中国科学技术大学
University of Science and Technology of China

代码生成

《编译原理和技术(H)》

张昱

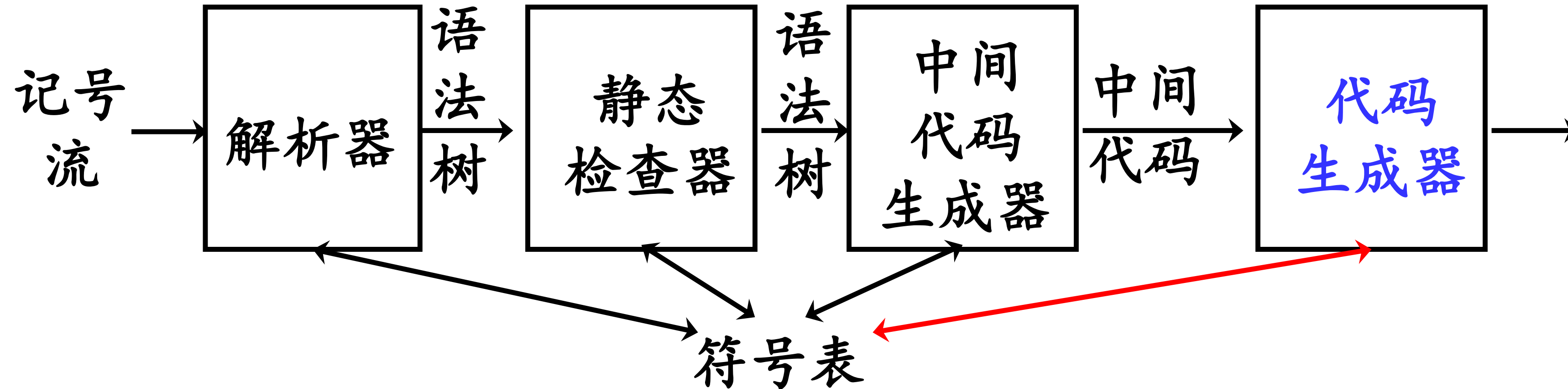
0551-63603804，yuzhang@ustc.edu.cn

中国科学技术大学

计算机科学与技术学院

中国科学技术大学
University of Science and Technology of China

本章内容



本章内容

- 代码生成：中间代码IR→目标机器指令序列
- 涉及目标机器指令选择、寄存器分配和计算次序选择等基本问题

8.1 代码生成器设计中的问题

- 目标程序
- 指令选择
- 寄存器的分配和指派
- 计算次序

代码生成器设计中的问题

- 目标程序(target program)
  - **绝对机器语言程序(absolute machine-language …)**
    - 目标程序将装入到内存的固定地方
    - 粗略地说，相当于现在的可执行目标模块
  - **可重定位目标模块(relocatable object module)**
    - 代码中含重定位信息，以适应重定位要求

代码生成器设计中的问题

□ 目标程序

■ 可重定位目标模块

```
.L7:
        testl %eax,%eax
        je .L3
        testl %edx,%edx
        je .L7
        movl %edx,%eax
        jmp .L7
.L3:
        leave
        ret
```

可重定位目标模块中，需要有蓝色部分的重定位信息

代码生成器设计中的问题

目标程序

- **绝对机器语言程序**
  - 目标程序将装入到内存的固定地方
  - 粗略地说，相当于现在的可执行目标模块
- **可重定位目标模块(relocatable object module)**
  - 代码中含重定位信息，以适应重定位要求
  - 允许对程序模块分别编译
  - 调用其它先前编译好的程序模块

代码生成器设计中的问题

目标程序

- 绝对机器语言程序
- 可重定位目标模块
  - 代码中含重定位信息，以适应重定位要求
  - 允许对程序模块分别编译
  - 调用其它先前编译好的程序模块
- 汇编语言程序(assembly-language program)
  - 生成汇编程序，可以避免编译器重复汇编器的工作
  - 从教学角度，增加可读性

代码生成器设计中的问题

指令的选择(instruction selection)

- 目标机器指令系统的性质决定指令选择的难易程度，指令系统的统一性和完备性是重要因素
- 指令的速度和机器特点是另一些重要的因素

代码生成器设计中的问题

代码生成机制

逐条语句地产生代码，常常会得到低质量的代码

例：三地址语句x = y + z (假设x、y和z都是静态分配)

```
MOV y,    R0    /* 把y装入寄存器R0 */
ADD  z,   R0    /* 把z加到R0上 */
MOV R0,   x     /* 把R0存入x中 */
```

代码生成器设计中的问题

语句序列 a = b + c
d = a + c
的一种目标代码如下：

```
MOV   b,   R0
ADD   c,   R0
MOV   R0,  a
MOV   a,   R0
ADD   c,   R0
MOV   R0,  d
```

```
int a,b,c,d;
void f() {
    a = b+c;
    d = a+c;
}
```

X86-32位汇编

```
movl   b(%rip), %edx
movl   c(%rip), %eax
addl   %edx, %eax
movl   %eax, a(%rip)
movl   a(%rip), %edx
movl   c(%rip), %eax
addl   %edx, %eax
movl   %eax, d(%rip)
```

```
.text
.comm   a,4,4
.comm   b,4,4
.comm   c,4,4
.comm   d,4,4
```

声明为未初始化的通用内存区域
符号，长度，对齐

代码生成器设计中的问题

中国科学技术大学 University of Science and Technology of China

arm-32位汇编

```
ldr     r3, .L2
ldr     r2, [r3]
ldr     r3, .L2+4
ldr     r3, [r3]
add     r3, r2, r3
ldr     r2, .L2+8
str     r3, [r2]
ldr     r3, .L2+8
ldr     r2, [r3]
ldr     r3, .L2+4
ldr     r3, [r3]
add     r3, r2, r3
ldr     r2, .L2+12
str     r3, [r2]
```

arm-32位汇编

```
.L2:
    .word   b
    .word   c
    .word   a
    .word   d
```

X86-32位汇编

```
movl    b, %edx
movl    c, %eax
addl    %edx, %eax
movl    %eax, a
movl    a, %edx
movl    c, %eax
addl    %edx, %eax
movl    %eax, d
```

```
int a,b,c,d;
void f() {
        a = b+c;
        d = a+c;
}
```

```
.text
.comm   a,4,4
.comm   b,4,4
.comm   c,4,4
.comm   d,4,4
```

声明为未初始化的通用内存区域

符号，长度，对齐

代码生成器设计中的问题

源程序	x86-64	AArch64	LoongArch64
int a,b,c,d;	.globl a .bss .align 4 .type a, @object .size a, 4 a: .zero 4 ...	.comm a,4,4 .comm b,4,4 .comm c,4,4 .comm d,4,4	.comm a,4,4 .comm b,4,4 .comm c,4,4 .comm d,4,4

X86-64:

GCC: (Ubuntu 10.5.0-1ubuntu1~22.04) 10.5.0

AArch64：

GCC: (GNU) 7.3.0

LoongArch64:

GCC: (LoongArch GNU toolchain rc1.5 (20240802)) 8.3.0

源程序	x86-64	AArch64	LoongArch64

a = b+c;	movl b(%rip), %edx
	movl c(%rip), %eax
	addl %edx, %eax
	movl %eax, a(%rip)
		adrp x0, b
		add x0, x0, :lo12:b
		ldr w1, [x0]
		adrp x0, c
		add x0, x0, :lo12:c
		ldr w0, [x0]
		add w1, w1, w0
		adrp x0, a
		add x0, x0, :lo12:a
		str w1, [x0]
			la.global $r12,b
			ld.w $r13,$r12,0
			la.global $r12,c
			ld.w $r12,$r12,0
			add.w $r12,$r13,$r12
			or $r13,$r12,$r0
			la.global $r12,a
			st.w $r13,$r12,0

d = a+c;	movl a(%rip), %edx
	movl c(%rip), %eax
	addl %edx, %eax
	movl %eax, d(%rip)
		adrp x0, a
		add x0, x0, :lo12:a
		ldr w1, [x0]
		adrp x0, c
		add x0, x0, :lo12:c
		ldr w0, [x0]
		add w1, w1, w0
		adrp x0, a
		add x0, x0, :lo12:d
		str w1, [x0]
			la.global $r12,a
			ld.w $r13,$r12,0
			la.global $r12,c
			ld.w $r12,$r12,0
			add.w $r12,$r13,$r12
			or $r13,$r12,$r0
			la.global $r12,d
			st.w $r13,$r12,0

中国科学技术大学 University of Science and Technology of China

代码生成器设计中的问题

语句序列 a = b + c
d = a + e
的一种目标代码如下：

```
MOV   b,   R0
ADD   c,   R0
MOV   R0,  a
MOV   a,   R0
ADD   e,   R0
MOV   R0,  d
```

由于a的值仍然存于寄存器R0中，因此该指令是冗余的。

代码生成器设计中的问题

语句序列 a = b + c

d = a + e

的一种目标代码如下：

MOV	b,	R0
ADD	c,	R0
MOV	R0,	a
~~MOV~~	~~a,~~	~~R0~~
ADD	e,	R0
MOV	R0,	d

如果a不再被使用，该指令也可以删除。

代码生成器设计中的问题

□ 代码生成机制

- 同一中间表示代码可以实现为多组指令序列
  不同实现之间的效率差别是很大的
- 例：语句a = a + 1可以有两种实现方式

```
MOV  a,   R0
ADD  #1,  R0
MOV  R0,  a
```

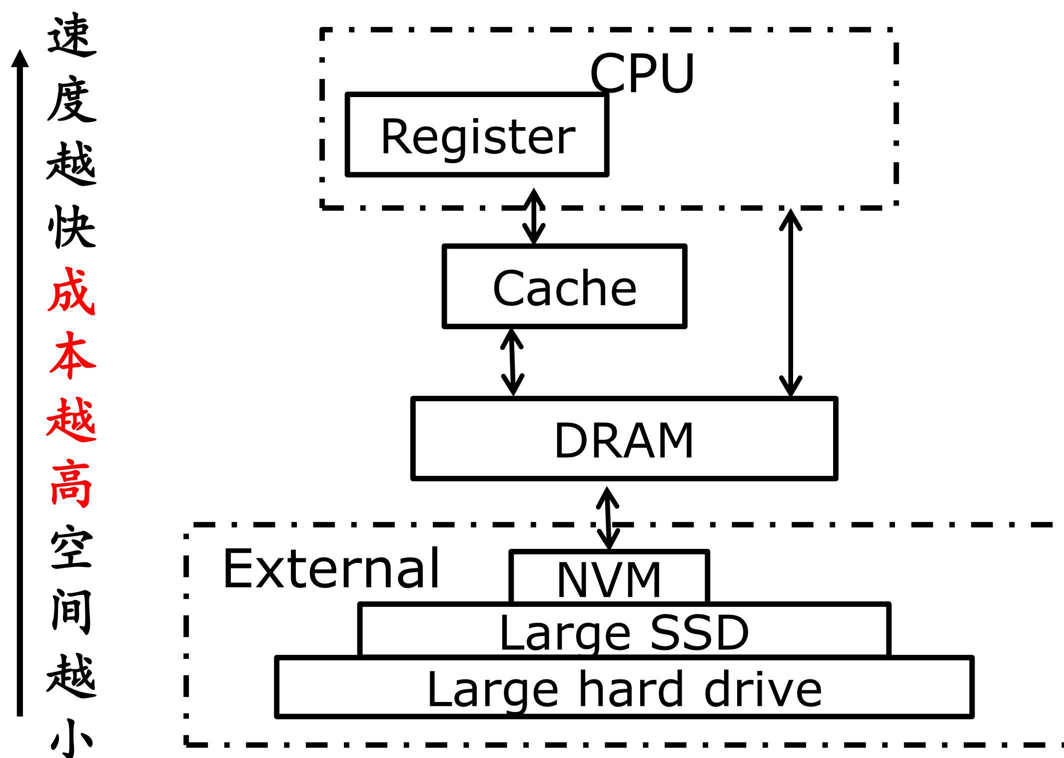
```
INC a
```

- 因此，生成高质量代码需要知道**指令代价**。

代码生成器设计中的问题

代码生成机制

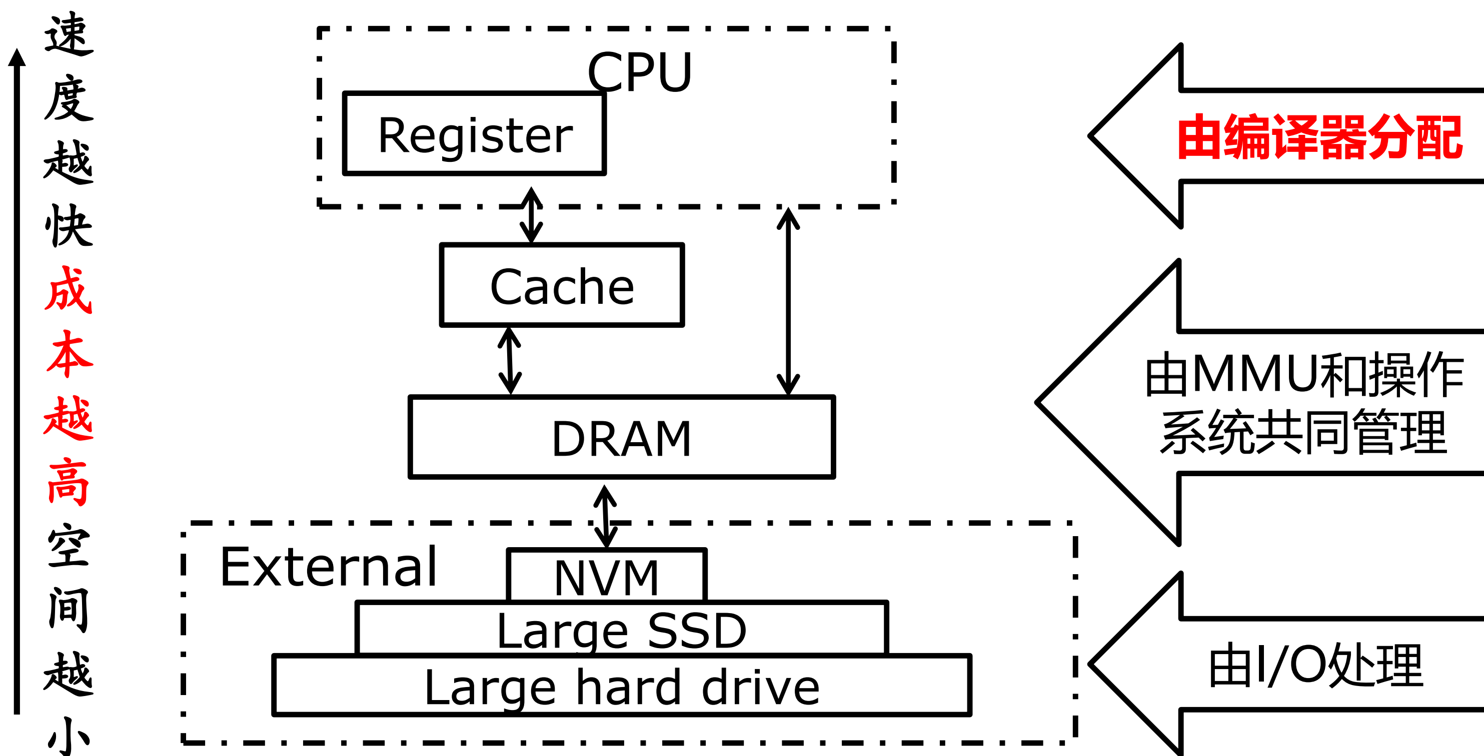
考虑 指令的代价和序列长度、运算对象和结果如何存储



代码生成器设计中的问题

□ 代码生成机制

考虑 指令的代价和序列长度、运算对象和结果如何存储



代码生成器设计中的问题

寄存器的合理使用

相比操作置于内存的运算对象，操作寄存器型操作数的指令要短一些，执行也快一些

- **寄存器分配(register allocation)**
  - 选择驻留在寄存器中的一组变量
- **寄存器指派(register assignment)**
  - 挑选变量要驻留的具体寄存器

代码生成器设计中的问题

计算次序的选择(evaluation order)

- 计算的执行次序会影响目标代码的执行效率
  例如，对表达式而言，一种计算次序可能会比其它次序需要较少的寄存器来保存中间结果
- 选择最佳计算次序是一个NP完全问题

中国科学技术大学
University of Science and Technology of China

8.2 目标语言

- 目标机器指令集
- 指令代价
- LLVM中的目标机器描述:tabgen

目标语言

□ 一个简单目标机器的指令系统

- 字节寻址，四个字节组成一个字
- 有*n*个通用寄存器R0, R1, …, Rn-1
- 二地址指令： op 源， 目的

MOV {源传到目的}

ADD {源加到目的}

SUB {目的减去源}

目标语言

例 指令实例

MOV R0, M

MOV 4(R0), M

4(R0)的值：$contents(4 + contents(R0))$

MOV *4(R0), M

*4(R0)的值：$contents(contents(4 + contents(R0)))$

MOV #1, R0

目标语言

指令的代价(instruction costs)

在上述简单的目标机器上，指令代价简化为

1 + 指令的源和目的寻址模式(addressing mode)的附加代价

目标语言

寻址模式和它们的汇编语言形式及附加代价

模式	形式	地址	附加代价
绝对地址	M	M	1
寄存器	R	R	0
变址	c(R)	$c + contents$(R)	1
间接寄存器	*R	$contents$(R)	0
间接变址	*c(R)	$contents(c + contents(R))$	1
直接量	#c	c	1

目标语言

指令代价简化为

1 + 指令的源和目的地址模式的附加代价

指令	代价
MOV R0， R1	
MOV R5， M	
ADD #1， R3	
SUB 4(R0), *12(R1)	

目标语言

□ 指令代价简化为

1 + 指令的源和目的地址模式的附加代价

指令	代价	
MOV R0， R1	1	寄存器
MOV R5， M	2	寄存器+内存
ADD #1， R3	2	常量+寄存器
SUB 4(R0), *12(R1)	3	变址+间接变址

目标语言

- 例 a = b + c，　a、b和c都静态分配内存单元
  - 可生成

```
MOV b, R0
ADD c, R0
MOV R0, a
```

  - 也可生成

```
MOV b, a
ADD c, a
```

目标语言

- 例 a = b + c， a、b和c都静态分配内存单元
  - 可生成

    MOV b, R0

    ADD c, R0　　　　代价= 6

    MOV R0, a

  - 也可生成

    MOV b, a

    ADD c, a　　　　代价= 6

目标语言

- 例 a = b + c， a、b和c都静态分配内存单元
  - 若R0，R1和R2分别含a，b和c的地址，则可生成

    MOV *R1, *R0

    ADD *R2, *R0　　　　代价= 2

  - 若R1和R2分别含b和c的值，并且b的值在这个赋值后不再需要，则可生成

    ADD R2, R1

    MOV R1, a　　　　代价= 3

LLVM:用tblgen描述后端

目标机器

寄存器、寄存器类、指令集、调用约定(calling convention)

TableGen

- C++风格的语法：TableGen编程指南
- LLVM中已定义的不同类型的后端
  - RegisterInfo, InstrInfo, AsmWriter...
- 通过提取不同架构的相同信息，避免冗余开发
- TableGen后端生成C++的.inc文件

利用 llvm-tblgen工具处理.td文件，生成描述后端的.inc文件

利用tblgen描述寄存器

中国科学技术大学 University of Science and Technology of China

在X86RegisterInfo.td文件中定义了X86Reg抽象类

```
class X86Reg<string n, bits<16> Enc, list<Register> subregs = []> : Register<n> {
  let Namespace = "X86";
  let HWEncoding = Enc;
  let SubRegs = subregs;
}
```

设置命名空间为X86

在Target.td中定义的寄存器抽象记录

再将X86Reg作为父类定义具体的寄存器

```
let SubRegIndices = [sub_16bit, sub_16bit_hi], CoveredBySubRegs = 1 in {
def EAX : X86Reg<"eax", 0, [AX, HAX]>, DwarfRegNum<[-2, 0, 0]>;
def EDX : X86Reg<"edx", 2, [DX, HDX]>, DwarfRegNum<[-2, 2, 2]>;
def ECX : X86Reg<"ecx", 1, [CX, HCX]>, DwarfRegNum<[-2, 1, 1]>;
def EBX : X86Reg<"ebx", 3, [BX, HBX]>, DwarfRegNum<[-2, 3, 3]>;
def ESI : X86Reg<"esi", 6, [SI, HSI]>, DwarfRegNum<[-2, 6, 6]>;
def EDI : X86Reg<"edi", 7, [DI, HDI]>, DwarfRegNum<[-2, 7, 7]>;
def EBP : X86Reg<"ebp", 5, [BP, HBP]>, DwarfRegNum<[-2, 4, 5]>;
def ESP : X86Reg<"esp", 4, [SP, HSP]>, DwarfRegNum<[-2, 5, 4]>;
def EIP : X86Reg<"eip", 0, [IP, HIP]>, DwarfRegNum<[-2, 8, 8]>;
}
```

利用tblgen描述寄存器类

- 在X86RegisterInfo.td文件中，除了定义寄存器之外，还定义许多寄存器类(Register class)

32位通用寄存器(general-purpose registers)类

```
def GR32 : RegisterClass<"X86", [i32], 32,
                         (add EAX, ECX, EDX, ESI, EDI, EBX, EBP, ESP,
                              R8D, R9D, R10D, R11D, R14D, R15D, R12D, R13D)>;
```

在Target.td中定义的寄存器类抽象记录

利用tblgen描述指令集

- 在X86InstrFormat.td中定义了所有指令的超类：

```
class X86Inst<bits<8> opcod, Format f, ImmType i, dag outs, dag ins,
              string AsmStr, Domain d = GenericDomain>
  : Instruction {
```

- 对于不同类型的指令，再定义不同的抽象类：

```
class I<bits<8> o, Format f, dag outs, dag ins, string asm,
        list<dag> pattern, Domain d = GenericDomain>
  : X86Inst<o, f, NoImm, outs, ins, asm, d> {
```

- 在X86InstrArithmetic.td文件中描述算术指令，如

```
def LEA16r   : I<0x8D, MRMSrcMem,
                 (outs GR16:$dst), (ins anymem:$src),
                 "lea{w}\t{$src|$dst}, {$dst|$src}", []>, OpSize16;
```

生成.inc文件

llvm-tblgen X86.td -gen-register-info

其部分输出：

GR32寄存器类变量在.inc文件中的表示 ➔

```
// GR32 Register Class...
const MCPhysReg GR32[] = {
  X86::EAX, X86::ECX, X86::EDX, X86::ESI,
  X86::EDI, X86::EBX, X86::EBP, X86::ESP,
  X86::R8D, X86::R9D, X86::R10D, X86::R11D,
  X86::R14D, X86::R15D, X86::R12D, X86::R13D,
};
```

```
namespace llvm {

class MCRegisterClass;
extern const MCRegisterClass X86MCRegisterClasses[];

namespace X86 {
enum {
  NoRegister,
  AH = 1,
  AL = 2,
  AX = 3,
  BH = 4,
  BL = 5,
  BP = 6,
```

各个寄存器在.inc中的表示：枚举类型

LLVM中的指令代价

- 在Target/XXX/XXXTransformationInfo.cpp中规定XXX架构中指令代价，以X86为例

保存指令代价的数据结构

```
/// Cost Table Entry
struct CostTblEntry {
  int ISD;
  MVT::SimpleValueType Type;
  unsigned Cost;
};
```

ISD：SelectionDAG结点

Type：目标机器值类型

```
static const CostTblEntry SLMCostTable[] = {
  { ISD::MUL,   MVT::v4i32, 11 }, // pmulld
  { ISD::MUL,   MVT::v8i16, 2  }, // pmullw
  { ISD::MUL,   MVT::v16i8, 14 }, // extend/pmullw/trunc sequence.
  { ISD::FMUL,  MVT::f64,   2  }, // mulsd
  { ISD::FMUL,  MVT::v2f64, 4  }, // mulpd
  { ISD::FMUL,  MVT::v4f32, 2  }, // mulps
  { ISD::FDIV,  MVT::f32,   17 }, // divss
  { ISD::FDIV,  MVT::v4f32, 39 }, // divps
  { ISD::FDIV,  MVT::f64,   32 }, // divsd
  { ISD::FDIV,  MVT::v2f64, 69 }, // divpd
  { ISD::FADD,  MVT::v2f64, 2  }, // addpd
  { ISD::FSUB,  MVT::v2f64, 2  }, // subpd
  // v2i64/v4i64 mul is custom lowered as a series of long:
  // multiplies(3), shifts(3) and adds(2)
  // slm muldq version throughput is 2 and addq throughput 4
  // thus: 3X2 (muldq throughput) + 3X1 (shift throughput) +
  //       3X4 (addq throughput) = 17
  { ISD::MUL,   MVT::v2i64, 17 },
```

部分指令代价的计算是有规律的

8.3 代码生成器的输入

- 中间代码IR
- 基本块优化
- 下次引用信息

三地址代码

一般形式：$x = y\ op\ z$

程序举例

```
prod = 0;
i = 1;
do {
    prod = prod + a[i] * b[i];
    i = i +1;
} while (i <= 20);
```

第i个元素的类型为int

(1) prod = 0
(2) i = 1
(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) $prod = t_6$
(10) $t_7 = i + 1$
(11) $i = t_7$
(12) if i <= 20 goto (3)

元素的地址要转换成按字节寻址

流图(变换成 SSA 格式)

B_1:

(1) prod = 0
(2) i = 1

B_2:

(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) $prod = t_6$
(10) $t_7 = i + 1$
(11) $i = t_7$
(12) if i <= 20 goto (3)

利用流图，可快速找到B_2的前驱基本块，按控制流逆向找到最近对 i和prod 的定值

(1) $prod_1 = 0$
(2) $i_1 = 1$
(3) $i_3 = \phi(i_1, i_2)$
(4) $prod_3 = \phi(prod_1, prod_2)$
(5) $t_1 = 4 * i_3$
(6) $t_2 = a[t_1]$
(7) $t_3 = 4 * i_3$
(8) $t_4 = b[t_3]$
(9) $t_5 = t_2 * t_4$
(10) $t_6 = prod_3 + t_5$
(11) $prod_2 = t_6$
(12) $t_7 = i_3 + 1$
(13) $i_2 = t_7$
(14) if i_2 <= 20 goto (3)

插入到块B_2入口处

B_1:

(1) $prod_1 = 0$
(2) $i_1 = 1$

B_2:

(3) $i_3 = \phi(i_1, i_2)$
(4) $prod_3 = \phi(prod_1, prod_2)$
(5) $t_1 = 4 * i_3$
(6) $t_2 = a[t_1]$
(7) $t_3 = 4 * i_3$
(8) $t_4 = b[t_3]$
(9) $t_5 = t_2 * t_4$
(10) $t_6 = prod_3 + t_5$
(11) $prod_2 = t_6$
(12) $t_7 = i_3 + 1$
(13) $i_2 = t_7$
(14) if i_2 <= 20 goto (3)

基本块的优化（局部优化）

删除局部公共子表达式

a = b + c	⇨	a = b + c
b = a – d		b = a – d
c = b + c		c = b + c
d = a – d		d = b

删除死代码

~~x = y + z~~ x此后不再被引用（x是死变量）

交换相邻的语句

t1 = b + c
t2 = x + y

代数变换

- x = x + 0
- x = x * 1
- x = y ** 2
- x = y * y

下次引用信息

如果一个变量的值当前存放在寄存器中且之后不再被使用，该寄存器就可以被分配给其他变量

变量接下来被使用的信息 => 帮助寄存器的分配和释放

- **名字的引用(use)**：假设三地址码语句i 对 x 赋值，语句j 将 x 作为运算对象，且 i 到j的控制流路径中无其他对x的赋值语句，则称语句j引用了语句i计算的x值
- **计算基本块B中下次引用信息的方法**

  从B中的最后一个语句开始，反向扫描到B的开始处，对每个语句i: x = y op z，在符号表中：
  - 设置x为不活跃和无下次引用
  - 设置y、z为活跃，并把它们的下次引用设置为语句i

中国科学技术大学
University of Science and Technology of China

8.4 一个简单的代码生成器

- 寄存器和地址的描述
- 代码生成算法
- 寄存器选择函数
- 为特殊语句产生代码

一个简单的代码生成器

基本思想

- 依次考虑基本块的每个语句，为其产生代码
  - 跟踪记录哪个值存放在哪个寄存器中
- 假定三地址语句的每种算符都有对应的目标机器算符
- 假定计算结果尽可能长地保留在寄存器中，除非：
  - 该寄存器要用于其它计算，或者
  - 到基本块结束

代码生成中的主要问题

如何最大限度地利用寄存器

一个简单的代码生成器

- 寄存器描述符(descriptor)和地址描述符

例：对a = b + c

- 如果寄存器Ri 含b，Rj含c，且b此后不再活跃
  产生ADD Rj, Ri，结果a在Ri中
- 如果Ri含b，但c在内存单元，b仍然不再活跃
  产生ADD c, Ri，或者产生
  MOV c, Rj
  ADD Rj, Ri
- 若c的值以后还要用，第二种代码较有吸引力

一个简单的代码生成器

- 在代码生成过程中，需要跟踪
  寄存器的内容和名字的地址

  - **寄存器描述符**记住每个寄存器当前存的是什么，即在任何一点，每个寄存器保存若干个(包括零个)名字的值

    例：

    ```
                // 语句前，R0保存变量a的值
    b = a       // 不为该语句产生任何指令
                // 语句后，R0保存变量a和b的值
    ```

一个简单的代码生成器

在代码生成过程中，需要跟踪

寄存器的内容和名字的地址

- **寄存器描述符**记住每个寄存器当前存的是什么，即在任何一点，每个寄存器保存若干个（包括零个）名字的值

- **名字（变量）的地址描述符**记住运行时每个名字的当前值可以在哪个场所找到。这个场所可以是寄存器、栈单元、内存地址、甚至是它们的某个集合

  例：产生MOV c, R0后，c值可在R0和c的存储单元找到

一个简单的代码生成器

在代码生成过程中，需要跟踪

寄存器的内容和名字的地址

- **寄存器描述符**记住每个寄存器当前存的是什么，即在任何一点，每个寄存器保存若干个（包括零个）名字的值
- **名字（变量）的地址描述符**记住运行时每个名字的当前值可以在哪个场所找到。这个场所可以是寄存器、栈单元、内存地址、甚至是它们的某个集合

  例：产生MOV c, R0后，c值可在R0和c的存储单元找到
- 名字的地址信息存于符号表，另建寄存器描述表
- 这两个描述在代码生成过程中是变化的

一个简单的代码生成器

寄存器选择函数

- 函数*getReg*(*I*)返回保存I: x = y *op* z的x值的场所L
  - 如果名字y在R中，这个R不含其它名字的值, 并且在执行x = y *op* z后y不再有下次引用，那么返回这个R作为L
  - 否则，如果有的话，返回一个空闲寄存器
  - 否则，如果x在块中有下次引用，或者*op*是必须用寄存器的算符，那么找一个已被占用的寄存器R(可能产生MOV R，M指令，并修改 M的描述)
  - 否则，如果x在基本块中不再引用，或者找不到适当的被占用寄存器，选择x的内存单元作为L

一个简单的代码生成器

代码生成算法

- 对每个三地址语句 x = y *op* z
  - 调用函数*getReg*决定放y *op* z计算结果的场所L
  - 查看y的地址描述，确定y值当前的一个场所y′
  - 如果y的值还不在L中，产生指令MOV y′，L
  - 产生指令*op* z′，L，其中z′是z的当前场所之一
  - 如果y和/或z的当前值不再引用，在块的出口也不活跃，并且还在寄存器中，那么修改寄存器描述,使得不再包含y和/或z的值

一个简单的代码生成器

- 赋值语句 $d = (a - b) + (a - c) + (a - c)$
  - 编译产生三地址语句序列：

$$t_1 = a - b$$
$$t_2 = a - c$$
$$t_3 = t_1 + t_2$$
$$d = t_3 + t_2$$

一个简单的代码生成器

语　句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$			
$t_2 = a - c$			
$t_3 = t_1 + t_2$			
$d = t_3 + t_2$			

一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含t_1	t_1在R0中
$t_2 = a - c$			
$t_3 = t_1 + t_2$			
$d = t_3 + t_2$			

一个简单的代码生成器

语　句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含t_1	t_1在R0中
$t_2 = a - c$	MOV a, R1 SUB c, R1	R0含t_1 R1含t_2	t_1在R0中 t_2在R1中
$t_3 = t_1 + t_2$			
$d = t_3 + t_2$			

一个简单的代码生成器

语　句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含t_1	t_1在R0中
$t_2 = a - c$	MOV a, R1 SUB c, R1	R0含t_1 R1含t_2	t_1在R0中 t_2在R1中
$t_3 = t_1 + t_2$	ADD R1,R0	R0含t_3 R1含t_2	t_3在R0中 t_2在R1中
$d = t_3 + t_2$			

一个简单的代码生成器

语　句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含t_1	t_1在R0中
$t_2 = a - c$	MOV a, R1 SUB c, R1	R0含t_1 R1含t_2	t_1在R0中 t_2在R1中
$t_3 = t_1 + t_2$	ADD R1,R0	R0含t_3 R1含t_2	t_3在R0中 t_2在R1中
$d = t_3 + t_2$	ADD R1,R0	R0含d	d在R0中
	MOV R0, d		d在R0和内存中

一个简单的代码生成器

- 前三条指令可以修改，使执行代价降低

修改前	修改后
MOV a, R0	MOV a, R0
SUB b, R0	MOV R0, R1
MOV a, R1	SUB b, R0
SUB c, R1	SUB c, R1
. . .	. . .

一个简单的代码生成器

为特殊语句产生代码

变址和指针语句

变址与指针运算的三地址语句的处理和二元算符的处理相同

<table>
<tr><td rowspan="2">语句</td><td colspan="2">i在寄存器Ri中</td><td colspan="2">i在内存Mi中</td><td colspan="2">i在栈中</td></tr>
<tr><td>代码</td><td>代价</td><td>代码</td><td>代价</td><td>代码</td><td>代价</td></tr>
<tr><td>a = b[i]</td><td>MOV b(Ri), R</td><td>2</td><td>MOV Mi, R
MOV b(R), R</td><td>4</td><td>MOV Si(Rs), R
MOV b(R), R</td><td>4</td></tr>
<tr><td>b[i] = a</td><td>MOV a, b(Ri)</td><td>3</td><td>MOV Mi, R
MOV a, b(R)</td><td>5</td><td>MOV Si(Rs), R
MOV a, b(R)</td><td>5</td></tr>
</table>

一个简单的代码生成器

- 为特殊语句产生代码
  - 变址和指针语句
  - 条件语句
    - 根据寄存器的值是否为下面六个条件之一进行分支
      - 负、零、正、非负、非零和非正

例，if x < y goto z

- 把 x 减 y 的值存入寄存器R
- 如果R的值为负，则跳到z

一个简单的代码生成器

为特殊语句产生代码

- 变址和指针语句
- 条件语句
  - 用条件码表示计算结果或装入寄存器的值是负，零还是正

    例：若if x < y goto z

    - CMP x, y
    - CJ< z

```
int a, b, c;
int main(){
        a = b + 4;
        if ( a < b )
                c = a;
        else
                c = b;
}
```

```
movl    b, %eax
addl    $4, %eax
movl    %eax, a
movl    a, %edx
movl    b, %eax
cmpl    %eax, %edx
jge     .L2
```

16位程序状态字寄存器PSW

- CF(进位标志位)
- ZF零标志位
- SF符号标志位
- OF溢出标志位
- PF奇偶标志
- AF辅助进位标志

SF=OF， >=跳转

8.5 寄存器分配算法

- 线性扫描算法
- 图着色算法
- LLVM中的寄存器分配

线性扫描算法

给定一个函数中变量的**活跃区间**，该算法将线性扫描所有活跃区间，并以**贪心方式**将寄存器分配给变量。

□ 术语

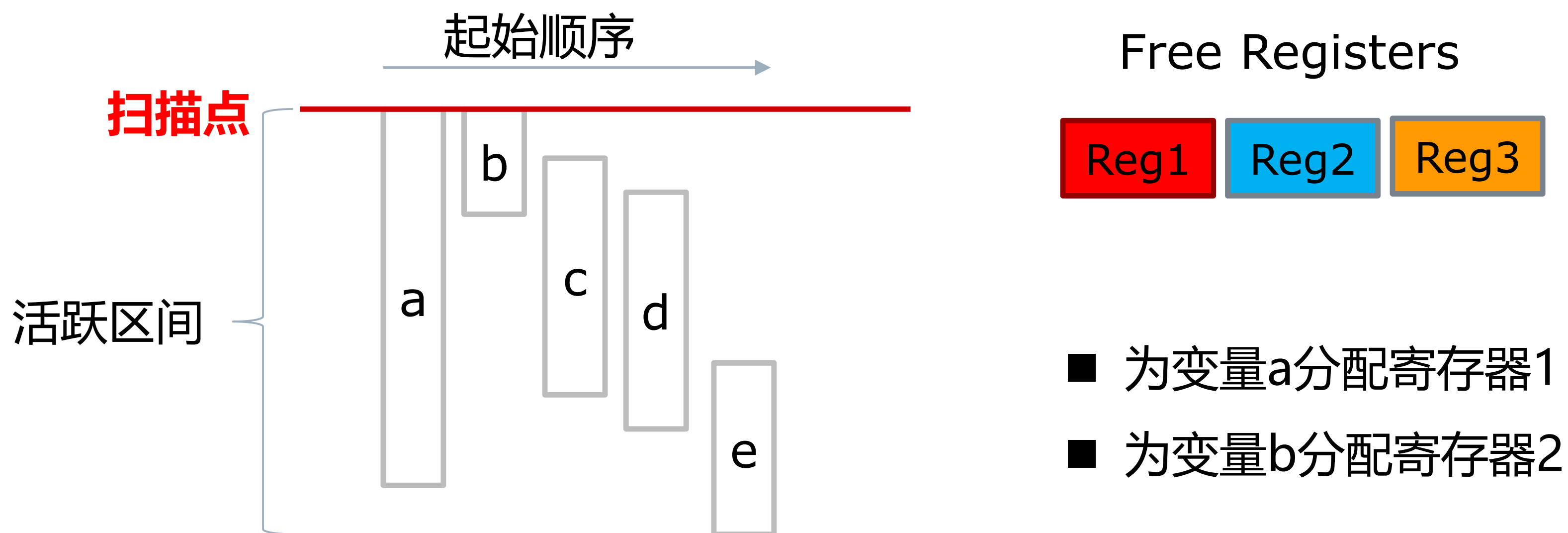
- **活跃区间live interval**：假设 IR 的指令按数字编号，变量 v 的活跃区间就是v被使用的第一条指令的编号i 以及v最后一次被使用的指令编号 j 构成的区间 $[i,j]$
- **激活表active list**：表示已经分配了寄存器的各活跃区间 的表，表中各活跃区间**按照结束位置递增**的顺序排列

[TOPLAS1999] Linear Scan Register Allocation

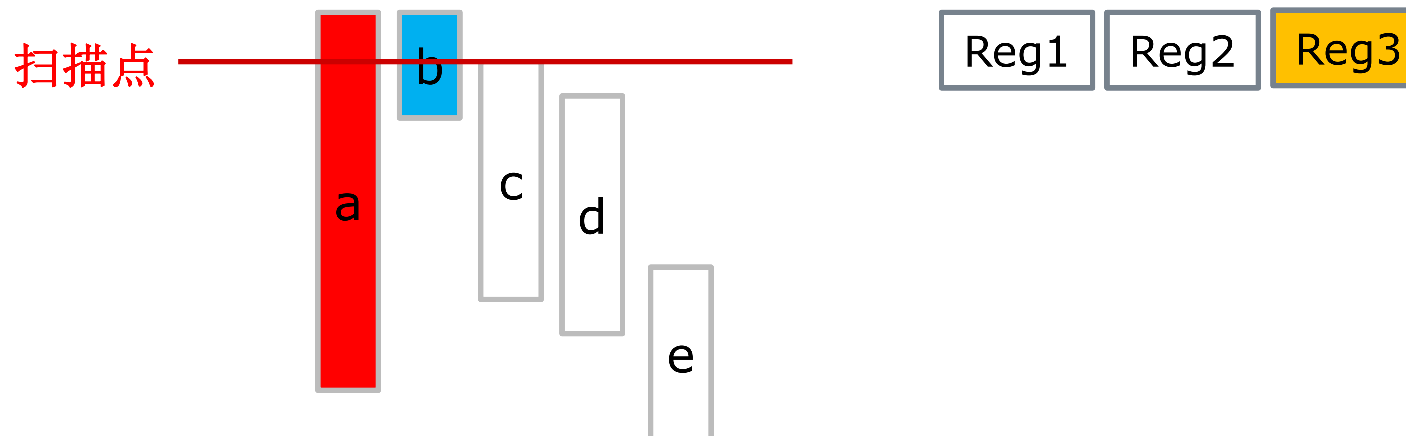
线性扫描算法

□ 算法

- 将所有活跃区间按照起始位置先后排序
- 线性扫描所有活跃区间，为变量分配寄存器
- 当没有空闲寄存器可分配时，溢出**结束位置距当前程序点最远**的活跃区间对应的变量

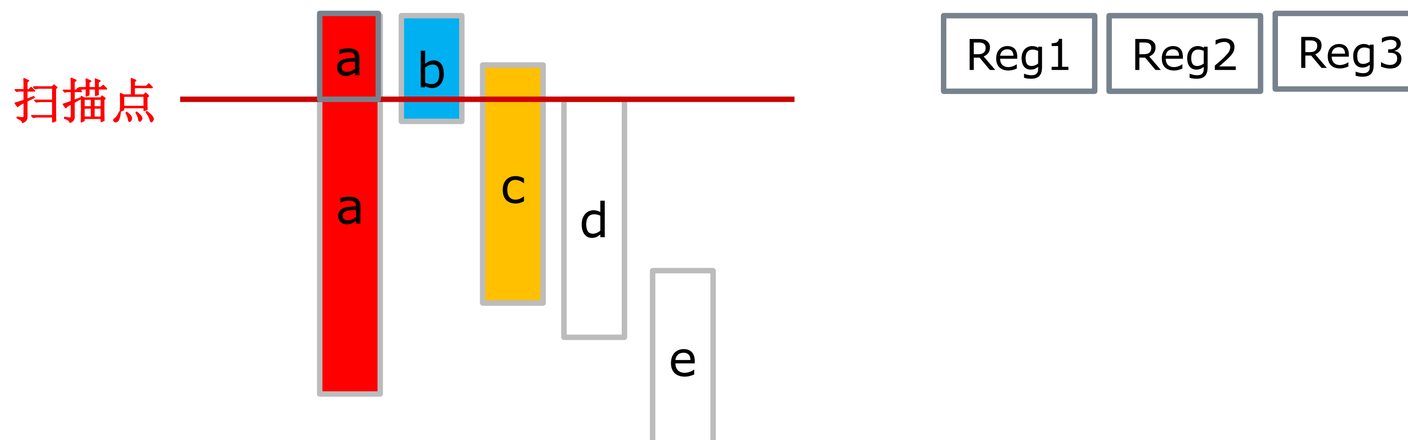


线性扫描算法



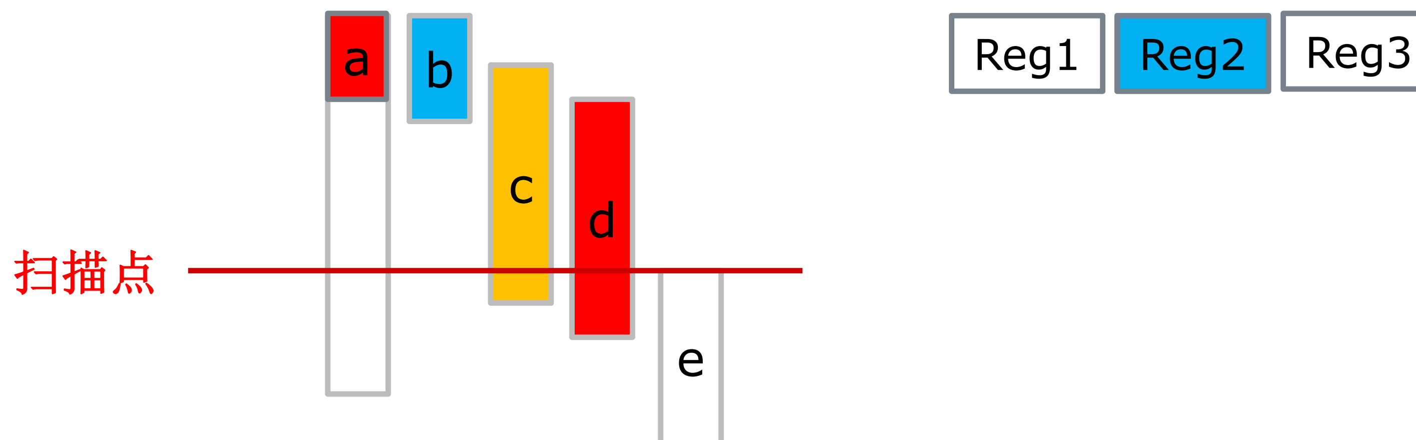
- 为变量c分配寄存器3

线性扫描算法



- 无空闲寄存器，溢出距离当前程序点最远的变量a，为变量d分配寄存器1

线性扫描算法



- 变量b 活跃区间结束，寄存器2恢复空闲状态
- 为变量e分配寄存器2

线性扫描算法

□ 算法

```
LinearScanRegisterAllocation
    active ← {}
    foreach live interval i, in order of increasing start point
        ExpireOldIntervals(i)
        if length(active) = R then
            SpillAtInterval(i)
        else
            register[i] ← a register removed from pool of free registers
            add i to active, sorted by increasing end point

ExpireOldIntervals(i)
    foreach interval j in active, in order of increasing end point
        if endpoint[j] ≥ startpoint[i] then
            return
        remove j from active
        add register[j] to pool of free registers

SpillAtInterval(i)
    spill ← last interval in active
    if endpoint[spill] > endpoint[i] then
        register[i] ← register[spill]
        location[spill] ← new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location[i] ← new stack location
```

局限性：活跃区间是粗粒度的

假设一个变量只在某个程序开头和结尾被使用，则此变量的活跃区间会是整个程序运行区间。

[TOPLAS1999]
Linear Scan
Register Allocation

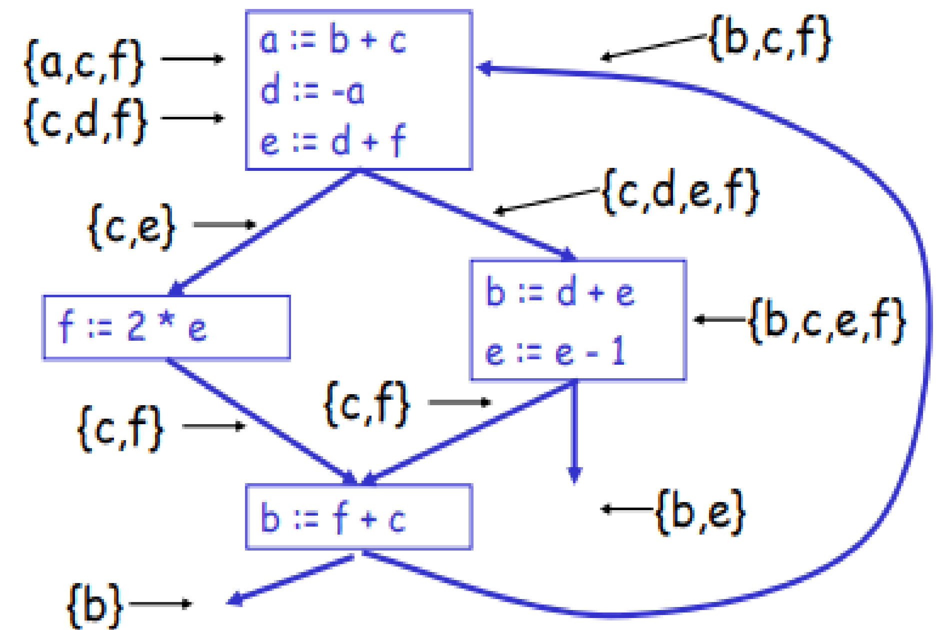
图着色算法

图着色算法

- 变量被赋予不同的节点，在同一个block内同时活跃的变量之间连边，表示不能被分配同一个寄存器
- 对构造出的图进行k着色，k为空闲寄存器的个数
- 按照着色结果对变量进行寄存器赋值

实现：可参考这里

- 计算每个程序点的活跃变量集合

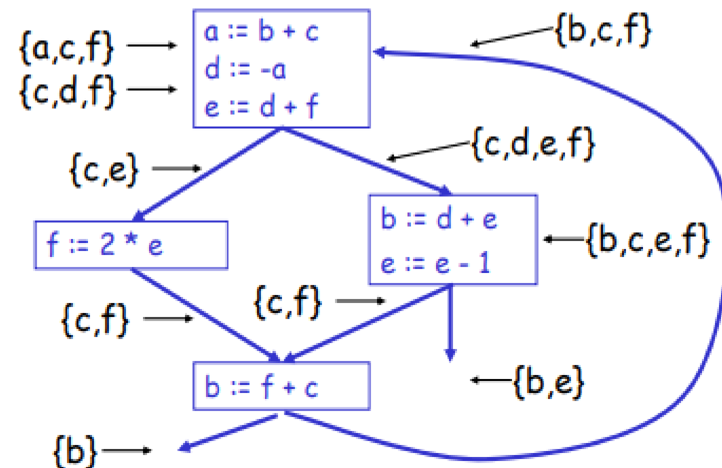


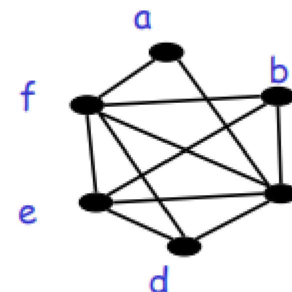
图着色算法

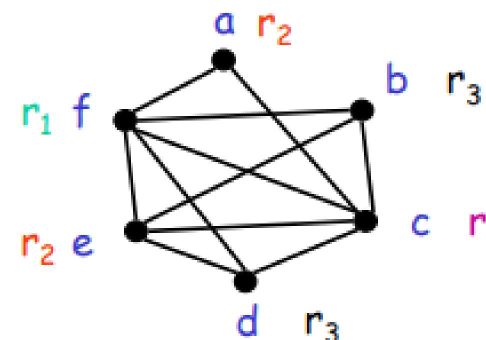
图着色算法

实现：可参考这里

- 计算每个程序点的活跃变量集合
- 构造寄存器干涉图
  (RIG, register interference graph)
  - 顶点：(临时)变量
  - 边(t1,t2)：t1和t2同时活跃
- 运用图着色算法给每个顶点分配**颜色**（此处为**寄存器**）

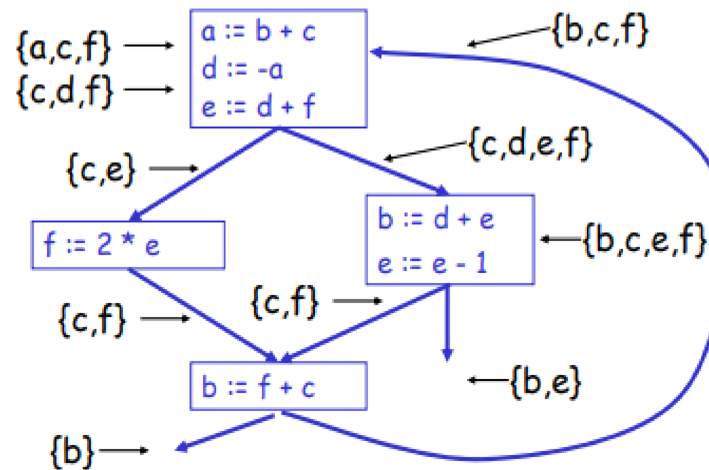


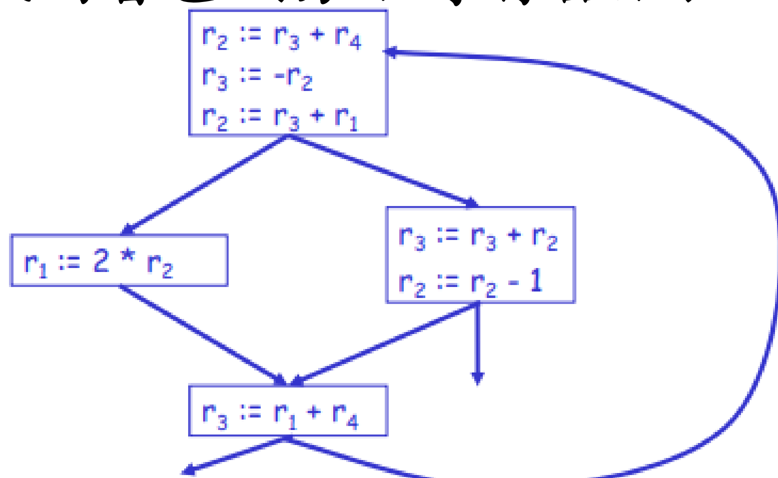


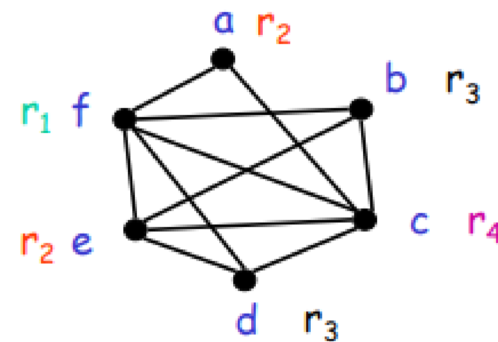


图着色算法

- 图着色算法
- 实现：可参考这里
  - 计算每个程序点的活跃变量集合
  - 构造寄存器干涉图
  - 运用图着色算法给每个顶点分配**颜色**（此处为**寄存器**）
  - 对代码着色（分配寄存器后）







图着色算法

挑战

- 实际的目标平台上，寄存器总是偏少的
- graph-coloring思路本身只解决一个问题：
  当有k种颜色（k个可用寄存器）的时候，程序是否可以**不溢出 (spill)** 就完成着色(寄存器分配), 如果是的话，这个分配是怎样的？
- **它不能解决更重要的spill问题**

LLVM中的寄存器分配算法

- Basic：线性扫描算法的改进，使用启发式的顺序对寄存器进行生存期赋值
- Fast：顺序扫描每条指令，对其中的变量进行寄存器分配，当没有寄存器可以分配时，选择溢出代价*最小的寄存器进行溢出操作
- Greedy：线性扫描算法的改进，Basic分配器的高度优化的实现，合并了全局生存期分割，努力最小化溢出代码的成本
- PBQP：基于分区布尔二次编程（PBQP）的寄存器分配器.其工作原理是构造一个表示寄存器分配问题的PBQP问题，使用PBQP求解器解决该问题，并将该解决方案映射回寄存器分配

openEuler的LLVM平行宇宙计划1

LLVM: 模块化架构(解耦)

- LLVM 9.0之后Apache License，相比GCC的GPL License对商业公司更友好
- LLVM社区贡献者已达2634人，涉及公司150+，LLVM峰会活跃度远超GCC
- 业界厂商（Apple、高通、ARM、Intel等）将自身编译器已切换到LLVM并演进
- 新兴语言（Swift、Rust）也纷纷采用LLVM编译器基础设施
- OS社区：MacOS、Android、ChromOS、OpenMandriva的系统默认编译器选择LLVM，Debian、Fedora允许软件包维护者选择GCC或LLVM构建

LLVM平行宇宙计划

- 使能LLVM编译器构建更多的openEuler软件包，挑战基于LLVM技术栈完成openEuler版本发布，这个工作是平行与目前openEuler版本发布工作的

openEuler的LLVM平行宇宙计划2

收益分析

- 基础性能：LLVM相对GCC更易编译优化增强，LLVM有更强大的LTO能力
- 软件包性能：软件包维护者可以选择GCC或LLVM作为构建工具链，可以释放更多精力在软件功能实现上
- 代码安全：Clang+LLVM对C/C++语言标准遵从更严格

重点工作方向

- 关键软件包的竞争力提升。包括kernel、ceph、mysql、qemu、openjdk等
- 软件包修复&版本发布。共涉及5405个软件包由gcc构建切换为llvm构建

如何加入

- llvm-project：https://gitee.com/openeuler/llvm-project
- 软件包修复&版本发布https://docs.qq.com/s/nQLURYS54g3KVxfWur0NrG
- 例会：双周四下午14:15~15:00
- 邮件列表：compiler@openeuler.org

本章小结

中间代码 → 目标程序

Tree、DAG
基本块+流图
+循环

- 考虑目标机器语言即指令集 → 目标机器指令序列
- 寻址模式 → 可重定位
- 合理利用寄存器
  理解指令代价
  各种优化策略 → 速度快+开销低

简单的代码生成算法