if-conversion 及其在 LLVM 上的实现

高翔

qasdfgtyuiop@gmail.com

April 1, 2014

Abstract

人们追求程序运行的性能的脚步从来都没有停歇过，为了使得程序更快地运行，人们不断的改进处理器的性能。然而，处理器的性能只是程序性能的一部分，要实现高性能，编译器生成的代码的执行效率也至关重要。更先进的构架给编译器提出了更高的要求，人们也一直在探索如何让编译器生成更高效率的代码。由于早先的编译器无法很好地处理控制依赖，Allen 等人于 1983 年提出了 if 转换（if-conversion）的概念 [1]。if-conversion 能够消除程序中的除了后向分支以外的所有分支，这样就使得程序中的控制依赖被转化为数据依赖，编译器可以在此基础上进行进一步优化。1987 年，Ferrante 等人提出了程序依赖图（PDG）的概念 [5]，使得编译器可以轻易的处理控制依赖。尽管现代的编译器可以轻易处理控制依赖，分支语句的存在有时候仍然会严重影响性能。分支语句会中断指令流水，造成严重的性能损失。程序中大约 20% 的语句是分支语句，如果一个分支语句会造成 2 个周期的延迟，那么会有大约 40% 的机器时间浪费在分支上 [3]。此外，分支语句会将程序划分为若干基本块，从而阻断了进一步的优化。If 转换可以通过现代处理器提供的条件执行指令，实现分支消除，这样做不仅可以减少分支造成的延迟，而且可以将若干小的基本块合并成一个大的基本块，进而为后续优化提供便利。由于上述原因，虽然现代编译器能够很好地处理控制依赖，if 转换仍然显得尤为重要。

We have never stopped pursuing the performance of the program, to achieve this goal, the performance of the processor has been highly developed. However, the performance of the processor is only part of the performance of the program. The performance of the code generated by the compiler is also an important factor. The modern architecture has brought greater challenges to the development of compiler. How to design a compiler that generates high performance instructions has been a problem for a long time. Early compiler couldn't process control dependency. To solve this problem, Allen et al. came up with the concept of if-conversion [1], which can remove all branches in the program except for the backward branch. The deletion of branches converts control dependency to data dependency,

making it possible for the compiler to do further optimizations. In 1987, Ferrante et al. came up with the concept of program dependency graph (PDG) [5], making it easier for compilers to handle control dependency. Albeit this improvement, if-conversion is also important since the branch can significantly harm the performance. Branch interrupts the pipeline of instructions, causing great loss in performance. According to statistics, about 20% instructions are branches in a program. Assume that one branch can cause about 2 cycles delay, then about 40% of machine time will be wasted in branch [3]. What's more, branch divides the program to several basic blocks, making it harder for further optimization. If-conversion removes branches by taking use of predicted execution mechanism of modern processor. This can not only eliminate the delay caused by branch, but also merge several small basic blocks to a large basic block, making a lot of optimizations possible. Due to the reasons stated above, if-conversion is also very important.

目录

1 简介

1.1 if-conversion 的发展简史

if-conversion 最早由 Rice 大学的 Allen 等人在 1983 年提出 [1]。当时已有向量机的出现，并且新一代的 Fortran 也一定会提供向量操作来支持向量机。不幸的是，由于之前版本的 Fortran 中不支持向量操作，所以相当大的一部分旧有的 Fortran 程序无法充分利用处理器的新特性，这就意味着，自动向量化的研究势在必行。当时 Rice 大学的人正在开发一个翻译程序，称为并行 Fortran 转换器（Parallel Fortran Converter，简称 PFC），它能将 Fortran 66 或者 Fortran 77 代码翻译成新一代 Fortran 的代码，并且通过自动向量化来充分利用向量操作。当时的编译器进行程序变换还是基于数据依赖，分支语句产生的控制依赖不能非常好的融入到当时的体系中去。与此同时，分支语句的存在也会使得自动向量化无法进行。由于这两个原因，Allen 就提出了将程序中的所有分支语句移除而转换成谓词执行的一些算法。Allen 的算法的目的比较看重向量化，这算法是直接在源代码上进行操作的，它可以将所有的前向分支跟出口分支消除。但是 Allen 的算法没有考虑到如何最小化谓词的分配等的问题。尽管 Allen 的算法可以将分支转换为谓词执行，但是编译器要想判断是否可以进行向量化，必须先对代码进行 if-conversion，这样就将所有的控制依赖转化为数据依赖，然后才能使用基于数据依赖的算法判断是否能够进行向量化，一旦无法向量化，不单单是 if-conversion 执行的时间被浪费掉，而且分支的消除反而会对代码的执行效率产生影响，糟糕的是，if-conversion 已经将代码变得面目全非无法还原。

1987 年，Ferrante 等人提出了程序依赖图（PDG）的概念 [5]。这是一个非常重要的工作，因为 PDG 是一个新的框架，在这个框架下，数据依赖跟控制依赖被统一地进行处理，一切旧有的算法都可以高效地纳入新的框架下。由于不再存在之前的编译器无法处理控制依赖的问题，也就没有必要再利用 if-conversion 将控制依赖转化为数据依赖了。同时这个框架对 if-conversion 的发展也起到了承上启下的作用，后来对 if-conversion 的研究都是在这个框架下的。

虽然不再需要用 if-conversion 将控制依赖转化为数据依赖，但是 if-conversion 并没还有因此失去它的意义。正如上一节所说，程序中的分支语句已经称为高性能体系结构中性能提升的主要障碍，而 if-conversion 恰好可以用谓词执行来代替分支，进而提高效率。只不过问题的核心已经不是像 Allen 所说的那样，将控制依赖转化为数据依赖，也不是向量化，而是减少由于分支语句的存在对程序性能造成的影响，这也对经过 if-conversion 转换之后的代码的性能提出了要求。

1991 年，Park 跟 Schlansker 提出了 RK 算法 [9]，这个算法用于消除所有最内层循环的循环体中的分支语句，这个算法可以最小化谓词的数量，以及定义谓词的操作的数量。但问题是，谓词以及定义谓词的操作的数量最小并不意

味着算法最优，分支消除也会对性能造成一定的负面影响，所以，到底是否进行分支消除，如果是，应该对哪部分分支进行消除也是需要研究的问题。

1992 年，Mahlke 等人提出了 Hyperblock 的概念 [17]。Hyperblock 是一个单入口多出口的谓词化的基本块的集合。这篇文章为选择性地 if-conversion 提供了一个框架，它告诉人们如何只对部分基本块进行 if-conversion 而保持一些其他的基本块不变：将基本块按照性质不同划分到不同的 Hyperblock 中去，然后对每个 Hyperblock 进行 if-conversion 以消除其中的分支。通过将基本块有选择地包括在 Hyperblock 之内或者排除在 Hyperblock 之外，就可以实现性能的最大化。这篇文章还提供了 Hyperblock 形成以后的一些优化算法。但是，这篇文章对 Hyperblock 选择的讨论比较少，并没有告诉人们如何划分 Hyperblock 才是最优的，自然，这也成为之后学者研究的内容。

至此，Hyperblock 概念已经提供了选择性地进行 if-conversion 的机制，并且 RK 算法已经告诉我们如何在选定 Hyperblock 以后产生最优代码。可以说，关于 if-conversion 的研究，剩下的只是策略的问题了。

1993 年，Warter 等人提出了逆向 if-conversion (reverse if-conversion) 的概念 [22]。在这算是一个小插曲，因为作者的工作重心并不是 if-conversion 本身，而只是提出可以利用 if-conversion 来进行全局指令调度：首先通过 if-conversion 将分支消除，然后进行局部调度，最后再用作者提出来的逆向 if-conversion 算法将代码还原。

1997 年，August 等人对 if-conversion 策略进行了详细地讨论 [2]。August 详细讨论了处理器对谓词执行的支持给编译器带来的挑战，Hyperblock 选择等等问题，并且提出了一个平衡控制流跟谓词执行的框架：首先在编译过程的早期大量应用 if-conversion 来发掘谓词执行带来的全部好处，此时形成的 Hyperblock 比目标体系能处理的大得多，然后在比较靠后的编译阶段进行部分逆向 if-conversion，根据目标机器调整每个 Hyperblock 的谓词化代码的数量，以平衡控制流跟谓词数目。

2000 年，Hazelwood 提出了动态 if-conversion 的概念 [7]，这里面的“动态”的意思是说，根据程序的运行情况来实时地对代码进行 if-conversion，相比较动态 if-conversion，之前的 August 的策略叫做静态 if-conversion，因为所有的转换工作以及是否转换的决定已经在编译的时候就确定了。动态 if-conversion 的优点是，静态分析很难得出准确的分支预测失败概率，并且这些参数往往会随着程序的运行而改变，动态 if-conversion 可以随时调整程序到最佳状态以保证程序随时都能高效运行。

1.2 本文的结构

本文的内容是对 if-conversion 研究工作的调研，以及 LLVM 中 if-conversion 的实现两个部分。其中调研可以分成机制跟策略两部分，机制着重解决怎样进行 if-conversion 的问题而不考虑其对性能的影响，而策略则解决选择哪些基本块进行 if-conversion，以及 if-conversion 对性能的影响的问题。

2 if-conversion 的运行机制

控制依赖的概念最早由 Ferrante 等人提出 [5]，RK 算法由 Park 跟 Schlansker 提出 [9]，Hyperblock 的概念由 Mahlke 等人给出 [17]，逆向 if-conversion 则是 Warter 等人的研究成果 [22]。

2.1 控制依赖

定义 1 (控制流图 **(control flow graph)**) 控制流图是一个有向图，它有唯一的入口节点 *START* 以及唯一的出口节点 *STOP*，图中每一个节点最多有两个后继。对于那些有两个后继的节点，它的两条出边分别被赋予属性 *T(真)* 跟 *F(假)*。对于图中的每个节点 N，都存在从 *START* 到 N，以及从 N 到 *STOP* 的路径。

定义 2 (后支配 **(post-dominate)**) 设 V 与 W 是控制流图 G 中的节点，如果从 V 到 *STOP* 的每条路径 *(起始位置的 V 并不计算在路径中)* 都包含 W，则称 V 被 W 后支配，或者 W 后支配 V，记为 $W \operatorname{pdom} V$。

定义 3 (直接后支配 **(immediate post-dominate)**) 设 X,Y 以及 Z 是控制流图 G 中的节点，如果 $Y \operatorname{pdom} X$，并且对于任意的满足 $Z \operatorname{pdom} X$ 以及 $Z \neq Y$ 的节点 Z，都有 $Z \operatorname{pdom} Y$，则称 Y 直接后支配 X，或者 X 被 Y 直接后支配，记为 $Y \operatorname{ipdom} X$

定义 4 (pdom 函数**)** 设 N 是节点的集合，$pdom$ 是一个映射，它将 N 中的节点 x 映射为所有后支配 x 的节点的集合，即 $pdom : N \to 2^N$。$pdom$ 的定义的数学表述为：$pdom\,(x) := \{y \in N : y \operatorname{pdom} x\}$

定义 5 (ipdom 函数**)** 设 N 是节点的集合，$ipdom$ 是一个映射，它将 N 中的节点 x 映射为 x 直接后支配的节点的节点，即 $pdom : N \to N$。$ipdom$ 的定义的数学表述为：$ipdom\,(x) := y \in N$, 其中 $y \operatorname{ipdom} x$

定义 6 (后支配节点树) 直接后支配关系构成一个树，称为后支配节点树。树的节点为控制流图 G 中的节点，若 $y \operatorname{ipdom} x$, 则 y 是 x 的双亲节点。显然，求 $ipdom\,(x)$ 即为求 x 在树中的双亲节点，求 $pdom\,(x)$ 即为求 x 在树中的所有祖先节点的集合。

计算图 G 的后支配节点树的算法见 [11] [6]。

定义 7 (控制依赖 **(control dependent)**) 设 G 为控制流图，X 以及 Y 是图中节点，称 Y 控制依赖于 X 当且仅当：

1. 存在从 X 到 Y 的路径 P，使得 Y 后支配 P 上除了 X 与 Y 的所有节点。
2. X 不被 Y 后支配

从定义可以看出，如果 Y 控制依赖与 X，则那么 X 必然有两个出口，其中一条出口导致 Y 必然被执行，另一条出口则导致 Y 可能不被执行。

定义 8 (CD 函数) 设 N 是节点的集合，C 是控制依赖的集合。CD 函数是一个映射，它将 N 中的节点 x 映射为 x 的所有控制依赖的集合，即 $CD : N \to 2^C$。若将 C 中的元素 $c \in C$ 表示为 $\pm y$，其中 $+y$ 表示 y 的 $true$ 边，$-y$ 表示 y 的 $false$ 边，则 CD 函数的定义可以为：$CD(x) := \{\pm y : x\text{控制依赖于} \pm y\}$

CD 函数的计算算法如算法 1

算法 1: ComputeCD

Input: 控制流图 $G(N, E, Start)$，其中 N 是 G 中节点的集合，E 是 G 中边的集合
Output: CD 函数

```
计算函数 pdom(x) 以及 ipdom(x);
for 每个 [x, y, label] ∈ E 并且 y ∉ pdom(x) do
    Lub = ipdom(x);
    if ¬label then
        x = −x;
    end
    t = y;
    while t ≠ Lub do
        CD(t) = CD(t) ∪ {x};
        t = ipdom(t);
    end
end
```

2.2 RK 算法

CD 函数并非是一个单值函数，不同的 x 可能对应相同的 $CD(x)$。由于具有相同控制依赖的不同节点的执行条件相同，所以将节点转化为谓词执行以后，他们的谓词也应该相同，这样就可以为 CD 函数的值域中的每个元素分配一个与之一一对应的谓词 p，谓词跟值域中元素的集合的对应关系记为 K 函数。显然，K 函数是个单值函数，它的逆 K^{-1} 必然存在，记函数 $R = CD \cdot K^{-1}$。可以看出，这么做将 CD 函数分解为 R 函数跟 K 函数：$CD = R \cdot K$。其中 R 函数负责将每个节点 x 与谓词 p 对应起来，而 K 则将每个谓词跟其对应的控制依赖对应起来。通过 R 函数，具有相同控制依赖（因而具有相同执行条件）的不同节点被映射到同一个谓词，也就是说，R 函数为每个节点实现的谓词分配，节点 x 对应的谓词 $p = R(x)$。至于 K 函数，则将每个谓词跟其控制依赖对应起来，这也为谓词的定义提供了参考。一种不错的谓词定义策略是，对于每一个谓词，在其对应的依赖，的相应节点末尾插入相应的赋值语句。例如 $K(p) = \{y, -z\}$，则在 y 与 z 节点插入 p 的定义语句。插入的时候，如果对该点依赖值为正，则谓词的值就是分支条件，如果为赋，作为谓词的值就是该点

分支条件的非。例如上面例子中，在 y 点插入 $p = t_y$，而在 z 点插入 $p = \neg t_z$。R 函数以及 K 函数的计算算法如算法 2所示。

算法 2: ComputeRK

Input: 控制流图 $G(N, E, Start)$，控制依赖函数 CD

Output: R 函数以及 K 函数

```
p = 1;                                          /* 谓词命名从 1 开始顺序命名 */
for x ∈ N do
    t = CD(x);
    if t ∈ K then
        R(x) = q，其中 q 为使得 K(q) = t 的谓词;
    else
        K(p) = t;
        R(x) = p++;
    end
end
```

计算好 R 函数以及 K 函数，并按照 R 函数跟 K 函数进行谓词分配与定义以后，只要对控制流图进行拓扑排序即可得到转换后的代码。但是此时的代码有一定的问题：存在谓词未被定义就被使用。造成这个问题的原因是因为，控制流图中的所有路径都被压缩到了一条路径上，这样就会导致某些分支的谓词未被定义。

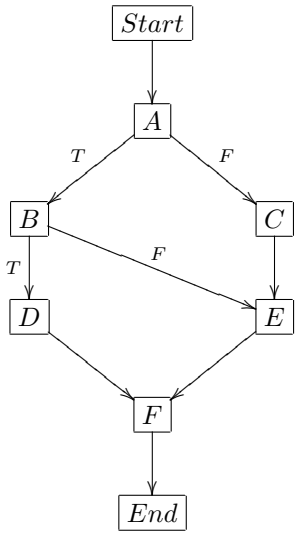


图 1: 谓词未被定义的出现情况

比如图 1中，D 只对 B 有控制依赖，对 A 并没有控制依赖，因此 D 的谓词只会在 B 处被定义，但是如果在 A 处分支的到时候走的是 false 那一支，则 B 将不会被执行，也就是说 D 的谓词将不会被定义。

要解决这个问题最简单最直观的方法是在程序开始的地方将所有的谓词初始化为 $false$，但是未被定义就被使用的谓词毕竟只是少数，这么做会引入很多不必要的浪费。聪明的做法是找到哪些谓词存在未被定义就被使用的问题。定义集合 $A(x) := \{p \in P' : \exists path(x, Stop), 使得 K(p) \cap path(x, Stop) = \varnothing\}$，其中 P' 为所有 $K(p) \neq \varnothing$ 的谓词的集合，$path(x, Stop)$ 为从 x 到 Stop 的路径。显然，$A(Start)$ 就是应该在程序开始的时候初始化为 $false$ 的谓词的集合。由于 $A(x)$ 的计算比较麻烦，一种方便的做法是注意到所有满足 $K(p) \cap \{\pm x \in E : x \,\mathrm{pdom}\, Head\} \neq \varnothing$（$Head$ 指的是 Start 的唯一直接后继）的谓词都不是 $A(Start)$ 的成员，所以可以将所有不满足上述条件的谓词全部在程序的开头初始化为 $false$，这样可以在浪费比较小的处理器资源的前提下省去计算 $A(x)$ 的麻烦。

$A(x)$ 的计算是通过将其转化为定义使用链问题来实现的。将所有的基本块 B_x 抽象为一个点 x，该点使用分配到 B_x 的谓词，并且定义由 $K(p)$ 给出的对应谓词，即对于每个 $b \in N$ 除了 $Stop$，定义 $Use(b) = \{p \in P' : p = R(b)\}$，$Def(b) = \{p \in P' : \pm b \in K(p)\}$，规定 $Use(Stop) = P'$。这样的话通过解数据流方程

$$\begin{cases} IN(b) = Use(b) \cup [OUT(b) - Def(b)] \\ OUT(b) = \bigcup\limits_{s \in succ(b)} IN(s) \end{cases}$$

得到 $IN(x)$，即可求得 $A(Start) = IN(Start)$。具体算法如算法 3

算法 3: SolveDataFlowEquations

Input: 控制流图 $G(N, E, Start)$，R 函数，K 函数

Output: IN 函数，OUT 函数

```
N' = N − {Stop};
计算 Def(b) 以及 Use(b);
IN = Use;
change = true;
while change do
    change = false;
    for b ∈ N' do
        OUT(b) = ∅;
        for s ∈ succ(b) do
            OUT(b) = OUT(b) ∪ IN(s);
        end
        old = IN(b);
        IN(b) = Use(b) ∪ [OUT(b) − Def(b)];
        if old ≠ IN(b) then
            change = true;
        end
    end
end
```

综上所述，if-conversion 的算法概括为算法 4。

算法 4: If-Conversion

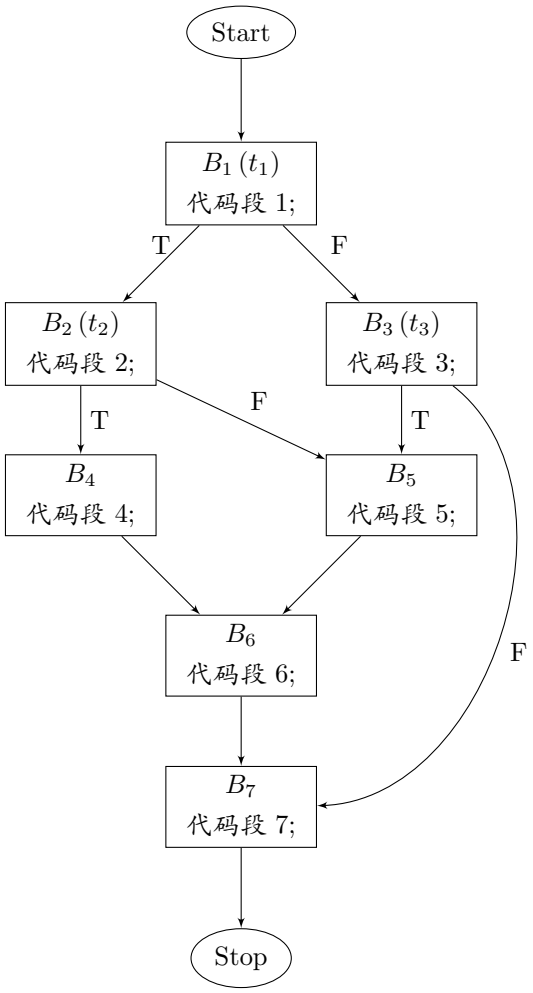
Input: G 是控制流图，N 是 G 中节点的集合，E 是 G 中边的集合

Output: 转换后的代码

```
计算 CD 函数;
计算 RK 函数;
for x ∈ N do
    p = R(x);
    if K(p) ≠ ∅ then
        给 x 分配谓词 p;
    end
end
for 每个谓词 p do
    for ±y ∈ K(p) do
        若是 +y 则在 y 尾部插入 p = t_y;
        若是 −y 则在 y 尾部插入 p = ¬t_y;
    end
end
对 G 进行拓扑排序，删掉分支语句，并对已分配谓词的基本块进行谓词化;
计算 IN(b);
for p ∈ IN(Start) do
    在程序的开头插入“p = false”;
end
```

使用 RK 算法进行 if-conversion 的示例如下：如下控制流图，基本块的标签表示为 $B_i\,(t_i)\,\&p_i$。图中 B_i 为基本块名称；t_i 表示该基本块末尾会按照条件 t_i 分支，如果基本块不进行分支，则 (t_i) 不写；而 p_i 则表示谓词，如果基本块被无条件执行，则 p_i 也不写。

转换前的控制流图为：



计算控制依赖得:

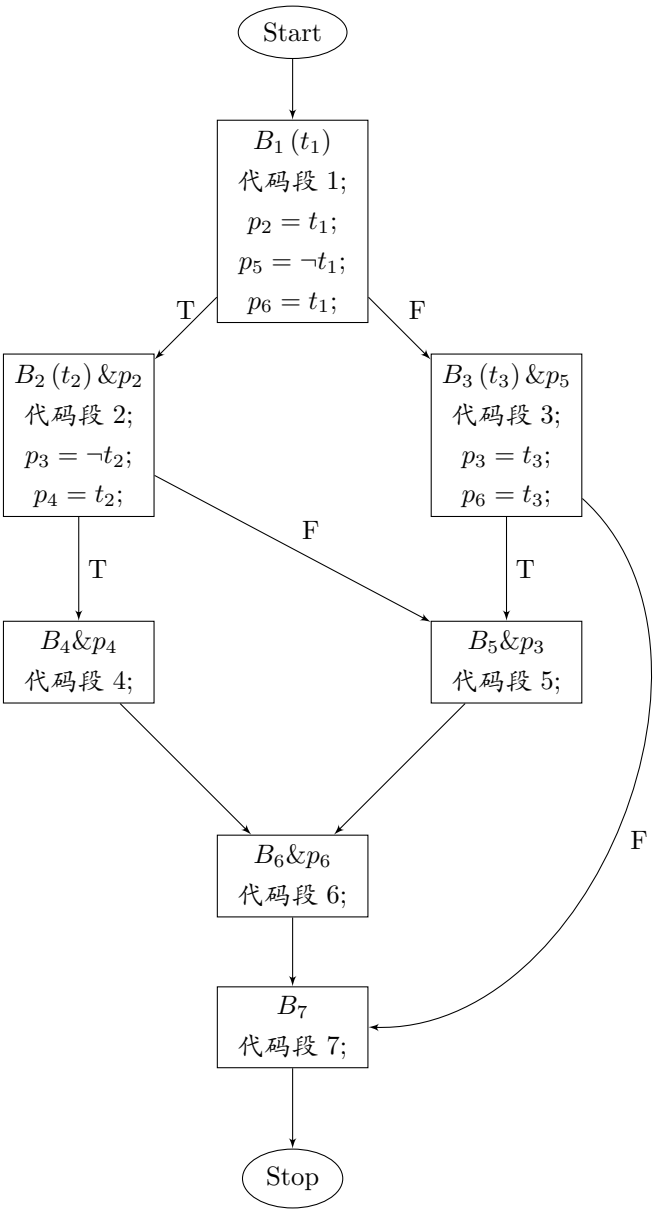
基本块	1	2	3	4	5	6	7
$CD\,(x)$	$\varnothing$	$\{1\}$	$\{-1\}$	$\{2\}$	$\{-2,3\}$	$\{1,3\}$	$\varnothing$

将控制依赖分解为 R 函数跟 K 函数:

基本块	1	2	3	4	5	6	7
$R\,(x)$	p_1	p_2	p_5	p_4	p_3	p_6	p_1

谓词	p_1	p_2	p_3	p_4	p_5	p_6
$K\,(x)$	$\varnothing$	$\{1\}$	$\{-2,3\}$	$\{2\}$	$\{-1\}$	$\{1,3\}$

按照 RK 函数进行谓词分配与定义:



进行拓扑排序即得排序以后的代码：

代码段 1;
$p_2 = t_1$;
$p_5 = \neg t_1$;
$p_6 = t_1$;
if(p_2) 代码段 2;
if(p_2) $p_3 = \neg t_2$;
if(p_2) $p_4 = t_2$;
if(p_5) 代码段 3;
if(p_5) $p_3 = t_3$;
if(p_5) $p_6 = t_3$;
if(p_4) 代码段 4;
if(p_3) 代码段 5;
if(p_6) 代码段 6;
代码段 7;

计算得 $IN\,(Start) = \{p_4\}$，于是最终代码为：

$p_4 = false$;
代码段 1;
$p_2 = t_1$;
$p_5 = \neg t_1$;
$p_6 = t_1$;
if(p_2) 代码段 2;
if(p_2) $p_3 = \neg t_2$;
if(p_2) $p_4 = t_2$;
if(p_5) 代码段 3;
if(p_5) $p_3 = t_3$;
if(p_5) $p_6 = t_3$;
if(p_4) 代码段 4;
if(p_3) 代码段 5;
if(p_6) 代码段 6;
代码段 7;

2.3 Hyperblock

通过 RK 算法，可以将无环的控制流图中的所有分支全部删除，并且转换为谓词执行。控制流图中的环对应于程序中的循环，几乎每一个程序都有循环，我们有的时候还是希望能对有环的控制流图进行处理。对于给定的控制流图，我们并不总是想要将整个流图中的所有分支删除，而是选择性的删除一部分分支。

要选择性地进行 if-conversion，就是根据需要对控制流图中的基本块进行挑选，可以在控制流图中圈出一部分基本块，通过 RK 算法对这一部分基本块

之间的分支进行消除，而保留其他部分的分支。可以把不经常被执行的基本块，或者太过长的基本块排除在外，而把经常执行又短小精悍的基本块包含在内，这样就能提高性能。如何选择基本块是策略问题，在本章不做详细讨论。

由于分支消除需要执行 RK 算法，所以对基本块的选择必须满足 RK 算法能够正确被执行的必要条件：只有一个入口。满足条件的基本块的集合称为 Hyperblock。

定义 9 (Hyperblock) *Hyperblock* 是一个谓词化的基本块的集合，这个集合只有一个入口，但是可以有多个出口。

虽然 Hyperblock 对基本块的选择提出了严格的限制，但是仍然可以通过一些变换将不满足 Hyperblock 定义的的基本块的集合转化为 Hyperblock，这样就给基本块的选取提供了较大的自由度。

将对 Hyperblock 定义的违背分为两种情况：(1) 从外部跳入，(2) 内部存在循环。对于第一种情况的处理方法是尾复制 **(tail duplication)**，对于第二种情况的处理方法是循环剥离 **(loop peeling)**。

尾复制从每一个由外部跳到内部的分支（跳转到这些块的入口处的分支排除）出发，沿着控制流图，将这些分支的所有直接或间接后继标记出来，然后将所有标记的基本块复制到外部，并将跳转到内部的分支重定向为跳转到其外部副本。如图 2所示，图中 (a) 是分支选择的结果，(b) 是尾复制以后的结果，(c) 是 if-conversion 之后的结果。

循环剥离则是将循环体以及其后继复制一份到外部，并将造成循环的跳转重定向到外部，这样循环的头一轮迭代留在内部被谓词化，而其后的几轮迭代则被移到外部执行。如图 3所示，图中 (a) 是分支选择的结果，(b) 是循环剥离以后的结果。

2.4 逆向 if-conversion

逆向 **if-conversion(reverse if-conversion**，简称 **RIC)** 由 Warter 在 1993 您引入，能够将谓词化的语句重新转换成分支语句。Warter 最初引入逆向 if-conversion 是为了利用 if-conversion 的分支消除功能实现全局调度，后来逆向 if-conversion 被用来平衡程序中的控制流跟谓词执行。

程序的分支是靠分支语句来实现的，分支语句将代码分割成若干基本块，每个基本块内部代码顺序执行，而分支语句则在基本块之间进行跳转。多条分支最终都汇集到一起是一个非常常见的事情，分支的汇集不需要任何语句的支持，多条路径最终跳转到统一基本块的时候，分支自动就汇集了。然而为了方便，引入隐式合并操作 **(implicit merge operation)** 的概念：认为在多条分支合并到的基本块中有一个隐式合并操作，该操作位于基本块中所有语句之前（就好比分支语句位于基本块中所有语句之后）。

类比控制依赖的概念，Warter 提出了反控制依赖的概念。

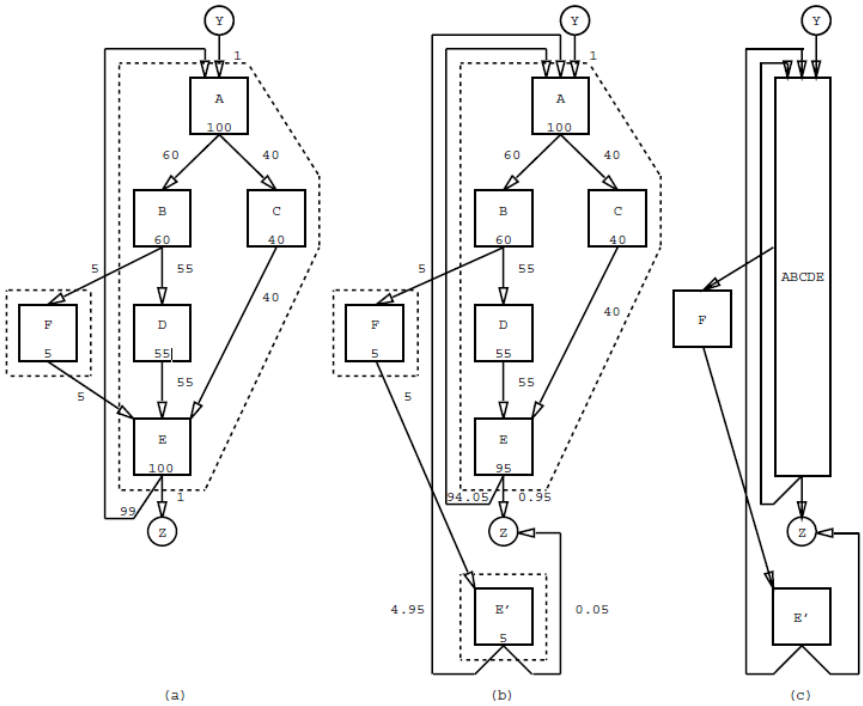

Y
1
A
100
60
40
B
60
C
40
5
55
40
F
5
D
55
55
5
E
100
99
1
Z
(a)
Y
1
A
100
60
40
B
60
C
40
5
55
40
F
5
D
55
55
5
E
95
94.05
0.95
Z
4.95
E'
5
0.05
(b)
Y
ABCDE
F
Z
E'
(c)

图 2: 尾复制

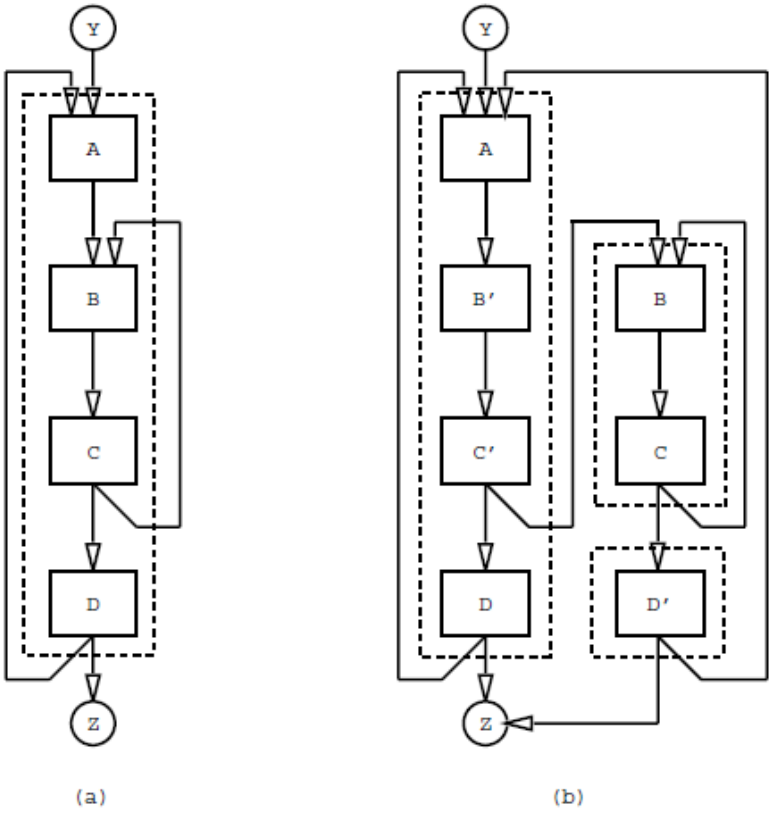


图 3: 循环剥离

定义 10 (支配 **(dominate)**) 设 V 与 W 是控制流图 G 中的节点，如果从 $START$ 到 V 的每条路径都包含 W，则称 V 被 W 支配。

定义 11 (反控制依赖 **(reverse control dependent)**) 设 G 为控制流图，X 以及 Y 是图中节点，称 X 反控制依赖于 Y 当且仅当：

1. 存在从 X 到 Y 的路径 P，使得 X 支配 P 上除了 Y 的所有节点。
2. X 不支配 Y

类似控制依赖的 $CD(x)$ 函数，可以为反控制依赖定义 $RCD(x)$ 函数。该函数的计算算法可以类比控制依赖的计算算法很容易地得到，在此不再赘述。

if-conversion 会将程序中的分支删除，并为程序中的每个语句添加谓词，分支语句则被转化成了谓词定义语句。于此类比，隐式合并操作会被转化为谓词合并操作 **(predicate merge operations)**。既然是谓词合并，就要弄清楚到底合并了哪些谓词。

分支语句分出去的分支有 true 跟 false 分支的区别，而隐式合并操作合并的分支则存在跳转 **(jump)** 与非跳转 **(non-jump)** 的区别。虽然在控制流图中，分支都是平等的，但是程序的存放是顺序存放，而指令的执行（如果没有跳转语句的话）也是顺序执行，既然是顺序存放，就要有先有后，这就导致分支汇集的地方，某些汇集来的分支（代码存放在隐式合并操作的正前方）不需要跳转语句直接顺序执行就能达到交汇点，其他的则必须经过跳转从别的地方跳转到此。

算法 5计算谓词合并操作合并的谓词，并且将这些为谓词为跳转与非跳转两类，并分别使用 $jump(t)$ 来表示这两类谓词的集合 $no\_jump(t)$。$jump(t)$ 是一个函数，它的定义域是谓词合并操作的集合，对于谓词合并操作 t，$jump(t)$ 是一个谓词的集合，组成这个集合的谓词被 t 合并，并且使用集合中谓词的基本块需要经过跳转语句到达谓词合并语句。同理，$no\_jump(t)$ 也是一个谓词的集合，使用集合中谓词的基本块要么不需要经过跳转即可顺序执行到达分支合并语句，要么不是谓词合并操作的直接前驱。

算法 5: ComputePredicatesToBeMerged

Input: 控制流图 $G(N, E, Start)$，其中 N 是 G 中节点的集合，E 是 G 中边的集合

Output: 对每个谓词合并操作 t，输出 $jump(t)$ 以及 $no\_jump(t)$

```
/* 对于基本块 X，记 P(X) 为 X 使用的谓词                                    */
for X ∈ N do
    for t ∈ RCD(X) do
        if X 不是 t 的直接后继，或者 t 在代码存放在 X 代码的正前方 then
            no_jump(t) = no_jump(t) ∪ {P(X)};
        else
            jump(t) = jump(t) ∪ {P(X)};
        end
    end
end
```

逆向 if-conversion 的算法维护两个数据结构，一个是集合 L，一个是函数 ρ。其中 L 表示的是当前所有可能的执行路径的集合，举一个例子方便理解，参见控制流图图 1，程序刚开始执行的时候，程序只有可能位于 A，此时 $L=\{A\}$，在基本块 A 执行完成以后，程序会进行分支，在编译阶段，无法获知程序会走那条分支，所以此时程序可能位于 B 或者 C，此时 $L=\{B,C\}$。函数 ρ 的被称作可行谓词集 **(allowable predicate set)**，它的定义域是 L，它将 L 中元素 X 映射为一个谓词组成的集合，X 被执行当且仅当该集合中所有谓词都为真。

算法从头开始扫描谓词化的程序，针对扫描过程中遇到的每个语句 op，分情况进行处理：

- 如果遇到的是谓词定义操作，则在 L 中所有满足 $P(op)\in\rho(X)$ 的元素 X 中插入分支语句，并且创建两个新的节点作为分支语句的 true 跟 false 分支添加到 L 中，并设置这两个节点的 ρ 函数为 $\rho(Succ_t)=\rho(X)\cup true(op)$，$\rho(Succ_f)=\rho(X)\cup false(op)$，并从 L 中移除该元素。
- 如果遇到的是谓词合并操作，则为 L 中所有满足 $[jump(op)\cup no\_jump(op)]\cap\rho(X)\neq\varnothing$ 的元素 X 计算 $\rho_{new}(X)=\rho(X)-jump(op)-no\_jump(op)$，并为所有 $jump(op)\cap\rho(X)$ 中的谓词在对应的 X 中插入相应跳转语句，然后对 $[jump(op)\cup no\_jump(op)]\cap\rho(X)$ 中的所有谓词搜索与 $\rho_{new}(X)=\rho(Y)$ 的元素 Y，如果找到，则 X 的后继就是 Y，如果没找到，则创建新的后继并且将其可行谓词集设置为 $\rho_{new}(X)$
- 如果遇到的是普通的谓词化执行语句，则只需在 L 中所有满足 $P(op)\in\rho(X)$ 的元素 X 中插入该语句即可。

逆向 if-conversion 的算法如算法 6所示

算法 6: RIC

Input: 控制流图 $G(N,E,Start)$，其中 N 是 G 中节点的集合，E 是 G 中边的集合

Output: 对每个谓词合并操作 t，输出 $jump(t)$ 以及 $no\_jump(t)$

```
/* 对于基本块 X，记 P(X) 为 X 使用的谓词                    */
for X ∈ N do
    for t ∈ RCD(X) do
        if X 不是 t 的直接后继，或者 t 在代码存放在 X 代码的正前方 then
            no_jump(t) = no_jump(t) ∪ {P(X)};
        else
            jump(t) = jump(t) ∪ {P(X)};
        end
    end
end
```

Algorithm RIC: Given hyperblock H, regenerate a correct control flow graph. For VLIW processors, insert_no-op fills each empty operation slot in an instruction with a no-op operation.

```
Let ∀ X ∈ H
    P(X) = predicate that X is control dependent upon

create root node
L = {root}
ρ(root) = {p0}
for op ∈ H in scheduled order
    for X ∈ L
        if op is predicate define operation
            if P(op) ∈ ρ(X)
                insert conditional branch operation in X
                create successor nodes Succ_t and Succ_f
                ρ(Succ_t) = ρ(X) ∪ true(op)
                ρ(Succ_f) = ρ(X) ∪ false(op)
                L = (L − X) ∪ Succ_t ∪ Succ_f
        if op is predicate merge
            for p ∈ (jump(op) ∪ no_jump(op))
                ρ_new(X) = ρ(X) − no_jump(op) − jump(op)
                if p ∈ ρ(X)
                    if p ∈ jump(op)
                        insert jump operation in X
                    for Y ∈ L
                        if ρ_new(X) ≡ ρ(Y)
                            if p ∈ jump(op)
                                Succ_t = Y
                            else
                                Succ_f = Y
                            L = L − X
                    if successor not found
                        if p ∈ jump(op)
                            create successor node Succ_t
                            ρ(Succ_t) = ρ_new(X)
                            L = (L − X) ∪ Succ_t
                        else
                            create successor node Succ_f
                            ρ(Succ_f) = ρ_new(X)
                            L = (L − X) ∪ Succ_f
        if op is predicated operation
            if P(op) ∈ ρ(X)
                insert op in X
    if target processor is VLIW and last op of instruction
        insert_no-op
```

2.5 乱序处理器

乱序处理器的发展提高了处理器执行指令的效率，但是这也给 if-conversion 提出了新的挑战，乱序处理器并不提供其他处理器那样子的谓词执行，相反，

乱序处理器提供的是 Φ 预测，Φ 不论在谓词是否为真时都会向寄存器写入数据，这就导致传统的 if-conversion 不适用。2003 年，Weihaw Chuang 等人详细讨论了此种情况下的 if-conversion 的处理 [4]。

3 if-conversion 的执行策略

3.1 分支与性能的关系

在现代的高性能处理器中，分支指令是一个代价昂贵的指令 [13]。深度流水以及多发射极大的提高了处理器的性能，但是频繁出现的分支导致的控制流转移会中断指令流水线的连续性，从而降低性能。为了解决分支语句对性能的影响，现代处理器往往配置分支预测机制，即在条件分支执行前，处理器对分支方向进行预测，并按照预测的方向进行取指执行。不幸的是，分支预测经常出错，一旦预测失败就会造成非常严重的性能损失。研究显示，错误的分支预测会使得性能降低 2-10 倍 [19] [21] [12]。

分支预测失败对性能的影响主要有三个方面。首先，处理器会在预测的分支方向取出大量指令执行，一旦分支预测失败，所有执行的结果都必须被丢弃。这就意味着，预测失败时，处理器浪费了大量的指令槽来执行无用的指令。浪费的指令槽的数目与处理器的发射宽度有关，处理器发射宽度越高，浪费的指令槽的数目也就越多。其次，分支预测失败以后，处理器必须撤销预测执行对处理器的影响，处理器必选让流水线的排空，同时还要无效化处理器缓存，以避免其内容被更新到处理器状态。再次，当发现预测失败的时候，处理器必须恢复到正确的分支进行执行，这就需要计算正确的指令地址，以及初始化正确方向的取指过程。另外，由于由于分支语句大量存在于程序中，超标量处理器必须要能够在一个周期内执行多个分支。假设指令流中包含 25% 的分支，那么一个 8 发射的超标量处理器必须具有每个周期执行至少 2 个分支的能力。一个周期执行多个分支增加了流水线的复杂度。在一个高发射率的处理器中，重复算术功能单元要比预测并执行多个分支要容易得多。

除了分支预测失败造成的处罚以外，分支语句会将程序分割为若干小的基本块，指令调度无法在基本块之间进行，从而影响性能。

为了解决造成的种种问题，现代处理器提供谓词执行的机制。谓词执行指的是指令的条件执行，它给指令增加一个布尔操作数，这个布尔操作数称为谓词。如果谓词的值为真，那么这个指令被正常地执行，而如果这个谓词的值为假，则这个指令就被当成空指令来执行。

有了处理器的支持，编译器就可以通过 if-conversion 来将分支转化为谓词执行。通过分支的移除，可以消除分支预测失败的惩罚，也可以避免让宽发射处理器在一个周期内处理多条分支从而提高了性能。

谓词执行的支持已经被非常广泛地应用在数值型程序以及非数值型程序的调度中 [18]。对于数值型的代码，使用软件流水调度对多个循环迭代进行重叠执行可以在超标量以及 VLIW 机器上获得高性能 [16] [10]。分支的移除使得

软件流水线能够进行更紧凑的调度以及更简短的代码展开。对于非数值型的程序，谓词执行使得决定树调度在深流水以及多指令发射的处理器上获得更高的性能 [8]。谓词执行允许多个控制路径上指令的并发执行，并且使指令在其依赖的分支被解析之前的执行成为可能。

虽然适当地应用 if-conversion 可以对性能进行提升，但是过多地进行 if-conversion 也会对性能造成不利的影响，并且不多地进行 if-conversion 也不一定改善性能。[18] [20]。

首先，if-conversion 会把一个区域内的所有的执行路径合并成一条较长的路径，所以，每次基本块被执行的时候，所有指令都必须被执行。如果不同的路径具有相近的频率跟大小，这个时候代码的执行是高效的。但是不同路径的大小以及频率往往是不同的，这就导致程序每次都要浪费大量的时间用来执行不频繁的分支或者长的分支，这对性能是不利的影响。同时，一些分支中的函数调用或者未解析的内存访问也会限制转换后的大基本块的优化以及调度。

其次，预测执行跟谓词执行并不能很好地符合到一起。预测执行指的是在指令真正需要被执行之前将指令执行，对于谓词指令来说，预测执行表示的是在其谓词计算之前对指令的执行。预测执行是超标量以及 VLIW 处理器中非常重要的指令级并行的来源，因为它能使得长延迟指令在调度中非常早地被初始化。

再次，由于多个基本块被合并成了一个大的基本块，他们将共享处理器资源，这就造成处理器资源的争用，这会增大分配的寄存器的数量。如果寄存器的分配对谓词不敏感，这将会对寄存器的分配产生压力 [14] [15]。

3.2 需要解决的两个问题

1997 年，August 在文章中详细讨论了处理器对谓词执行的支持给编译器带来的挑战，以及 if-conversion 的研究需要解决的问题 [2]。August 指出，if-conversion 的研究需要解决两个问题，一个是基本块的选择，如何选择基本块才能使得目标代码效率最高，另一个则是 if-conversion 执行时机的问题，在优化的早期执行 if-conversion 有利于向后面的过程暴露更多的潜在优化，而在编译的晚期进行 if-conversion 有利于的基本块的选取。

3.3 静态平衡框架

August 建议，在编译过程的早期大量应用 if-conversion 来发掘谓词执行带来的全部好处，此时形成的 Hyperblock 比目标体系能处理的大得多，然后在比较靠后的编译阶段进行部分逆向 if-conversion，根据目标机器调整每个 Hyperblock 的谓词化代码的数量，以平衡控制流跟谓词数目。整个过程的工作流程如图 4所示。

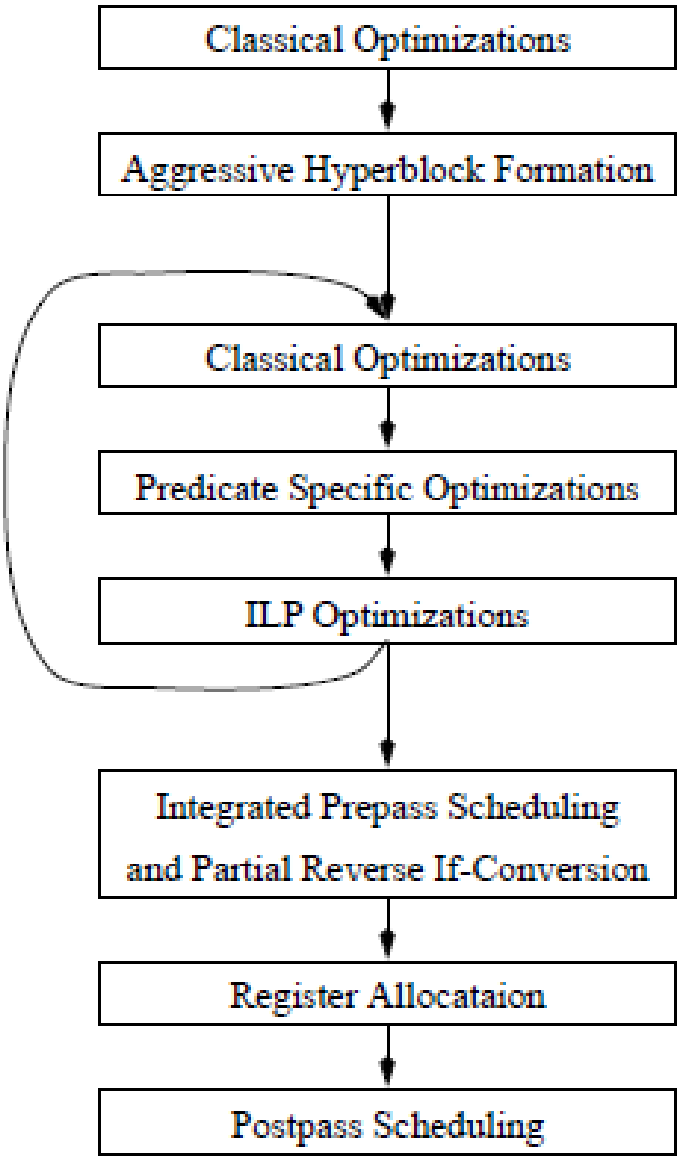


图 4: if-conversion 的执行策略

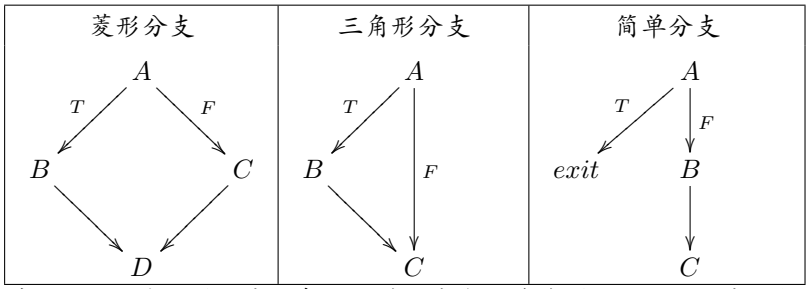
3.4 动态平衡框架

动态 if-conversion 的概念是 Hazelwood 在 2000 年提出的 [7]，这里面的“动态”的意思是说，根据程序的运行情况来实时地对代码进行 if-conversion。相比较动态 if-conversion，之前的 August 的策略叫做静态 if-conversion，因为所有的转换工作以及是否转换的决定已经在编译的时候就确定了。动态 if-conversion 的优点是，静态分析很难得出准确的分支预测失败概率，并且这些参数往往会随着程序的运行而改变，动态 if-conversion 可以随时调整程序到最佳状态以保证程序随时都能高效运行。Hazelwood 的做法是，程序运行的过程中，动态监视程序的某些运行时参数，比如说分支命中率，分支预测失败惩罚等等，当这些参数达到某个阈值的时候，即对程序进行 if-conversion，将相应的分支转换成谓词执行，当这些参数达到另一个阈值的时候，进行逆向 if-conversion 这样通过动态地对程序进行分支与谓词执行之间的转换，达到性能最优。

4 LLVM 上 if-conversion 的实现

尽管 if-conversion 的研究成果丰硕，但是不幸的是这些成果并没有在 LLVM 中得到应用。在 LLVM 3.4 中，if-conversion 的实现如下：

LLVM 中的 if-conversion 的实现非常简单，并没用所谓的 RK 算法，而是用的一个相当简单的算法。在 libCodeGen 目录中有两个文件与 if-conversion 相关，一个是 IfConversion.cpp，另一个是 EarlyIfConversion.cpp。其中，前者作用于机器码，由 AnalyzeBranch 调用，后者则作用于 SSA 格式的 IR，这是给乱序处理器 (out-of-order processor) 使用，乱序处理器的谓词执行能力比较弱。LLVM 中 if-conversion 的机制很简单，它只考虑结构化编程，则分支可以解成三种类型：简单分支，三角分支，菱形分支。三种分支的控制流图分别如下所示:



对于这三种类型的分支，都可以将分支条件作为谓词，处理起来比较简单。

5 致谢

最需要感谢的是那些为 if-conversion 做出突出贡献的科学家们，正是你们的不懈努力，我们才能有今天的高性能的编译器。其次感谢张昱老师以及张维师兄在我论文书写过程中给予的帮助，你们的帮助让我能够更好地完成我的工

作。再次感谢计算机学院的教学秘书薛佳老师，谢谢你在时间问题上一直对我的宽容与帮助。

附录

附录 A　Allen 的工作

if-conversion 最早由 Rice 大学的 Allen 等人于 1983 年提出 [1]，但是由于 Allen 的工作与后来的研究关联性不大，所以正文中对 Allen 的工作并没有叙述，现将 Allen 的工作在此附录中介绍一下。Allen 的算法直接工作在 Fortran 源代码上，它将代码中的所有分支移除，进而为接下来的向量化做铺垫。

Allen 将 Fortran 中的语句分为如下几种类型：

定义 12 (动作语句 (**action statements**)) 动作语句指的是能导致计算状态改变或者产生重要的副作用的语句，比如：赋值、读、写、函数调用。

定义 13 (分支语句 (**branch statements**)) 分支语句指的是能够显式的引起控制转移的语句，比如：*goto* 语句。注意函数调用不是分支语句而是动作语句，因为在给定的模块中，函数调用可以看成一个宏操作。

定义 14 (迭代语句 (**iterative statements**)) 迭代语句指的是能造成其他语句循环执行的语句，例如：*do* 语句。

定义 15 (占位符语句 (**placeholder statements**)) 占位符语句指的是没有任何动作，只是用来占位的语句，例如：*Fortran* 中的 *continue* 语句。

if-conversion 的目标是消除所有的分支语句，分支的消除会导致一些语句被替代为条件执行语句，这些谓词执行语句的条件被称为谓词 (guard):

定义 16 (谓词 (**guard**)) ：条件动作语句的谓词 *(guard)* 是一个布尔表达式，这个表达式表示这条语句执行的条件，当且仅当布尔表达式为真时，这条语句的动作才会执行。

Fortran 中条件动作语句形式如下：

```
1 IF(guard) statement
```

Allen 将分支分为如下 3 种类型：

定义 17 (出口分支 (**exit branch**)) 出口分支指的是能终止一个或者多个循环的分支，例如：

```
    DO 100 I=1,100
        IF (ABS(A(I)-B(I)).LE. DEL) GOTO 200
100 CONTINUE
    ...
200 CONTINUE
```

定义 18 (前向分支 (**forward branch**)) 前向分支指的是跳转目标在当前语句之后但在相同的循环中的语句，例如：

```
    DO 100 I = 1, 10
        IF (A(I).EQ.0.0) GOTO 100
        B(I) = B(I)/A(I)
100 CONTINUE
```

定义 19 (后向分支 (**backward branch**)) 后向分支指的是跳转目标在当前语句之前但在相同的循环中的语句，例如：

```
10  I = I + 1
    A(I) = A(I) + B(I)
    IF (I .LE. 100) GOTO 10
```

if-conversion 使用如下两种变换来消除程序中的分支：

1. 分支重定位 (**branch relocation**)：将分支语句一层层移出循环，直至分支语句与其目标在嵌套在相同的循环中。该过程把所有的出口分支转换为前向或者后向分支。
2. 分支移除 (**branch removal**)：将所有的前向分支移除，取而代之的是，计算每个语句的谓词，并将动作语句用条件动作语句来代替。

在分支重定位算法中，Allen 引入了出口标记 (**exit flag**) 的概念：

定义 20 (出口标记 (**exit flag**)) 出口标记是一个布尔变量，每一个出口分支都有与其对应的一个出口标记。在程序执行的过程中，该变量在其对应分支被执行之前的值始终为真，而在对应分支被执行以后其值变为假。在这里，出口分支 i 的出口标记记为 ex_i，与其对应的 *Fortran* 变量名记为 *EXi*

在上面的定义中出现了两个容易混淆的概念，出口标记 ex_i 及其对应的 Fortran 变量 EXi。这两个概念的区别是，小写字母 ex_i 是算法以及行文中出现的变量，为了更容易地写出算法，也为了行文方便，在对小写字母的变量进行操作的时候并不会考虑布尔表达式的求值效率。但是一旦要进行程序生成，布尔表达式的效率就变得尤为重要，因为复杂的布尔表达式会使生成的程序运行缓慢，于是对小写字母变量进行的操作，在真正被转化为代码的时候，必须进行布尔化简以后才能写进程序。为了区别，经过布尔化简以后的版本中的变量 (也就是最终写进程序中的变量)，用大写字母 EXi 表示。类似的表示方法在后文中还会经常出现。

在只有一个循环、一个出口分支的情况下，例如：

```
    DO 100 I = 1,100
        S1
        IF (X(I)) GOTO 200
        S2
100 CONTINUE
    S3
200 S4
```

程序中唯一的一个出口分支对应的出口标记为 ex_1。由出口标记的定义不难看出，在循环开始的时候，该变量的值应该为真，因为此时分支语句显然没有执行。此后运行过程中，该变量的值一直保持为真，直到分支被成功执行，即 X(I) 为假时，该变量才为假。由于出口分支被消除，而循环中的每个动作语句将会被替换为条件动作语句。显然，语句的控制条件就是 ex_1。

不难看出，重定位之后的代码应该为：

```
    EX1 = .TRUE.
    DO 100 I = 1,100
        IF (EX1) S1
        IF (EX1) EX1 = .NOT. X(I)
        IF (EXI) S2
100 CONTINUE
    IF (.NOT. EX1) GOTO 200
    S3
200 S4
```

不难理解，在一个循环、n 个出口分支的情况下，每个语句（包括所有新引入的对哥哥出口标记赋值的语句）的控制条件为：$ex_1 \wedge ex_2 \wedge \ldots \wedge ex_n$

这个方法也可以非常容易的拓展到多个循环、多个出口分支的情况：

```
    DO 200 I = 1,100
50      S1
        DO 100 J=1,100
            S2
            IF X(I,J) GOTO 300
            S3
            IF Y(I,J) GOTO 50
            S4
100     CONTINUE
        S5
200 CONTINUE
300 S6
```

经过分支重定向以后变为：

```
    EX1 = .TRUE.
    DO 200 I =1,100
50      IF (EX1) S1
        IF (EX1) EX2 = .TRUE.
        DO 100 J=1,100
            IF (EX1 .AND. EX2) S2
```

```
7             IF (EX1 .AND. EX2) EX1 = .NOT. X(I,J)
8             IF (EX1 .AND. EX2) S3
9             IF (EX1 .AND. EX2) EX2 = .NOT. Y(I,J)
10            IF (EX1 .AND. EX2) S4
11 100     CONTINUE
12        IF (EX1 .AND. .NOT. EX2) GOTO 50
13        IF (EX1) S5
14 200 CONTINUE
15    IF (.NOT. EX1) GOTO 300
16 300 S6
```

该算法的伪代码如算法 7所示。

算法 7: relocate_branches(x)

Input: x 是一个 DO 循环

Output: 不含出口分支的版本

/* loop_guard 表示所有出口标记的合取 */

1 $loop\_guard = true$;
2 **for** 跳出循环 x 的每个出口分支 *IF(P) GOTO S1* **do**
3 创建一个新的出口标记 ex_i，其对应的 FORTRAN 变量为 EXi;
4 在 x 前插入语句"EXi = .TRUE.";
5 $loop\_guard = loop\_guard \wedge ex_i$;
6 在循环后插入分支语句“IF (.NOT. EXi) GOTO S1”;
7 将出口分支替换为赋值语句“EXi = .NOT. P”;
8 **end**
9 **for** 循环 x 中的每个 *DO* 循环 y **do**
10 relocate_branches(y);
11 **end**
12 **for** 循环 x 中的每个非循环语句 y **do**
13 $guard(y) = guard(y) \wedge loop\_guard$;
14 **end**

前向分支是最简单的分支，它的移除称为前向分支移除 **(forward branch removal)**。为了进行前向分支移除，引入当前条件 **(current condition)** 概念：

定义 21 (当前条件 **(current condition))** 当前条件是一个逻辑表达式，它表示现在正在处理的表达式被执行的条件。

前向分支在两个地方影响控制流：

1. 分支处：在没有其他控制流改变的情况下，前向分支之后紧跟的语句只有在分支没有发生的条件下才会被执行。所以，如果分支处的当前条件为 cc_1，分支的条件 p，那么紧跟分支之后的语句的控制条件为 $cc_1 \wedge \neg p$
2. 目标处：控制流到达分支目标处有两种情况，一种是分支语句没有执行，控制流直接顺序执行到达目标处，另一种则是通过分支语句直接到达目标处。如果目标前面的那条语句的控制条件是 cc_2，则目标处的控制条件应该为 $cc_2 \vee (cc_1 \wedge p)$

引入分支标记 **(branch flags)** 的概念:

定义 22 (分支标记 **(branch flags)**) *每个前向分支 (编号为 i) 对应于一个被称为分支标记 (branch flags) 的变量 br_i, br_i 为真当且仅当对应分支语句的分支条件为真。*

例如代码:

```
    DO 100 I = 1, 100
        IF (A(I).GT.10) GOTO 60
            A(I) = A(I) + 10
            IF (B(I).GT.10) GOTO 80
                B(I) = B(1) + 10
60          A(I) = B(I) +A(I)
80  B(I) = A(I) - 5
100 CONTINUE
```

为两个分支分别引入一个布尔变量 br_1 跟 br_2, 他们对应的 Fortran 变量分别为 BR1 和 BR2。为了计算 BR1 和 BR2, 我们分别在两个分支处插入代码:

```
BR1 = A(I).GT.10
```

以及

```
BR2 = B(I).GT.10
```

由之前的描述, 不难发现示例中的各个动作语句的控制条件条件为:

行号	控制条件
3	$\neg br_1$
5	$\neg br_1 \wedge \neg br_2$
6	$br_1 \vee (\neg br_1 \wedge \neg br_2)$
7	$br_1 \vee (\neg br_1 \wedge br_2) \vee (\neg br_1 \wedge \neg br_2)$

为了防止控制条件的不断增长, 编译器必须具备布尔化简的能力, 例如第七行对应的布尔表达式恒真。我们用 $\mu(p)$ 表示逻辑表达式 p 经过布尔化简以后转换成逻辑表达式的 Fortran 语句表示。经过前向分支消除以及布尔优化后的结果为:

```
    DO 100 I = 1, 100
        BR1 = A(I).GT.10
        IF (.NOT. BR1) A(I) = A(I) + 10
        IF (.NOT. BR1) BR2 = B(I).GT.10
        IF (.NOT. BR1 .AND. .NOT. BR2) B(I) = B(I) + 10
        IF (BRI .OR. .NOT. BR2) A(I) = B(I) +A(I)
        B(I) = A(I) + 5
100 CONTINUE
```

前向分支消除的伪代码如8所示。

算法 8: $forward\_convert\,(x, cc_0)$

Input: x 是要考虑的分支，cc_0 是 x 前的控制条件

Output: 返回 x 的控制条件，输出不含前向分支的版本

/* $predicate\_list\,(x)$ 是一个队列，队列包含所有的由于分支到 x 而必须在 x 处分离的分支条件 */

$cc_1 \leftarrow cc_0$;
while $not\_empty\,(predicate\_list\,(x))$ **do**
    $p \leftarrow get\_from\_queue\,(predicate\_list\,(x))$;
    $cc_1 \leftarrow cc_1 \vee p$;
end
switch $statement\_type(x)$ **do**
    case 前向分支 *IF(P) GOTO y*
        创建一个新的分支标记 br_i，其对应的 FORTRAN 变量为 BRi;
        把 x 替换为“IF($\mu\,(cc_1)$) BRi = P”;
        $add\_to\_queue\,(predicate\_list\,(y)\,, cc_1 \wedge br_i)$;
        $cc_1 \leftarrow cc_1 \wedge \neg br_1$;
    end
    case 前向分支 *GOTO y*
        $add\_to\_queue\,(predicate\_list\,(y)\,, cc_1)$;
        $cc_1 \leftarrow false$;
        删除分支 x;
    end
    otherwise
        设置 x 的控制条件为 x;
    end
endsw
return cc_1

后向分支会产生隐式循环，因此无法被 if-conversion 消除。然而更严重的是，后向分支导致从循环外跳转到循环内成为可能，这就导致8产生不正确的代码。一种简单的做法是，保持隐式循环区内的所有代码不变，不对隐式循环区内的代码进行优化，然而这是一个非常严重的限制，因此必须考虑后向分支存在的情况下代码的优化。本节介绍完全分支移除算法9，这个算法可以在后向分支存在的前提下删除所有的前向分支。

考虑下面这个例子:

```
    IF(X) GOTO 200
    ...
100 S1
    ...
200 S2
    ...
    IF(Y) GOTO 100
```

S1 的控制条件必须反映下面事实:

1. 仅当 X 为假的时候，S1 才会在在第一轮得到执行
2. 如果是从后向分支跳转过来的，那么 S1 将永远得到执行

这暗示我们引入一个新的标记来表示 S1 的执行是否是由于后向分支跳转到此而执行的，这个标记称之为向后分支标记 **(branch back flag)**，记为 bb，其对应的 FORTRAN 变量为 BB。

这样，例子中的程序就可以化为：考虑下面这个例子:

```
    BR1 = X
    IF(.NOT. BR1) ...
    BB1 = .FALSE.
100 IF(.NOT. BR1 .OR. BR1 .AND. BB1) S1
    IF(.NOT. BR1 .OR. BR1 .AND. BB1) ...
200 S2
    ...
    IF(Y) THEN
        BB1 = .TRUE.
        GOTO 100
    ENDIF
```

可以看出，后向分支 j 的目标语句 y 被执行，可以分为两种情况:

1. 顺序执行: y 前面的语句执行完成以后，顺序执行到 y，此时 y 的控制条件就是 y 前面语句的控制条件 cc_y。
2. 后向分支: 控制流通过分支 i 进入隐式循环区，然后通过后向分支 j 跳转至此。这种情况发生的条件是 $br_i \wedge bb_j$

于是，y 的控制条件应为 $cc_y \vee (br_i \wedge bb_j)$。如果有不止一个分支能够进入隐式循环区，则将式中的 br_i 换为所有这些进入隐式循环区的分支的分支标记的析取。

考虑另外一个例子:

```
    IF(X) GOTO 200
100 S1
    GOTO 300
200 S2
    IF(Y) GOTO 100
300 S3
```

显然，正确的 S2 的控制条件应该是 $br_1 \wedge \neg bb_1$，要移除 S2 前面的分支语句，$\neg bb_1$ 必须被添加到相应语句的控制条件里面去。考虑到这一点，转换后的代码应该为:

```
    BR1 = X
    BB1 = .FALSE.
100 IF(.NOT. BR1 .OR. BR1 .AND. BB1) S1
    /*GOTO 300 已经被移除*/
200 IF(.NOT. BB1 .AND. BR1) S2
    IF(.NOT. BB1 .AND. BR1 .AND. Y) THEN
        BB1 = .TRUE.
        GOTO 100
```

```
9   ENDIF
```

完全分支移除的算法如算法 9所示。

算法 9: *remove_branches* (x, cc_0)

Input: x 是要考虑的分支，cc_0 是 x 前的控制条件

Output: 返回 x 后面语句的控制条件 cc_1，输出不含前向分支的版本

```
1  cc_1 ← cc_0;
2  while not_empty (predicate_list (x)) do
3      p ← get_from_queue (predicate_list (x));
4      cc_1 ← cc_1 ∨ p;
5  end
6  switch statement_type(x) do
7      case 前向分支 IF(P) GOTO y
8          创建一个新的分支标记 br_i，其对应的 FORTRAN 变量为 BRi;
9          把 x 替换为 "IF(μ (cc_1)) BRi = P" ;
10         process_branch (x, y, cc_1 ∧ br_i);
11         cc_1 ← cc_1 ∧ ¬br_1;
12     end
13     case 前向分支 GOTO y
14         process_branch (x, y, cc_1);
15         cc_1 ← false;
16         删除分支 x;
17     end
18     case 后向分支 IF(P) GOTO y
19         创建一个新的分支标记 bb_j，其对应的 FORTRAN 变量为 BRj;
20         在 y 之前插入 "BBj = .FALSE." ;
21         将 x 替换为代码段:
22          "TPk = μ (cc_1)"
23          "IF(TPk) TPk = P"
24          "IF(TPk) BBj = .TRUE."
25          "IF(TPK) GOTO y" ;
26     end
27     otherwise
28         guard (x) ← guard (x) ∧ cc_1;
29     end
30 endsw
31 return cc_1
```

其中的 $process\_branch$ 如10所示。

算法 10: $process\_branch\,(x, y, br)$

Input: x 是要考虑的分支,y 是分支的目标，br 是分支的条件

```
1 stmt_guard ← true;
2 for x 跳入的每个隐式循环区 do
      /* 设 bb_j 是控制这个隐式循环区的向后分支标记，x_j 是相应后向分支的目标 */
3     add_to_queue (predicate_list (x_j) , br ∧ bb_j);
4     stmt_guard ← stmt_guard ∧ ¬bb_j;
5 end
6 add_to_queue (predicate_list (y) , br ∧ stmt_guard);
```

参考

[1] John R Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189. ACM, 1983.

[2] David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A framework for balancing control flow and predication, 1997.

[3] Roger A. Bringmann, Scott A. Mahlke, Richard E. Hank, John C. Gyllenhaal, and Wen mei W. Hwu. Speculative execution exception recovery using write-back suppression. In *MICRO*, pages 214–223. ACM/IEEE, 1993.

[4] Weihaw Chuang, B. Calder, and J. Ferrante. Phi-predication for light-weight if-conversion. *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*

[5] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[6] D Harel. A linear algorithm for finding dominators in flow graphs and related problems. *Proceedings of the seventeenth annual ACM symposium on Theory of computing - STOC 1985*, 1985.

[7] Kim M. Hazelwood and Thomas Conte. A lightweight algorithm for dynamic if-conversion during dynamic optimization, 2000.

[8] Peter Y.-T. Hsu and Edward S. Davidson. Highly concurrent scalar processing. In *ISCA*, pages 386–395, 1986.

[9] Mike Schlansker Joseph C. H. Park. On predicated execution, 1991.

[10] Ardent Computer Ttt Joseph P. Brutt. Overlapped loop support in the cydra 5.

[11] Thomas Lengauer, Robert Endre, and Tar Jan. A fast algorithm for finding dominators in a flowgraph, 1979.

[12] Y. Patt M. Alsup H. Scales M.Shebanow M. Butler, T. Yeh. Single instruction stream parallelism is greater than two, 1991.

[13] Scott A. Mahlke, Richard E. Hank, Roger A. Bringmann, John C. Gyllenhaal, David M. Gallagher, and Wen mei W. Hwu. Characterizing the impact of predicated execution on branch prediction, 1994.

[14] Eduardo Quiñones. Selective predicate prediction for out-of-order processors, 2006.

[15] Eduardo Quiñones, Departament D' arquitectura De Computadors, Antonio González, and Joan manuel Parcerisa. Improving branch prediction and predicate execution in out-of-order processors, 2007.

[16] B. R. Rau, D. W. L. Yen, W. Yen, and J. P. Bratt. The cydra 5 departmental supercomputer: design philosophies, decisions, and trade-offs. *Computer*, 22(1):12–35, 1989.

[17] et al. Scott A. Mahlke. Effective compiler support for predicated execution using the hyperblock, 1992.

[18] et al. Scott A. Mahlke. Effective compiler support for predicated execution using the hyperblock, 1992.

[19] Michael D. Smith, Mike Johnson, and Mark A. Horowitz. Limits on multiple instruction issue, 1989.

[20] Zuwei Tian and Guang Sun. An if-conversion algorithm based on predication execution. *Information Technology Journal*, 9(5):984–988, May 2010.

[21] David W. Wall. Limits of instruction-level parallelism, 1991.

[22] Nancy J. Warter, Scott A. Mahlke, Wen mei W. Hwu, and B. Ramakrishna Rau. Reverse if-conversion, 1993.