if-conversion 及其在 LLVM 上的实现

高翔

qasdfgtyuiop@gmail.com

March 3, 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 [4]，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) [4], 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 分支与性能的关系

1.1 分支对性能的阻碍

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 [9]。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 [15] [17] [8]。

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行被丢弃。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。

1.2 谓词执行对性能的改善

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 [13]。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 [12] [7]。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 [5]。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。

1.3 分支消除对性能的影响

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。[13] [16]。

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 [10] [11]。

2 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 [4]。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 [6]，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，之前也分析过（见第1.3节），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 [14]。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 [18]。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。单看工作本身，对 if-conversion 并没有大的贡献，而只是一个小小的应用，本来是不足以写进 if-conversion 的发展史的，但是由于逆向 if-conversion 的提出对后面的研究有着很大的启发作用，因此这篇文章还是对 if-conversion 有着深远影响的。

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。

有了机制，又有了策略，可以说，此时 if-conversion 的研究最重要的工作已经完成，if-conversion 像是一个完成的大厦，剩下的就是修修补补了。从 1983 年 Allen 的开山之作，到 1997 年 August 为大厦添上最后一块砖，整个过程历时 14 年，5 篇意义重大的文章。

3 if-conversion 的运行机制

控制依赖的概念最早由 Ferrante 等人提出 [4]，RK 算法由 Park 跟 Schlansker 提出 [6]，Hyperblock 的概念由 Mahlke 等人给出 [14]，逆向 if-conversion 则是 Warter 等人的研究成果 [18]。

3.1 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 (后控制 (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 后控制，或者 W 后控制 V，记为 $W \operatorname{pdom} V$。

定义 3 (直接后控制 (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 直接后控制 X，或者 X 被 Y 直接后控制，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 映射为所有后控制 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 直接后控制的节点的节点，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 (后控制节点树) 直接后控制关系构成一个树，称为后控制节点树。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 的后控制节点树的算法见附录 A

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 后控制 P 上除了 X 与 Y 的所有节点。
2. X 不被 Y 后控制

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：$CD\,(x) := \{\pm y : x$控制依赖于 $\pm\, y\}$

3.2 RK 算法

3.3 Hyperblock

3.4 逆向 if-conversion

4 if-conversion 的执行策略

4.1 if-conversion 带来的挑战

4.2 August 的基于逆向 if-conversion 的平衡算法

5 LLVM 上 if-conversion 的实现

6 待办事项

1. 整理参考文献，bibtex 里面现在好多条目都没有写期刊
2. 关于分支以及谓词执行对性能的影响，不需要展开讨论，但是需要完善参考文献

附录

附录 A　后控制节点树的计算

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] 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.

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

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

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

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

[9] 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.

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

[11] 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.

[12] 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.

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

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

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

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

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

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