark: SROA: IR instruction count changed from 19 to 11; Delta: -8 [-Rpass-analysis=size-info]
fib.c:3:9: remark: SROA: Function: fib: IR instruction count changed from 19 to 11; Delta: -8 [-Rpass-analysis=
size-info]
fib.c:3:9: remark: Simplify the CFG: IR instruction count changed from 11 to 10; Delta: -1 [-Rpass-analysis=
size-info]
fib.c:3:9: remark: Simplify the CFG: Function: fib: IR instruction count changed from 11 to 10; Delta: -1 [-Rpass-analysis=
size-info]
fib.c:6:24: remark: fib not inlined into fib because it should never be inlined (cost=never): recursive [-Rpass-missed=inline]
    return (fib(n-1) + fib(n-2));
           ^
fib.c:6:13: remark: fib not inlined into fib because it should never be inlined (cost=never): recursive [-Rpass-missed=inline]
    return (fib(n-1) + fib(n-2));
           ^
fib.c:3:9: remark: Tail Call Elimination: IR instruction count changed from 10 to 9; Delta: -1 [-Rpass-analysis=
size-info]
int64_t fib(int64_t n) {
    ^
fib.c:3:9: remark: Tail Call Elimination: Function: fib: IR instruction count changed from 10 to 9; Delta: -1 [-Rpass-analysis=
size-info]
fib.c:6:24: remark: Cannot SLP vectorize list: vectorization was impossible with available vectorization factors [-Rpass-missed=slp-vectorizer]
    return (fib(n-1) + fib(n-2));
           ^
```

编译器报告的“好”与“不好”

“好”：编译器生成的分析/优化报告能够告知我们编译器在编译时具体做了哪些分析/优化的详细信息。通过这些信息我们可以了解编译器进行程序分析的结论，也可以了解哪些代码被成功进行了变换（优化），这对于我们更加深入地理解编译优化的过程非常有帮助。

“不好”：大多数编译器用户都比较难以理解这些报告呈现出来的信息，需要有更多的上下文才能理解这些报告。

- 信息比较多；
- 报告内包含很多编译术语的缩写（e.g 上一页中的 *SROA*）；
- 报告中呈现的信息可能不完整；
- 不是所有的LLVM “优化遍”都会产生详细的优化报告（这个跟开发相应“优化遍”的开发者相关）。

Q: 到底什么是SROA?

本次课程专题

- 编译器常见优化能力
- 标量优化
- 结构体优化
- 函数调用优化
- 循环优化
- 链接时间优化 (LTO)

新宾利法则

数据结构

- 打包和编码
- 数据增添
- 预先计算
- 编译时做初始化
- 缓存
- 延迟计算
- 稀疏性

程序逻辑

- 常数折叠与传播
- 公共子表达式消除
- 代数恒等替换
- 创建快速通道
- 逻辑短路
- 判断排序
- 组合判断

循环

- 循环不变量外提
- “哨兵”设置
- 循环展开
- 循环合并
- 消除无用迭代

函数

- 函数内联
- 尾递归消除
- 粗化递归

编译器在满足一定的限制条件下可以自动完成这些优化

更多的编译优化

数据结构

- 寄存器分配
- 内存变量寄存器化
- 复杂数据结构的标量替代
- 内存对齐

函数

- 函数体内判断外移
- 参数消除

程序逻辑

- 冗余指令消除
- 强度削弱
- 死码删除
- 等价指令替换
- 分支重排序
- 全局值编号

循环

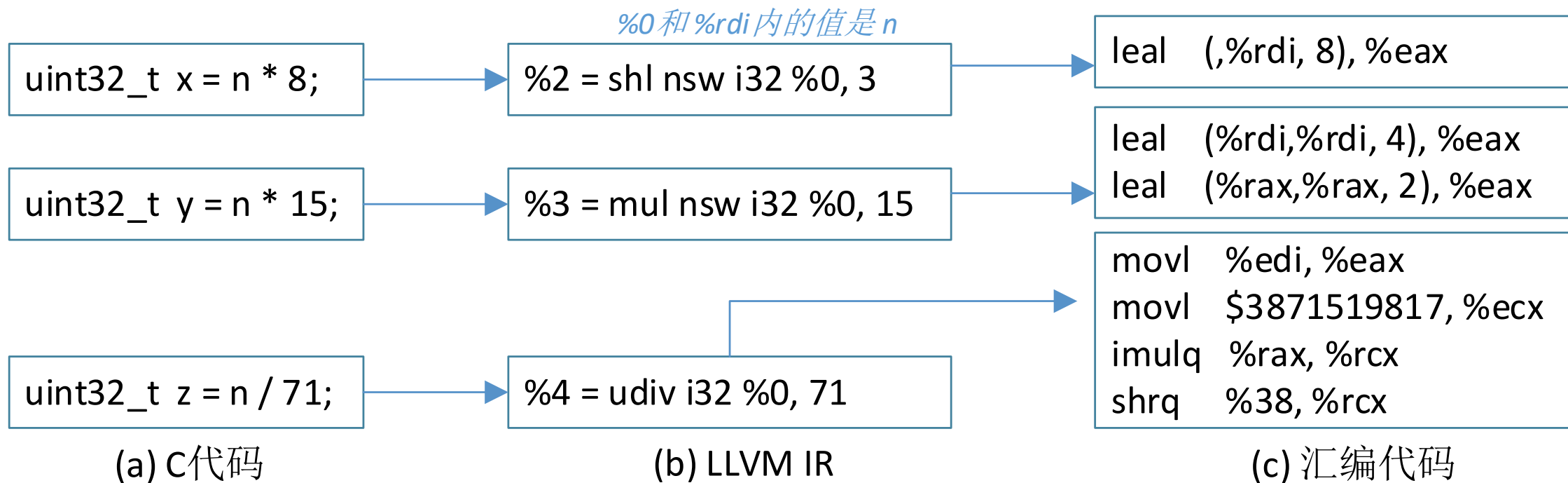
- 向量化优化
- 循环体内判断外移 ←
- 等价替换
- 循环分裂
- 循环歪斜
- 循环分块
- 循环交换

新的编译优化方法
在不断涌现！

```
for (...)
  A
  if (lic)
    B
  C
```

```
if (lic)
  for (...)
    A; B; C
else
  for (...)
    A; C
```

针对计算进行编译优化的三个例子



Q: 如何理解最右边的汇编代码?

优化案例: n个天体的行为仿真

编译器的很多优化能力来自于多个编译优化的组合!

假设在二维空间中有n个天体, 这n个天体在引力作用下的行为可以用程序来仿真。

$$F_{21} = (Gm_1 m_2 / |r_{12}|^2) \text{unit}(r_{21})$$



天体在引力影响下的行为仿真程序

```
typedef struct vec_t {  
    double x, y;  
} vec_t;  
typedef struct body_t {  
    // Position vector  
    vec_t position;  
    // Velocity vector  
    vec_t velocity;  
    // Force vector  
    vec_t force;  
    // Mass  
    double mass;  
} body_t;
```

(a)

```
static vec_t vec_add(vec_t a, vec_t b) {  
    vec_t sum = { a.x + b.x, a.y + b.y };  
    return sum;  
}  
  
static vec_t vec_scale(vec_t v, double a) {  
    vec_t scaled = { v.x * a, v.y * a };  
    return scaled;  
}  
  
static double vec_lenght2(vec_t v) {  
    return v.x * v.x + v.y * v.y;  
}
```

(b)

```
void simulate(body_t *bodies, int nbodies, int nsteps, int time_quantum) {  
    for (int i = 0; i < nsteps; ++i) {  
        calculate_forces(nbodies, bodies);  
        update_positions(nbodies, bodies, time_quantum);  
    }  
}
```

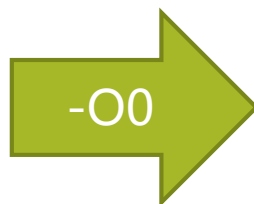
(d)

```
void update_positions(int nbodies, body_t *bodies,  
                     double time_quantum){  
    for (int i = 0; i < nbodies; ++i) {  
        // Compute the new velocity of bodies[i]  
        vec_t new_velocity =  
            vec_scale(bodies[i].force,  
                    time_quantum / bodies[i].mass);  
        // Update the position of bodies[i] based on  
        // the average of its old and new velocity.  
        bodies[i].position =  
            vec_add(bodies[i].position,  
                    vec_scale(vec_add(bodies[i].velocity,  
                                    new_velocity),  
                            time_quantum / 2.0));  
        // Set the new velocity of bodies[i].  
        bodies[i].velocity = new_velocity;  
    }  
}
```

(c)

vec_scale: -O0编译

```
typedef struct vec_t {  
    double x, y;  
} vec_t;  
  
static vec_t  
vec_scale(vec_t v, double a) {  
    vec_t scaled = { v.x * a, v.y * a };  
    return scaled;  
}
```



```
define internal @vec_scale(double %0, double %1, double %2) #0 {  
    %4 = alloca @struct.vec_t, align 8  
    %5 = alloca @struct.vec_t, align 8  
    %6 = alloca double, align 8  
    %7 = bitcast @struct.vec_t* %5 to { double, double }*  
    %8 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 0  
    store double %0, double* %8, align 8  
    %9 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 1  
    store double %1, double* %9, align 8  
    %10 = getelementptr inbounds @struct.vec_t, @struct.vec_t* %4, i32 0, i32 0  
    %11 = getelementptr inbounds @struct.vec_t, @struct.vec_t* %4, i32 0, i32 1  
    %12 = load double, double* %10, align 8  
    %13 = load double, double* %11, align 8  
    %14 = fmul double %12, %13  
    store double %14, double* %10, align 8  
    %15 = getelementptr inbounds @struct.vec_t, @struct.vec_t* %5, i32 0, i32 0  
    %16 = getelementptr inbounds @struct.vec_t, @struct.vec_t* %5, i32 0, i32 1  
    %17 = load double, double* %15, align 8  
    %18 = load double, double* %16, align 8  
    %19 = fmul double %17, %18  
    store double %19, double* %15, align 8  
    %20 = bitcast @struct.vec_t* %4 to { double, double }*  
    %21 = load { double, double }, { double, double }* %20, align 8  
    ret { double, double } %21  
}
```

clang -O0 -S -emit-llvm

*clang编译器版本是11.0.0-2

vec_scale: -O1编译

```
typedef struct vec_t {  
    double x, y;  
} vec_t;  
  
static vec_t  
vec_scale(vec_t v, double a) {  
    vec_t scaled = { v.x * a, v.y * a };  
    return scaled;  
}
```



```
define internal fastcc @vec_scale(  
    double %0, double %1, double %2) unnamed_addr #1 {  
    %4 = fmul double %0, %2  
    %5 = fmul double %1, %2  
    %6 = insertvalue { double, double } undef, double %4, 0  
    %7 = insertvalue { double, double } %6, double %5, 1  
    ret { double, double } %7  
}
```

clang -O1 -S -emit-llvm

*clang编译器版本是11.0.0-2

Q: clang编译器是如何一步一步实现这个优化的?

处理vec_scale LLVM IR(-O0)中的参数a

```
define internal { double, double } @vec_scale(double %0, double %1, double %2) #0 {
```

...

```
%6 = alloca double, align 8
```

...

```
store double %2, double* %6, align 8
```

...

```
%13 = load double, double* %6, align 8
```

```
%14 = fmul double %12, %13
```

...

```
%18 = load double, double* %6, align 8
```

```
%19 = fmul double %17, %18
```

...

```
}
```

```
static vec_t vec_scale(vec_t v, double a) {  
    vec_t scaled = { v.x * a, v.y * a };  
    return scaled;  
}
```

在栈上分配一个存储空间


把a存在栈上

把a从栈上取出

Q: 寄存器%13和%18内的值是什么?

标量参数的拷贝传播优化

```
define internal { double, double } @vec_scale(double %0, double %1, double %2 ) #0 {  
  ...  
  %6 = alloca double, align 8  
  ...  
  store double %2, double* %6, align 8  
  ...  
  %13 = load double, double* %6, align 8  
  %14 = fmul double %12, %2  
  ...  
  %18 = load double, double* %6, align 8  
  %19 = fmul double %17, %2  
  ...  
}
```



第一步：用原始寄存器替代从栈上的取值

第二步：删除死码

经过标量参数的拷贝传播优化后的LLVM IR

```
define internal { double, double } @vec_scale(double %0, double %1, double %2) #0 {  
  %4 = alloca %struct.vec_t, align 8  
  %5 = alloca %struct.vec_t, align 8  
  %6 = alloca double, align 8  
  %7 = bitcast %struct.vec_t* %5 to { double, double }*  
  %8 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 0  
  store double %0, double* %8, align 8  
  %9 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 1  
  store double %1, double* %9, align 8  
  store double %2, double* %6, align 8  
  %10 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %4, i32 0, i32 0  
  %11 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 0  
  %12 = load double, double* %11, align 8  
  %13 = load double, double* %6, align 8  
  %14 = fmul double %12, %13  
  store double %14, double* %10, align 8  
  %15 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %4, i32 0, i32 1  
  %16 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 1  
  %17 = load double, double* %16, align 8  
  %18 = load double, double* %6, align 8  
  %19 = fmul double %17, %18  
  store double %19, double* %15, align 8  
  %20 = bitcast %struct.vec_t* %4 to { double, double }*  
  %21 = load { double, double }, { double, double }* %20, align 8  
  ret { double, double } %21  
}
```

```
define internal { double, double } @vec_scale(double %0, double %1, double %2) #0 {  
  %4 = alloca %struct.vec_t, align 8  
  %5 = alloca %struct.vec_t, align 8  
  %7 = bitcast %struct.vec_t* %5 to { double, double }*  
  %8 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 0  
  store double %0, double* %8, align 8  
  %9 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 1  
  store double %1, double* %9, align 8  
  %10 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %4, i32 0, i32 0  
  %11 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 0  
  %12 = load double, double* %11, align 8  
  %14 = fmul double %12, %2  
  store double %14, double* %10, align 8  
  %15 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %4, i32 0, i32 1  
  %16 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 1  
  %17 = load double, double* %16, align 8  
  %19 = fmul double %17, %2  
  store double %19, double* %15, align 8  
  %20 = bitcast %struct.vec_t* %4 to { double, double }*  
  %21 = load { double, double }, { double, double }* %20, align 8  
  ret { double, double } %21  
}
```

 Q: 结构体参数也能做类似的优化吗?

处理vec_scale LLVM IR(-O0)中的结构体参数

```
define internal { double, double } @vec_scale(double %0, double %1, double %2) #0 {  
  ...  
  %5 = alloca %struct.vec_t, align 8  
  %7 = bitcast %struct.vec_t* %5 to { double, double }*  
  %8 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 0  
  store double %0, double* %8, align 8  
  %9 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 1  
  store double %1, double* %9, align 8  
  ...  
  %11 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 0  
  %12 = load double, double* %11, align 8  
  %14 = fmul double %12, %2  
  ...  
  %16 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 1  
  %17 = load double, double* %16, align 8  
  %19 = fmul double %17, %2  
  ...  
}
```

在栈上为结构体
分配存储空间

存储结构体的第
一个域

存储结构体的第
二个域

读取结构体的第
一个域

读取结构体的第
二个域

Q: %12中的值是什么?

对结构体的第一个域进行拷贝传播优化

```
define internal { double, double } @vec_scale(double %0, double %1, double %2) #0 {  
  ...  
  %5 = alloca %struct.vec_t, align 8  
  %7 = bitcast %struct.vec_t* %5 to { double, double }*  
  %8 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 0  
  store double %0, double* %8, align 8  
  %9 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 1  
  store double %1, double* %9, align 8  
  ...  
  %11 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 0  
  %12 = load double, double* %11, align 8  
  %14 = fmul double %0, %2  
  ...  
  %16 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 1  
  %17 = load double, double* %16, align 8  
  %19 = fmul double %17, %2  
  ...  
}
```

死码删除

用%0替换了%12

对结构体的第二个域进行拷贝传播优化

```
define internal { double, double } @vec_scale(double %0, double %1, double %2) #0 {  
  ...  
  %5 = alloca %struct.vec_t, align 8  
  %7 = bitcast %struct.vec_t* %5 to { double, double }*  
  %8 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 0  
  store double %0, double* %8, align 8  
  %9 = getelementptr inbounds { double, double }, { double, double }* %7, i32 0, i32 1  
  store double %1, double* %9, align 8  
  ...  
  %11 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 0  
  %12 = load double, double* %11, align 8  
  %14 = fmul double %0, %2  
  ...  
  %16 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %5, i32 0, i32 1  
  %17 = load double, double* %16, align 8  
  %19 = fmul double %1, %2  
  ...  
}
```

死码删除

用%1替换了%17

经过参数优化后vec_scale的LLVM IR

```
define internal { double, double } @vec_scale(double %0, double %1, double %2) #0 {  
  %4 = alloca %struct.vec_t, align 8  
  %10 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %4, i32 0, i32 0  
  %14 = fmul double %0, %2  
  store double %14, double* %10, align 8  
  %15 = getelementptr inbounds %struct.vec_t, %struct.vec_t* %4, i32 0, i32 1  
  %19 = fmul double %1, %2  
  store double %19, double* %15, align 8  
  %20 = bitcast %struct.vec_t* %4 to { double, double }*  
  %21 = load { double, double }, { double, double }* %20, align 8  
  ret { double, double } %21  
}
```

Q: 对于返回值是否也可以做类似的优化?

vec_scale最终的优化结果

```
define internal fastcc { double, double } @vec_scale(double %0, double %1, double %2) unnamed_addr #1 {  
    %4 = fmul double %0, %2  
    %5 = fmul double %1, %2  
    %6 = insertvalue { double, double } undef, double %4, 0  
    %7 = insertvalue { double, double } %6, double %5, 1  
    ret { double, double } %7  
}
```

总结：编译器会尽量把各种数据结构的变量存储在寄存器内！

update_positions: 函数调用优化

update_functions中的一句C代码:

```
vec_add(bodies[i].position,  
vec_scale(vec_add(bodies[i].velocity,  
                new_velocity),  
time_quantum / 2.0));
```

```
%6 = fmul double %2, 5.000000e-01
```

```
...
```

```
%26 = extractvalue { double, double } %25, 0
```

```
%27 = extractvalue { double, double } %25, 1
```

```
%28 = call fastcc { double, double }
```

```
@vec_scale(double %26, double %27, double %6)
```

```
define internal fastcc { double, double } @vec_scale(double %0, double %1, double %2) unnamed_addr #1 {  
    %4 = fmul double %0, %2  
    %5 = fmul double %1, %2  
    %6 = insertvalue { double, double } undef, double %4, 0  
    %7 = insertvalue { double, double } %6, double %5, 1  
    ret { double, double } %7  
}
```

Q: 是否可以函数内联?

函数内联

```
%6 = fmul double %2, 5.000000e-01
```

```
...
```

```
%26 = extractvalue { double, double } %25, 0
```

```
%27 = extractvalue { double, double } %25, 1
```

```
%28 = call fastcc { double, double } @vec_scale(double %26, double %27, double %6)
```

```
%4.in = fmul double %26, %6
```

```
%5.in = fmul double %27, %6
```

```
%28 = insertvalue { double, double } undef, double %4.in, 0
```

```
%28 = insertvalue { double, double } %28, double %5.in, 1
```

```
ret { double, double } %7
```

“删除” 函数调用(call)及函数返回(ret)指令

从vec_scale中拷贝LLVM IR

函数内联后进一步的优化机会

函数内联带来更多的优化机会！

```
%6 = fmul double %2, 5.000000e-01
...
%26 = extractvalue { double, double } %25, 0
%27 = extractvalue { double, double } %25, 1
%4.in = fmul double %26, %6
%5.in = fmul double %27, %6
%28 = insertvalue { double, double } undef, double %4.in, 0
%28 = insertvalue { double, double } %28, double %5.in, 1
%29 = extractvalue { double, double } %28, 0
%30 = extractvalue { double, double } %28, 1
```

倒数第4条和倒数第3条LLVM IR指令把%4.in与%5.in两个寄存器的值打包进一个含有两个双精度浮点域的结构体，然后倒数第2条和倒数第1条LLVM IR指令又立即把刚被打包进入结构体的两个域值取出来，其实这4条LLVM IR指令都可以被删除，后续LLVM IR中对%29与%30两个寄存器的引用可以分别被%4.in及%5.in两个寄存器替换。

链式函数调用

```
vec_add(bodies[i].position, vec_scale(vec_add(bodies[i].velocity, new_velocity), time_quantum / 2.0));
```

```
%6 = fmul double %2, 5.000000e-01
```

```
...
```

```
%25 = call fastcc { double, double } @vec_add(double %22, double %24, double %19, double %20)
```

```
%26 = extractvalue { double, double } %25, 0
```

```
%27 = extractvalue { double, double } %25, 1
```

```
%4.in = fmul double %26, %6
```

```
%5.in = fmul double %27, %6
```

```
...
```

```
%35 = call fastcc { double, double } @vec_add(double %32, double %34, double %4.in, double %5.in)
```

Q: 这两个vec_add的调用是否也可以内联?

两个vec_add函数调用被内联后的结果

```
vec_add(bodies[i].position, vec_scale(vec_add(bodies[i].velocity, new_velocity), time_quantum / 2.0));
```

```
%6 = fmul double %2, 5.000000e-01  
...  
%26 = fadd double %22, %19  
%27 = fadd double %24, %20  
%4.in = fmul double %26, %6  
%5.in = fmul double %27, %6  
...  
%36 = fadd double %32, %4.in  
%37 = fadd double %34, %5.in
```

(a)

```
double scale = time_quantum / 2.0;  
double xv = bodies[i].velocity.x + new_velocity.x;  
double yv = bodies[i].velocity.y + new_velocity.y;  
double sxv = xv * scale;  
double syv = yv * scale;  
double new_x = bodies[i].position.x + sxv;  
double new_y = bodies[i].position.y + syv;
```

(b)

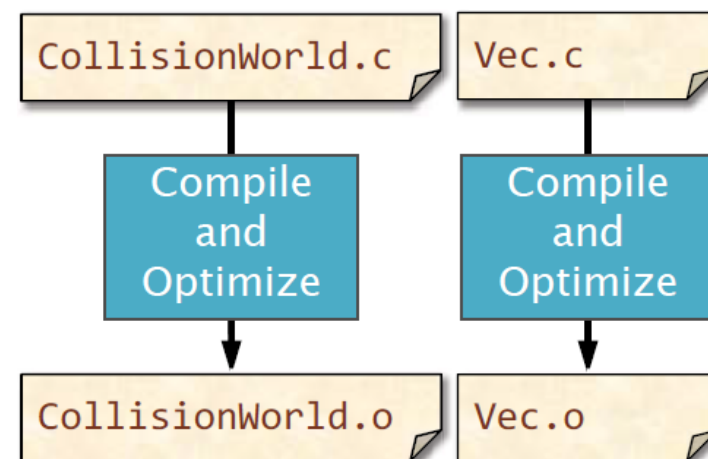
(b) 是和 (a) 语义等价的C代码！

总结： 函数内联以及函数内联后的其它优化能够大大消除函数调用的开销！

函数内联问题

为什么编译器不内联所有的函数调用呢？

- 函数内联会增加代码大小，从而可能影响性能；
- 有些函数是不能被内联的，比如说递归函数调用，只有尾递归调用（recursive tail calls）才能被内联；
- 编译器无法内联一个不在本编译单元定义的函数，除非进行跨编译单元的全程序优化（whole-program optimization）；



如何控制函数内联？

问题：编译器通常是如何判定内联一个函数是否会影响性能的呢？

答案：通常编译器只能根据一些启发式规则（比如函数体的大小）来进行**猜测**。

程序员可以通过一些程序属性来告知编译器是否做函数内联优化：

- 对于确定需要被内联的函数，用 `__attribute__((always_inline))` 属性去修饰；
- 对于确定不能被内联的函数，用 `__attribute__((no_inline))` 属性去修饰；
- 如果需要做全程序优化，可以用 **LTO** (Link-Time Optimization) 编译选项去促使相关的全局优化在程序链接时进行。

函数特化 (Function Specialization)

- 函数特化对同一个函数（假设为 f ）不同调用点（ $C1$ 、 $C2$ 、... C_m ）的实参进行分析并按某些实参值的组合进行分类，从而使得原来的一个函数定义（ f ）变成了 n 个函数定义（ f_1 、 f_2 、... f_n ）。分类以后对函数 f 的每个调用点 C_j （ $1 \leq j \leq m$ ）进行替换，用对某个 f_i （ $1 \leq i \leq n$ ）的调用取代对原函数 f 的调用。一般来说函数特化后的某些实参可能变成了常数，或者某些实参之间的关系变成了一种恒定关系，这样函数体的控制流或者数据流会变得简单，从而达到优化效果。

```
void f(int a)
{
    if (a >= 0) {
        /* Code executed when a >= 0 */
    }
    else {
        /* Code executed when a < 0 */
    }
}
```

```
void f1()
{
    /* Code executed when a >= 0 */
}

void f2()
{
    /* Code executed when a < 0 */
}
```

循环优化

循环在程序设计语言中是广泛使用的一个控制结构。由于循环体不断地被反复迭代执行，一个程序的性能通常被这个程序中的主要循环所决定，所以循环是编译器的主要优化对象之一。

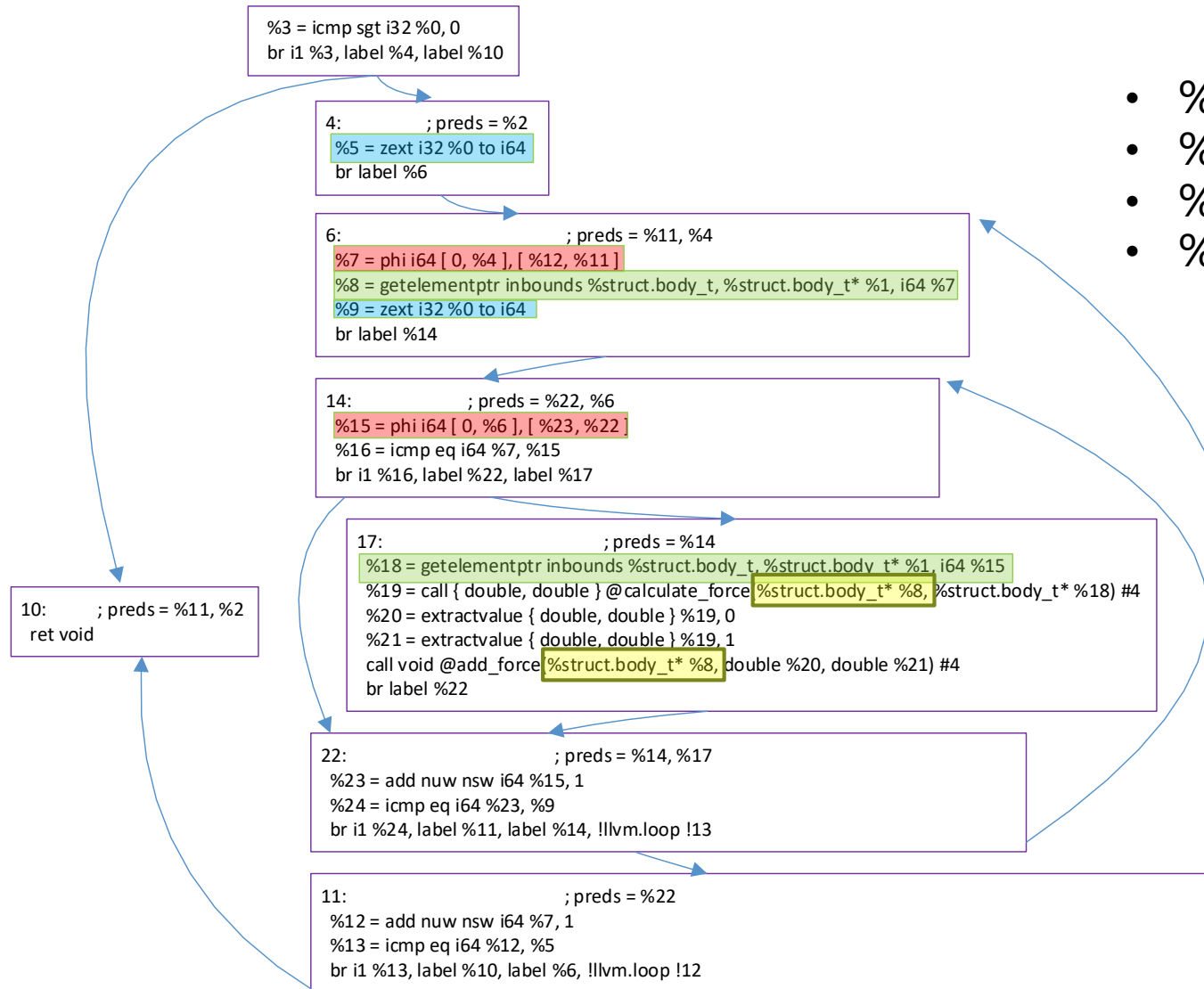
calculate_forces的循环不变量外提机会

```
void calculate_forces(int nbodies, body_t *bodies) {  
    for (int i = 0; i < nbodies; ++i) {  
        for (int j = 0; j < nbodies; ++j) {  
            if (i == j) continue;  
            add_force(&bodies[i],  
                    calculate_force(&bodies[i], &bodies[j]));  
        }  
    }  
}
```

Q: 用"*clang -O1 -S -emit-llvm*"来编译, 结果会如何?

循环不变量外提优化示例

- %0、%5和%9对应的都是参数nbodies
- %7对应于循环控制变量i
- %15对应于循环控制变量j
- %8等于 &bodies[i]



```
void calculate_forces(int nbodies, body_t *bodies) {
    for (int i = 0; i < nbodies; ++i) {
        body_t *bi = &bodies[i];
        for (int j = 0; j < nbodies; ++j) {
            if (i == j) continue;
            add_force(bi, calculate_force(bi, &bodies[j]));
        }
    }
}
```

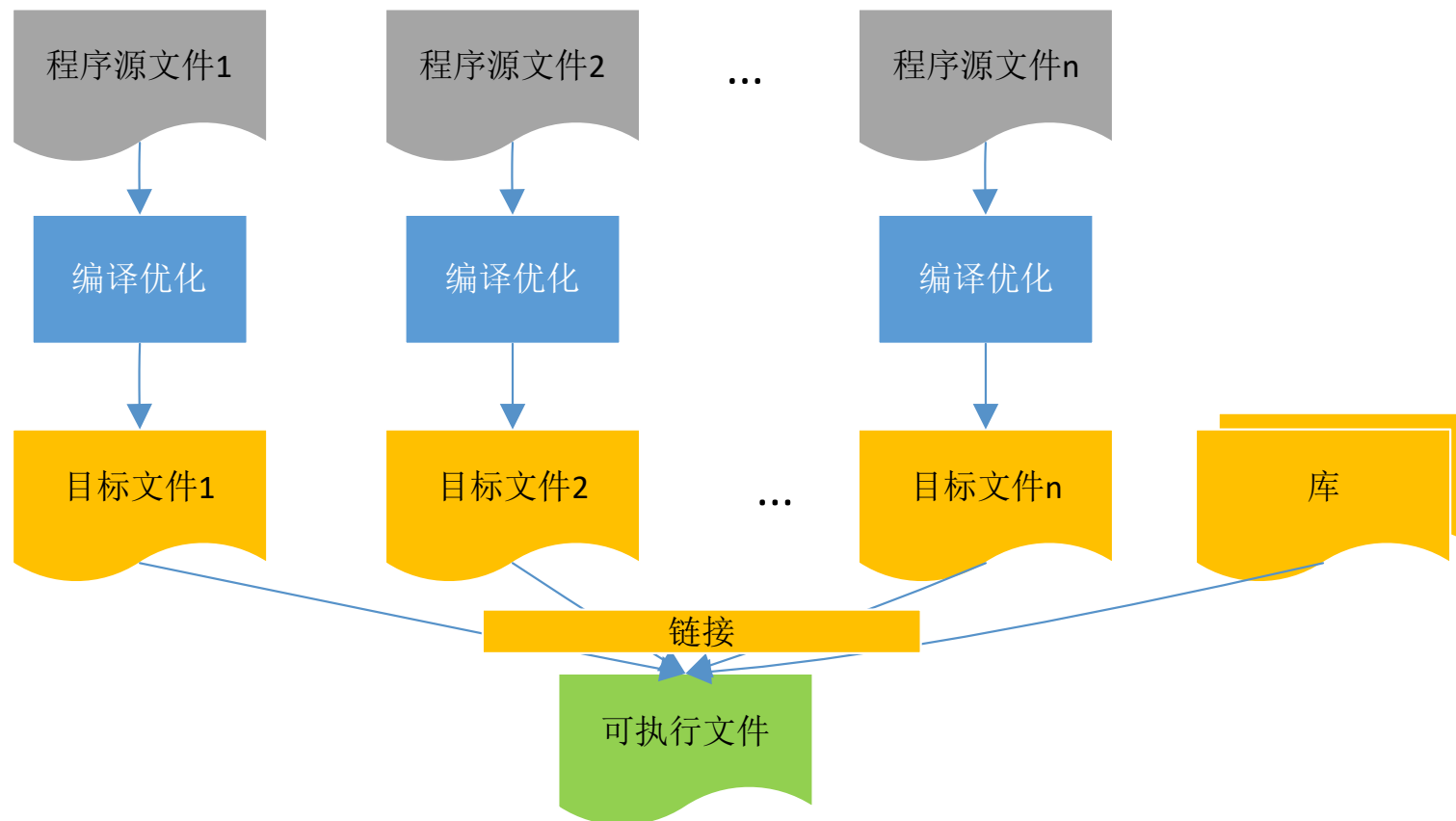

编译器不能做的优化

```
void calculate_forces(int nbodies, body_t *bodies) {  
    for (int i = 0; i < nbodies; ++i) {  
        for (int j = 0; j < nbodies; ++j) {  
            if (i == j) continue;  
            add_force(&bodies[i],  
                    calculate_force(&bodies[i], &bodies[j]));  
        }  
    }  
}
```

从物理学的角度上我们可以知道 $F_{12} = -F_{21}$ ，然而这样一种利用对称性来减少重复计算的优化对编译器来说是一个挑战，毕竟依赖编译器来自动发掘出这个对称性是一件非常困难的事情。

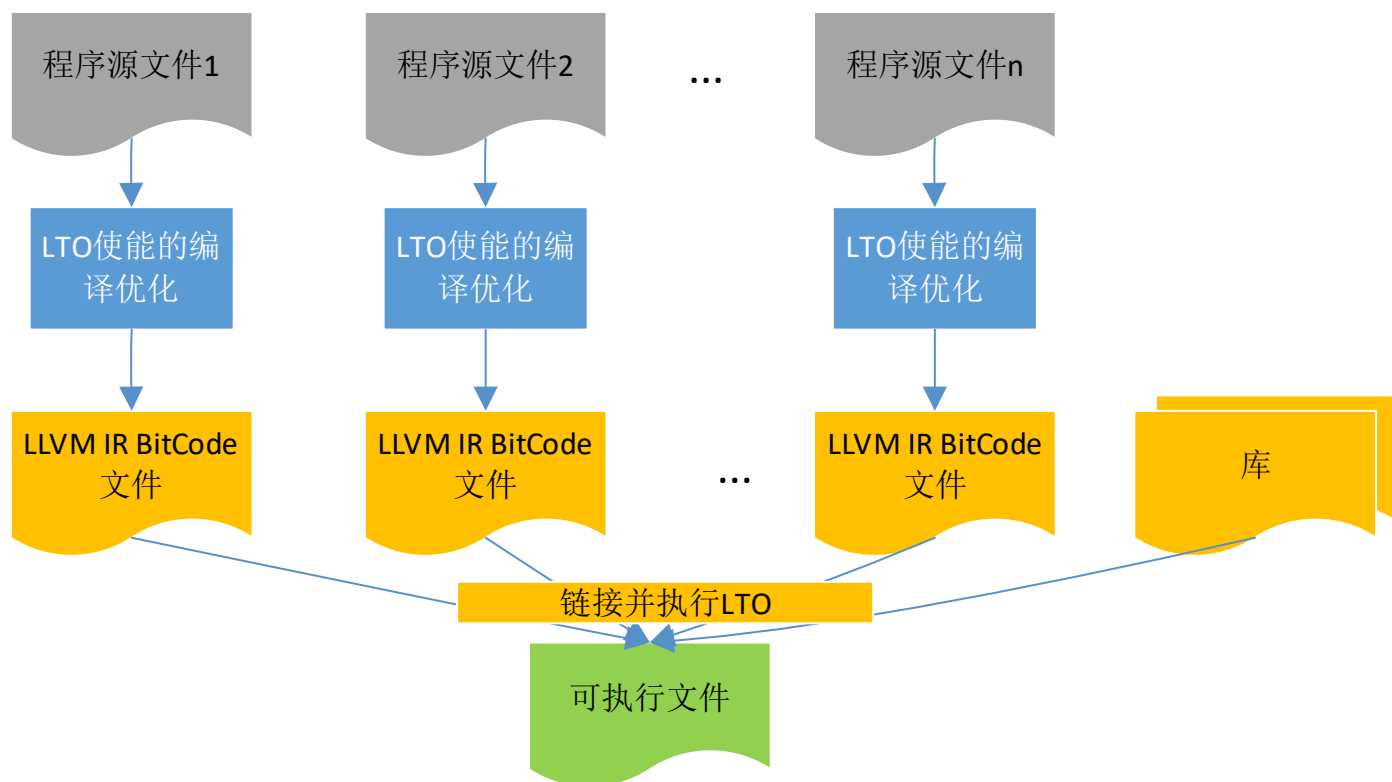
常见的以文件为单位的编译过程

在编译器的执行过程中，编译器的编译对象是一个个包含高级语言所编写程序的源文件，每一个源文件也叫作一个编译单元。编译器在编译一个文件时，只能对当前被编译文件内的代码进行变换和优化。



LLVM编译器中进行LTO的大致流程

为了让跨文件的优化成为可能，很多现代编译器支持链接时间优化（Link-Time Optimization，简称LTO）。以LLVM编译器的链接时间优化为例，每个程序源文件先被编译成LLVM IR的BitCode而不是目标文件，在链接的时候，所有文件的LLVM IR BitCode都可以被看见，从而允许进行更大范围的基于全局所有文件的优化。



Clang/LLVM 的LTO编译选项

在Clang/LLVM中，链接时间优化的编译选项是 *-flto*

- 每个程序源文件都可以用 *-flto* 编译器选项编译成LLVM BitCode
- 在链接阶段，需要用到gold链接器（而不是通常使用的链接器ld）去把各个BitCode文件链接在一起
 - *-flto -fuse-ld=gold*

本次课程总结

- 编译器是执行代码优化的一个强大工具，本次课程介绍的编译优化方法只是冰山一角；
- 编译器所做的优化必须保证语义等价，所以在在一个优化所需相关信息比较难确认时，编译器只能选择遵守保守原则不去做这个优化；
- 很多编译优化可以减少程序的工作量，一个编译优化进行后也可能会带来其它的编译优化机会；
- 通过分析编译器运行过程中产生的程序分析/优化报告，我们可以更加清楚地理解编译器在编译一个程序时做了哪些分析/优化以及又有哪些优化没有被成功实施；
- 如果通过阅读编译器在编译过程中产生的报告还不足以获得所需要的理解，深入分析编译生成的LLVM IR或者汇编代码也是一种途径；
- 如果编译单元内的优化还不足以达成优化目标，我们也可以尝试链接时间优化等跨文件的优化方法。

