数据流分析的关键技术研究*)

汪小飞[1] 赵克佳[1] 田祖伟[1,2]

(国防科学技术大学计算机学院 长沙 410073)[1] (湖南省第一师范学校 长沙 410002)[2]

摘 要 数据流分析在编译优化中起着非常关键的作用,尤其是想实现一个具有技术主动权的高性能优化编译器,对数据流分析方法的研究必不可少。本文介绍了数据流分析方法的基本概念和基本原理,介绍了数据流方程的一种解决方法。并结合GCC这个具体的编译器,简要分析了其中数据流分析的具体实现方法。

关键词 数据流分析,编译优化,GCC,迭代算法

Research of Key Techniques in Data Flow Analysis

WANG Xiao-Fei[1] ZHAO Ke-Jia[1] TIAN Zhu-Wei[1,2]

(School of Computer, National University of Defence Technology, Changsha 410073)[1]

(Hunan First Normal University, Changsha 410002)[2]

Abstract In compile optimization data flow analysis is a key factor. When it comes to realizing a high performance optimizing compiler, it is necessary to research techniques of data flow analysis. In the article the basic concept and principle of data flow analysis are introduced, and a technique is discussed as well. Data flow analysis technique used in GCC (GNU Compiler Collection) is analyzed briefly in the article.

Keywords Data flow analysis, Compiling optimization, GCC, Iterative algorithm

1 引言

对程序进行各种等价的变换,使得从变换后的程序出发,能生成更有效的目标代码,我们通常称这种变换叫做优化。优化可以在程序的各个阶段进行,但最主要的一类优化是在目标代码生成以前,对语法分析后的中间代码进行的。

优化的主要目的是为了产生高效的代码,优化的技术有很多,其中很多种技术都要用到数据流分析的结果,可以很夸张地说:没有数据流分析,也就没有了编译的优化。

例如:在一个程序运行时,相当多的时间是花在循环上,所以基于循环的优化是非常重要的。对循环中的代码,我们可以进行的优化手段有:代码外提、强度削弱、删除归纳变量等等。仅以代码外提为例:所谓的代码外提就是指循环中的代码要随着循环反复的执行,但其中有些运算的结果往往是不变的,这时,我们可以将它外提到循环外。这样一来,程序运行的结果没有变,但是程序运行的速度却提高了很多,我们把这种优化手段叫做代码外提。

在我们将不变运算提到循环前置节点时,要求该不变运算所在的节点是循环所有出口节点的必经节点。这是我们就要用到活跃变量和到达一定值的概念和分析结果。这就是数据流分析要来解决的问题,通过数据流分析,我们才能知道变量在某点是不是活跃变量,这样才能进行代码外提的优化工作。不仅代码外提需要数据流分析的结果,很多的优化工作都要以数据流分析为基础。所以数据流分析对编译优化的重要性就不言而喻了。

2 数据流分析的简介

2.1 什么是数据流分析?

数据流分析是一项编译时使用的技术,它能从程序代码中收集程序的语义信息,并通过代数的方法在编译时确定变量的定义和使用。通过数据流分析,可以不必实际运行程序就能够发现程序运行时的行为,这样可以帮助大家理解程序。数据流分析被用于解决编译优化、程序验证、调试、测试、并行、向量化和并行编程环境等问题。简单地说:所谓的数据流分析,就是对程序中数据的使用、定义及其之间的依赖关系等各方面的信息进行收集的过程。

2.2 基本概念

基本块与控制流程图 进行数据流分析首先要搞清楚什么是基本块和控制流程图(Control flow),这是进行数据流分析的基础。所谓基本块就是一个连续的线性代码的语句序列,程序控制流从它的开始处进入,并从它的结尾处离开,不可能有中断或者是分支(末尾除外)。简单地说就是一个单入单出的区域。

对一个给定的程序,我们都可以把它划分为一系列的基本块。我们将每一个基本块都蜕化成一个节点,这样就产生了该程序(或该函数)的控制流程图。

只有在划分了程序的基本块和构造程序的控制流程图以后,我们才能进行数据流分析和进一步的编译优化。

数据流方程 为了优化代码,编译器需要把程序作为一个整体来收集信息,并把这些信息分配给流图的各个基本块。通过在程序的各个点建立和求解与信息有关的方程系统即可收集数据流信息。典型的方程形式如下:

$$out[S]=gen[S]\cup(in[S]-kill[S])$$

这个方程的意思是:当控制流通过一个语句(或基本块)时,在语句(或基本块)末尾得到的信息或者是在该语句(或基本块)中产生的信息,或者是进入语句(或基本块)开始点时携带的并且没有被这个语句注销的那些信息。这样的方程就叫做数据流方程。

*)基金项目:国家"863"计划软件重大专项"高性能微处理器优化编译器"(2002AA1Z2105)。**汪小飞** 硕士研究生,研究方向为高性能编译优化等。**赵克佳** 研究员,研究方向为程序设计语言及编译系统。**田祖伟** 硕士研究生,研究方向为高性能编译优化及算法分析。

利用数据流方程，我们可以得到一种解决数据流问题的重要方法。我们首先建立流图，然后同时计算每个节点(大多是以基本块为节点)的 in 和 out 集合。

3 数据流问题的分类

对于编译优化来说，涉及的数据流分析问题都是属性无关的类型，即，它们对感兴趣的每一个对象指定一个格元素，这个对象可以是变量定值，表达式计算，或其它的任何东西。

同样编译优化考虑的几乎所有问题都是一个方向的，或者是向前的(forward)，或者是向后的(backward)。双向(bi-directional)问题要求同时向前和向后传播，并且一般情况下将其公式化和求解都比单向问题要复杂。不过，在优化中双向问题非常少见。而且也已可以被更为现代的只使用单向分析的形式所替代。

与程序优化有关的最重要的数据流分析信息包括：

到达定义(readching definitions) 到达定义问题确定过程中一个变量的哪些定义可以到达该变量的每一个使用。到达定义是使用位向量的格的向前问题，其中位向量的每一位对应变量的一个定义。

可用表达式(available expressions) 可用表达式问题确定在过程的每个点上哪些表达式是可用的，在某点可用的含义是指：从入口到该点的每一条路径上存在着该表达式的一个计算，并且在此路径上的这个计算之后到该点之间，出现在该表达式中的所有变量都没有被重新赋值。可用表达式是位向量上的格的向前问题，其中位向量的每一位对应表达式的一个定义。

活跃变量(live variables) 对于给定的变量和程序中给定的点，活跃变量问题确定沿着此点到出口的路径上是否存在对该变量的使用。这是一个使用位向量的向后问题，其中变量的每一个使用在位向量中有一个位置。

向上暴露使用(upwards exposed uses) 此问题确定在特定点上哪些变量的使用可以由特定的定义而到达。这是一种使用位向量的向后问题，一个变量的每一个使用在位向量中相应地有一位。它是到达定义的对偶问题，到达定义连接定义到使用，而向上暴露使用连接使用到定义。注意这两者是有所不同的。

复写传播分析(copy-propagation analysis) 复写传播分析确定从一个复写赋值，比如说 $x \leftarrow y$，到变量 x 的使用的每一条路径上都不存在对 y 的赋值。这是一个使用位向量的向前问题，位向量的每一位表示一个赋值。

常数传播分析(constant-propagation analysis) 常数传播分析确定从对一个变量的常数赋值，比如说 $x \leftarrow const$，到 x 的使用的每一条路径上，对 x 的赋值都只是此常数值 *const*。这是一个向前问题。

部分冗余消除(partial-redundancy analysis) 部分冗余消除确定在某条执行路径上被执行了两次(或多次)，而其操作数在这些计算之间没有修改过的那些计算。

在优化中遇到的不仅仅是上面列出的这些数据流问题，但它们是其中最重要的一些问题。

4 数据流分析使用的方法

解数据流问题有下面列出的多种方法：

1. 强连通区域方法；
2. 迭代算法；
3. T1-T2 分析；
4. 结点列表算法；
5. 图形文法方法；
6. 消去法，例如，区间分析；
7. 高级语法制导的方法；
8. 结构分析；
9. 位置式(slotwise)分析。

我们主要来看一个最常用也是最实用的迭代法。

数据流信息的求解方法有两种，对于结构化的程序可采用语法制导的求解方法，对于任意控制流的程序可采用数据流方程的迭代求解方法。语法制导的方法比较简单和快速，但有其一定的局限性，所以，大部分情况下我们最多使用的方法还是数据流方程的迭代求解方法。

以到达定义的迭代算法为例，我们来看看它的具体实现算法：

我们为每个基本块 B 定义 out [B]，gen [B] ，kill [B] 和 in [B]，在这里，每个块 B 可以看作是一个或多个赋值语句的串联。假设已经计算出了每一个块的 gen 和 kill，我们可以创建两组和 in 与 out 有关的方程。

$$In[B] = \bigcup_{B\text{的前驱}P} out[P]$$

$$out[B] = gen[B] \cup (in[B] - kill[B])$$

第一组方程是基于这样的观察得来的，即 in [B]是从 B 的所有前驱到达定义的并。第二组方程则是我们的对所有语句都成立的数据流方程。如果流图有 n 个基本块，我们将得到 2n 个方程。通过循环计算 in 和 out 集合即可对这 2n 个方程进行求解。从上面的方程可以看出，作为 out 集合的并，in 集合在 out 集合为空时亦为空，因此，我们可以从空的 in 集合开始。具体的算法如下：

输入：已经算出每个块 B 的 kill [B]和 gen [B]的流图。

输出：每个块 B 的 in [B]和 out [B]。

方法：我们使用迭代法，对所有的 B，从 in [B]＝空开始，然后逐步收敛到 in 和 out 的期望值。因为我们必须重复迭代一直到 in(out)收敛，我们利用一个布尔变量 change 来记录在对块的每一遍扫描中 in 是否发生了变化。算法框架如下：

```
/* 初始化 out，假设对所有的 B，in[B]＝空 */
(1) for 每个块 B do out[B]:＝gen[B];
(2) change:＝ture; /* 使得 while 循环继续下去 */
(3) while change do begin
(4)       change:＝false;
(5)       for 每个块 B do begin
(6)             In[B]= ⋃(B的前驱P) out[P];
(7)             oldout:＝out[B];
(8)             out[B]:= gen[B] ∪(in[B]－kill[B]);
(9)              if out[B] ≠ oldout then change:＝true
          end
    end
```

直观上，算法传播定义到尽可能远的地方，只要它们没有被注销，在某种意义上说，模拟程序所有可能的执行。可以看出该算法最终总会停止，因为任何控制流图结点 N 的 out [N]的大小不会减小。一旦加入一个定值，它便永远留在这个集合中。因为所有定值集合是有限的，最终总有一遍 while 循环的结果使得第(9)行每个 N 的 oldout＝out [N]，change 仍为 false，算法终止。算法的终止是安全的，因为 out 没有改变，那么下一遍的 in 也不会改变。如果 in 不变，那么 out 也不变，所有以后的遍都不会有任何改变。

5 GCC 中的数据流分析简介

GCC 编译系统主要由三部分组成：与语言相关的前端、与语言无关的后端以及与机器相关的机器描述。在 Linux、Unix 等平台上广泛使用的编译器集合，它能够支持多种语言前端，包括 C、C++、Objective-C、Ada、Fortran、Java 等[1,2]。

如果想拥有一个具有技术主动权的高性能优化编译器。解读 GCC 源代码，深入了解它的实现技术是这个项目实现过程中必不可少并且非常重要的一环。而对其的数据流分析必不可少。

GCC 中的数据流分析主要是由 flow. c 文件来完成的。

这个文件包含着编译过程中的数据流分析，它计算数据流信息，从而决定哪些指令可以合并和分配寄存器。

它的入口函数是 life_analysis，其大部分功能都由 life_analysis 函数来完成，其它的函数绝大部分都是 life_analysis 的辅助函数，分析函数的第一步是将函数划分为基本块，find_basic_blocks 完成这个任务。接下来就是由 life_analysis 来决定哪个寄存器是活跃的，哪个寄存器已经死掉了。

find_basic_blocks 划分当前函数的 rtl 指令到每个基本块同时构建控制流程图(CFG)。基本块信息记录在 basic_block_info 队列里；而 CFG 则存在于指向基本块的边的结构里。同时，find_basic_blocks 也发现哪些不可到达的循环并且删除它们。

在调用 find_basic_blocks 结束后，life_analysis 马上就被调用，它是用基本块的信息来决定哪一个实寄存器或伪寄存器是活跃的。

关于在哪里寄存器是活跃的信息存在于两个部分中：一个是在指令的 REG_NOTES 中，一个存在于向量 basic_block→global_live_at_start 中。basic_block→global_live_at_start 为每个基本块都设有一个元素，这个元素是一个位向量，其中的每一位代表一个实寄存器或者是伪寄存器。假如在基本块的开始处寄存器是活跃的，那么这一位就置 1。

两种类型的元素能被添加到一个指令的 REG_NOTES。对于每个寄存器来说，在下列两种情况都出现的时候，一个 REG_DEAd 记录将添加到一个指令的 REG_NOTES 中：

1. 这个寄存器的值在后续的指令中都不再需要；

2. 这条指令没有重新给这个寄存器赋值(在多字的实寄存器的情况下，每一个寄存器中的值都必须被指令重置，从而来避免出现一个 REG_DEAD 的记录)。

在大部分的情况下，一个记录在 REG_DEAD 中的对象会在指令中被使用。有种很少见的特例：假如一个指令使用的是一个多字的寄存器，而且仅有一部分的寄存器在后续的指令中被使用，在这种情况下，REG_DEAD 将标记那些在后续的指令中不再需要的实寄存器。当一个指令集仅有一些而不是全部的实寄存器在这样的多字的操作数中使用时，这种类型的部分 REG_DEAD 记录并不会(在指令中)出现。忽略储存在一条指令中的对象的 REG_DEAD 记录是可以选择的，而且这样做(增加局部 REG_DEAD 记录的复杂性)的想法并不能证明是完全正确的。

REG_UNUSED 记录是为那些被指令所设置但是在随后并没有被使用的寄存器所准备的。(如果每一个被该条指令所设置的寄存器都没有被使用，而且这条指令并没有指向内存或者是有其他的 side-effect，那么这条指令要被立即删除掉)。假如一个多字的实寄存器仅有部分在后续的指令中被使用，那么 REG_UNUSED 记录将标记那些没有被使用的部分。

在指令后决定哪些寄存器是活跃的，可以从基本块的开始处开始和扫描指令。然后标记哪些指令设置了哪个寄存器，而哪个寄存器又在这里被杀死了。

life_analysis 函数有三个参数：具体的形式为：life_analysis (f, file, flags)，其中 file 是要载入分析的文件，f 是函数的第一个指令，flags 是一组被用于计算数据流信息的 PROP_* flags。它的主要功能就是执行数据流分析，同时，还完成下列功能：

• life_analysis 设立指令的 LOG_LINK 区域，这是因为这个信息将非常的有用。

• life_analysis 删除那些死指令，这些指令的唯一作用是储存了一个永远也不会被使用的变量。

• life_analysis 注意到这样一种情况，一个寄存器作为一个内存地址被应用，则可以被结合为这个寄存器的先前的或者是随后的增量或者是减量，这时，这个单独的增量或者是减量的命令将被删除，而这个地址将被改变为一个 POST_INC 或者是类似的 rtx。

• 每次当一个增量或者是减量的地址被产生时，一个 REG_INC 元素将被添加到指令的 REG_NOTES 列表。

• life_analysis 在每一个确定的向量中填写关于包含下列寄存器使用的信息：REG_N_REFS, REG_N_DEATHS, REG_N_SETS, REG_LIVE_LENGTH, REG_N_CALLS_CROSSED 和 REG_BASIC_BLOCK。

• 假如函数没有修改堆栈的指针，life_analysis 将设置 current_function_sp_is_unchanging。

flow. c 文件中的 life_analysis 所分析的寄存器的活跃情况，其实就是在利用上述的数据流分析的迭代分析方法，来计算每个块的 in 集合和 out 集合。在 life_analysis 中还用到了 flow. c 中定义的许多其它函数，篇幅关系就不一一介绍了。

结束语 目前 GCC 中数据流分析算法还不是很完善，还存在一定的不足和需要进一步研究的地方，主要包括：如何在数据流分析中尽可能多地删除无用信息，以提高数据流分析和编译优化的效率；如何根据程序的具体特性选择数据流分析的方法；如何针对具体的体系结构，充分利用其底层的特性，来提高数据流分析的速度等等。这些问题都需要进一步研究，从而改进现有的数据分析算法，提高编译优化的效率。我们通过解读 GCC 的数据流分析方法，可以利用它的优点，避免它的缺点，为能设计一个拥有具有技术主动权的高性能优化编译器打下基础。

参 考 文 献

1 Stallman R M. Using and Porting the GNU Compiler Collection for GCC 3. 1. June 2001

2 赵克佳，沈志宇. GCC 支持多平台的编译技术. 计算机工程与应用，1996

3 Pingali R J K. Dependence- Based Program Analysis. Department of Computer ScienceCornell University, Ithaca, NY 14853

4 Moonen L. A Generic Architecture for Data Flow Analysis to Support Reverse Engineering