有效的和精确的数据依赖测试

Dror E. Maydan, John L. Hennessy and Monica S. Lam

Computer Systems Laboratory

Stanford University, CA 94305

摘要

数据依赖测试是在数值类程序中检测循环级并行性的一个基本步骤。该问题等价于整数线性规划，因而一般来说无法有效的解它。当前的方法为了提高编译器的性能一般使用不精确的解法，因而牺牲了一些潜在的并行性。

本论文指出在实际情况中，数据依赖可以被有效地和精确地计算出来。主要有三个想法导致这一结果。首先，我们对一些特殊情况开发了一些有效的算法，每种情况一个算法。第二，我们引入了存储技术来缓存以前的测试结果，因而避免了对同一输入做多次依赖测试。第三，我们指出本方法可以被扩展到计算距离向量和方向向量，并且可以使用符号项而不损失精确性和有效性。我们已经在SUIF系统（一个stanford开发的通用型编译器）中实现了我们的算法。针对PERFECT Club基准测试，我们运行该算法并且在所有测试例子中都有效的得到了精确的结果。

1 介绍

针对用Fortran类的语言编写的程序的并行化的工作集中在并行化源代码中的循环。为了完成它，我们必须分析数组引用的模式。比如：

This research was supported in perr by a fellowship from AT&T Betl Laboratories and by DARPA contract NOO014-S7-K-0S28.

```
for i = 1 to 10 do
   a[i] = a[i+10]+3
end for
for i = 1 to 10 do
   a[i+1] = a[i]+3
end for
```

在第一个循环中所有的迭代可以并发的执行，因为被写入的位置和被读取的位置不重合。在第二个例子中，每一个读都应用了上一迭代写入的数据，所以必须顺序执行。数据依赖测试回答下面的问题：两个数组引用是否会在不同的迭代中引用同一个位置。既然没有整数i, i' 满足 1< i, i' <10 和 i = i'+ 10, 所以我们说第一个例子中的两个引用是独立的。由于存在整数 i, i' 满足1< i, i' <10 和 i + 1 = i', 第二个例子中的两个引用是依赖的。

理想情况下，人们总是想确切知道引用之间是否存在依赖。在存在依赖的情况下假设没有依赖会导致错误的结果。 在没有依赖的情况下假设存在依赖会阻止并行化。在最坏的情况下，精确的数据依赖的计算代价对编译器来说太高了。如下文所示, 数据依赖测试和整数规划是等价的。已知的整数规划算法要么依赖于循环边界的数值，或者是$O(n^{O(n)})$，其中n是循环变量的个数 [8] [9] [13]. 即使对于实际碰到的小问题，这些算法仍然太费时了。

针对这个问题，已经有很多算法提出来了。每一个都对准确性和效率做了折中。传统的算法试图证明不存在依赖，如果算法失败则假设存在依赖 [2][4] [18][19]. 如果这种算法返回依赖，我们不知道是否做了近似。 一些算法则保证对一些特例输入给出精确结果。简单循环余数[15] 和 Li 与Yew的工作[10] 属于这一类。很少有人分析这些算法实际应用的精确性或效率。

据我们所知，已有的算法没有哪个即足够精确又足够有效。实际上， Shen, Li 和 Yew 发现有些情况比如耦合下标经常出现而且用传统算法无法精确分析[14]。

我们的方法是使用一系列特例精确测试。如何输入不是一个算法适用的形式，我们试用下一个。使用一系列测试允许我们精确处理很大范围的输入。我们在PERFIXX Club [7]上评估我们的算法。PERFIXX Club包含13个科学基准测试。我们发现对每一个测试案例我们的算法都给出了精确的结果。

串级精确测试也比串级使用不精确测试更有效率。通过首先尝试最广泛且代价最小的测试，在很多情况下，我们可以返回确定的结果而只需要一次测试，即使是有依赖的情况。即使是其它情况，我们也仅仅需要多个测试的可用性。我们从不需要应用超过一个的测试。作为对比，串级不精确测试要求使用所有的测试直到最后找到真的依赖。如下文所示，大多数实际案例都是存在依赖的。实际上，大多数方向向量测试也是以来的。

下一步，我们转向提高我们方法的效率。我们的基础测试从不费时的到相对费时的。有人说依赖测试是在相对小的输入下执行大量的测试 [11]。我们发现实际上是在小的输入上重复的执行少量的重复测试。在实际应用程序中，数组的应用模式很少改变，并且大多数循环界限是从1到n，其中n在整个程序中都是一样的。所以我们可以通过存储（记住上次测试的结果）来减少很多的计算量。我们通过哈希方案来有效的实现它。

我们将本方案扩展到距离和方向向量。我们首先使用标准层次方法，基于Burke 和 Cytron的工作 [5]。虽然我们的算法保持精确，但是它导致计算量大幅增加。简单的剪枝方法可以将计算量大幅下降。.

最后，我们将算法扩展到处理符号项。这保持了算法的精确性，只是稍微增加了一些开销。

2 问题定义

我们首先给出一般性的数据依赖测试问题的标准定义。在一般情况下，我们允许多维数组和嵌套梯形循环。

我们将问题域局限在循环界限是外层嵌套循环变量的线性整数函数并且所有的数组引用是循环变量的线性整数函数。这些约束条件不一定要在源程序中满足。我们使用优化技术 (常数传播，归纳变量和前向替换 [16]) 来增加这些条件的可满足性.

给定下面的通用规范化循环（步长规范化到1）：

for $i_1 = L_1$ **to** U_1 **do**
  for $i_2 = L_2(i_1)$ **to** $U_2(i_1)$ **do**
    ...
    for $i_n = L_n(i_1,\ldots,i_{n-1})$ **to** $U_n(i_1,\ldots,i_{n-1})$ **do**
      $a[f_1(\vec{i})][f_2(\vec{i})]\ldots[f_m(\vec{i})] = \ldots$
      $\ldots = a[f'_1(\vec{i})][f'_2(\vec{i})]\ldots[f'_m(\vec{i})]$
    end for
  end for
end for

其中 L, U, f, f' 是已知的线性函数。数组的两个引用存在依赖当且仅当

$$\begin{aligned}&\exists \text{ integer } i_1,\ldots,i_n,i'_1,\ldots,i'_n \text{ such that}\\ &\quad f_1(\vec{i}) = f'_1(\vec{i'}),\ \ldots,\ f_m(\vec{i}) = f'_m(\vec{i'})\\ &\quad L_1 \le i_1, i'_1 \le U_1\\ &\quad \ldots\\ &\quad L_n(i_1,\ldots,i_{n-1}) \le i_n, i'_n \le U_n(i_1,\ldots,i_{n-1})\end{aligned}$$

用矩阵形式可以等价表示为

$$\exists \text{ integral } \vec{x} \quad \text{such that } A_1\vec{x} = \vec{b_1}, A_2\vec{x} \le \vec{b_2} \qquad (1)$$

其中整数矩阵A1和A2以及向量b1和b2给定。通过将（1）中的ax=b替换为ax<=b和-ax<=-b，我们可以得到等价的表示：

$$\exists \text{ integral } \vec{x} \quad \text{such that } A\vec{x} \le \vec{b} \qquad (2)$$

2.1 数据依赖和整数规划的等价性

理想情况下人们希望数据依赖测试是精确的。然后数据依赖等价于整数规划，一个被仔细研究了的问题。标准的整数规划问题是[13]

$$\text{find the max } \vec{x} \text{ such that } A\vec{x} < \vec{b},\ \vec{x} \text{ integral} \qquad (3)$$

很明显，（2）是（3）的一个特例。所以数据依赖问题可以规约到整数规划问题。另一个多项式等价版本的整数规划问题是[13]

$$\exists\ \vec{x} \text{ such that } A\vec{x} = \vec{b},\ \vec{x} \geq 0,\ \vec{x} \text{ integral} \qquad (4)$$

我们可以通过构造下面的程序将（4）规约到数据依赖问题

```
for x1 = 0 to unknown do
    ...
  for xn = 0 to unknown do
    a[A1,1x1 + ... A1,nxn]...[Am,1, x1 + ...] = ...
    ... = a[−A1,n+1x1 + ... − A1,2nxn]...[...]
  end for
end for
```

所有已知的整数规划算法在最坏情况下都太慢了。IP是NP完备的。任何已有的算法要么和变量的个数呈指数关系，要么和系数的大小呈线性关系。典型的程序只有很少的数组维数并且没有很深层次的循环嵌套，所以和变量的个数呈指数关系是可以接受的。系数的大小则有可能非常大。

常用算法的复杂性（分支定界法、隔平面法）在最坏情况下依赖于系数的大小。Lenstra [9] 和 Kannan [8] 已经开发出不依赖于系数的算法，但是在最坏情况下，Kannan的算法是 $O(n^{O(n)})$ ，其中n是变量的个数。不幸的是，这对于通用的数据依赖测试仍然是太昂贵了。所以我们认为对于任何可能构想到的情况都适用的测试算法是不可能的。然而，如我们下文所示，有效的特例算法适用于我们找到的基准测试的任何情况。

3 数据依赖测试

在本节，我们描述我们使用的单独的测试。

3.1 扩展 GCD 测试

我们使用Banerjee的扩展 GCD 测试 [4] 作为其它测试的预处理步骤。虽然它不是精确的，它允许我们将问题转化为简单和更小的形式，增加我们测试的可适用性。该测试忽略界限，判断一组方程是否有整数解。从方程(1) 我们看出它等价于：是否存在整数向量 $\vec{x}$ such that $A\vec{x} = \vec{b}$。如果该系统是不存在依赖，那么原系统也是不存在依赖。因为循环界限只是增加了额外的约束。如果它存在依赖，整个系统也许存在也许不存在依赖。在这种情况下，我们可以利用扩展GCD测试的结果变换变量从而简化原始问题。原始的GCD测试来自数论。单个方程 $a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$ 存在整数解当且仅当gcd(a_i)能整除b。它给我们一个忽略界限的单维数组的精确测试。

Banerjee 将该方法扩展到多维领域。方程 $\vec{x}A = \vec{c}$ 总是可以分解成一个幺模整数矩阵U和 echelon矩阵D (其中d_{11}>0) ，使得UA=D。该分解是高斯消除法的一种扩展。于是原方程存在整数解当且仅当存在整数向量t使得 $\vec{t}D = c$。。既然D 是echelon矩阵，这一反向替换很容易。如果这样的t不存在，那么原系统无依赖。否则，在忽略界限的情况下，原系统存在依赖。然后我们增加界限并继续。

如果t存在，那么x=tU。如果系统不是满秩的，通常情况下也不是，那么x存在一定的自由度。比如

$$\vec{t} = (1, t_1)\ U = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ then } \vec{x} = (1, t_1)$$

其中t_1，也就是x_2可以取任何整数。

Wolfe 指出对于x的边界约束可以转化为t的边界约束 [19]. 比如:

```
for i = 1 to 10 do
    a[i+10] = a[i]
end for
```

原始的依赖问题是

$$\text{integers } i, i' \text{ such that}$$
$$i + 10 = i' \text{ and}$$
$$1 \leq i, i' \leq 10$$

扩展GCD测试告诉我们(i, i')= (t_1, t_1+ 10)。将约束转换为t表示可以得到

$$1 \leq t_1 \leq 10 \text{ and}$$
$$1 \leq t_1 + 10 \leq 10$$

因为下面一些原因，这一转换非常有价值。首先，我们减少了变量的个数。一般说来，每个独立方程减少一个变量。第二，我们减少了约束的数量。在此之前，每一个循环底限和高限都产生一个约束，数组的每一个维数也产生一个等式约束。一个等式约束要表示成两个不等式约束，比如ax=b替换为ax<=b和-ax<=-b。所以我们一共得到2*L+2*d 约束 (其中L是嵌套循环的个数， d是数组的维数)。现在，所有的等式约束都表示在边界约束中。所以我们只剩下2 * L 个约束。

大多数整数规划算法的复杂性依赖于约束的个数和变量的个数。所以，一般上GCD预处理将使得别的算法运行更快。但是更重要的是，新的约束形式比原来的更简单。我们消除了等式约束，它对于下面提到的Acyclic测试是必须的。而且我们减少了约束中变量的个数。在前面的例子中，一些约束（等式的那个）含有多个变量，而转换后的所有约束都只含有一个变量。我们将会看到，我们方法中的第一个精确测试，单变量测试，只在每个约束只含有一个变量时有效。

3.2 单变量测试

Banerjee 指出如何广义GCD测试的结果最多只有一个自由变量，那么人们可以很容易精确的解出它 [4]。每一个约束仅仅是自由变量的上界或者下界。人们仅需要遍历约束，计算合适的上界和下界，并保存一个最佳的上下界。如果最后下界比上界大，那么测试的结果是不存在依赖，否则就存在依赖。Banerjee指出之一情况可以推广到任意多变量系统，只要每个约束条件最多只含有一个变量。每一个约束仅仅是某个自由变量的下界或者上界。通过遍历所有约束，记住每个变量的紧上下界。如何有任意一个变量的下界大于上界，那么系统是不存在依赖，否则就存在依赖。这一测试是知名的单循环单维精确测试[3]的超集。很明显，如果只有一层循环和一维数组，不可能存在包含多个自由变量的约束。这一测试也适用于很多常见多维案例，包括如后文所示的耦合下标例子。

```
for i1 = L1 to U1 do
  for i2 = L2 to U2 do
    a[i1][i2] = a[i1 + const1][i2 + const2]
  end for
end for

for i1 = L1 to U1 do
  for i2 = L2 to U2 do
    a[i1][i2] = a[i2 + const1][i1 + const2]
  end for
end for
```

依赖于为了演示该算法，我们详细说明下面的这个例子。

```
for i1 = 1 to 10 do
  for i2 = 1 to 10 do
    a[i1][i2] = a[i2 + 10][i1 + 9]
  end for
end for
```

依赖GCD测试将设置$i_1=t_1$，$i_1'=t_2$，$i_2=t_2+9$，$i_2'=t_1-10$。将约束表示为变量t的形式，我们得到下式：

$$1 \le t_1 \le 10$$
$$1 \le t_2 \le 10$$
$$1 \le t_2 + 9 \le 10$$
$$1 \le t_1 - 10 \le 10$$

第一个约束将t1的下限设置为1，最后一个约束将t1的上限设置为0.因为t1的下限大于上限，所以该系统不存在依赖。

本算法效率很高，它需要 $O(num\_constraints + num\_vars)$ 步操作，每步操作只需要很少的计算量。即使部分约束含有多个变量，将这一测试应用于合适的那部分约束也将减少后续算法的约束数量。

3.3 无环测试

我们开发出无环测试来处理一个约束含有多个变量的情况。我们根据一个约束的多个变量创建一个有向图。对每一个变量t_i，我们向图中增加两个节点i和-i。我们检查单个约束条件中的每一对自由变量t_i和t_j。我们首先将约束表示为$a_i t_i \le \ldots + a_j t_j$。如果$a_i$和$a_j$都大于0，那么在图中增加一条从i到j的边。然后我们将约束表示为$-a_j t_j \le \ldots - a_i t_i$，于是增加一条从-j到-i的边。如果$a_i$小于0，我们将i和-i互换即可。同样，如果$a_j$小于0，我们将j和-j互换。

对于每个变量需要两个顶点，因为我们要区分$t_i + t_j + \ldots \leq 0$和$t_i - t_j + \ldots \leq 0$。比如下面的例子，$t_1 + 2t_2 - t_3 \leq 0$。我们创建一个六节点的图。约束可以表示为$t_1 \leq -2t_2 + t_3$，所以添加一条从1到-2的边，同时添加一条从1到3的边。同样的，该约束可以表示为$2t_2 \leq -t_1 + t_3$，所以添加一条2到-1的边和一条2到3的边。此外，用t3表示约束可一样增加两条边，一条从-3到-2，一条从-3到-1。如果我们有多个约束，没每一个约束重复上面的步骤。

如果得到的图形没有环，那么我们就可以用简单的替换法精确的解原系统。如果没有环，那么存在一个节点i（假设i>0，不损失一般性），没有边进入该节点，也就是它是该图的深度优先搜索树的叶节点。根据图的构造方法，我们可以断定下面的约束是不存在的，$a_j t_j \leq a_i t_i + \ldots \text{ where } a_i \geq 0$。所以所有涉及到$t_i$的约束具有如下的形式：

$$
\begin{aligned}
&a_{i,1} t_i \leq f_1(t) \\
&a_{i,2} t_i \leq f_2(t) \\
&\quad \ldots \\
&a_{i,n} t_i \leq f_n(t)
\end{aligned}
$$

其中f是一个线性函数，包含除了t_i外的所有变量，并且$a_{i,k} > 0, 1 \leq k \leq n$。所以$t_i$只有一个方向的约束，小于其它变量的某个函数。我们将t_i的下限设置为L_i (在单变量测试阶段为t_i计算出来的下界)。也有可能$L_i = -\infty$，如果t_i不存在下界的话。如果在$t_i = L_i$ 时系统有解，那么它也是原系统的解。如果原系统在$t_i > L_i$ 时有解，那么设置$t_i = L_i$并不改变原系统的约束。所以当$t_i = L_i$ 时系统有解当且仅当原系统有解。

如果原来的节点是-i而不是i，那么所有的约束将是另一个方向，比如$a_{i,k} < 0, 1 \leq k \leq n$。此时我们将$t_i$设置为上界$U_i$。

一旦设置好t_i，我们继续原来的深度优先搜索并设置其它变量。我们继续进行下去直到碰到了矛盾，一个下界大于上界，或者没有变量剩余。如果我们消去了所有的变量而没有找到矛盾，那么系统存在依赖。否则系统不存在依赖。我们下面通过一个例子来演示该算法。

$$
\begin{aligned}
&1 \leq t_1, t_2 \leq 10 \\
&0 \leq t_3 \leq 4 \\
&t_2 \leq t_1 \\
&t_1 \leq t_3 + 4
\end{aligned}
$$

T1两侧都有约束，但是我们可以设置t_2为$L_2=1$。于是我们得到，

$$
\begin{aligned}
&1 \leq t_1 \leq 10 \\
&0 \leq t_3 \leq 4 \\
&1 \leq t_1 \\
&t_1 \leq t_3 + 4
\end{aligned}
$$

现在可以选择设置t_1或者t_3，我们设置t_1为$L_1=1$，剩下的约束为

$$
\begin{aligned}
&0 \leq t_3 \leq 4 \\
&t_3 \geq 1 - 4
\end{aligned}
$$

t_3可以设置为0到4之间的任何值。因为没有矛盾，所以原系统存在依赖。

即使图中存在环，本算法也可以解所有不在环中的变量。这为后续步骤简化了系统。

图的创建也不是必须的。我们也可以仅仅搜索只有一个方向约束的变量并设置它们。这一方法的复杂性高于创建一个深度优先搜索树，不过它更容易实现。

这一算法需要GCD预处理以消除等式约束。约束$i_1 = i_2$将会表示为$i_1 <= i_2$ 和$i_1 >= i_2$，这将在图中产生一个环。

3.4 简单回路留数测试

如果图中存在环，我们尝试简单回路留数测试。Pratt 开发了一个简单的依赖测试算法，它对约束是$t_i \leq t_j + c$形式有效 [12]。我们创建一个图，其中每个变量一个节点。对于上面的不等式，我们continue节点ti到节点tj创建一个权重为c的有向边。假设存在另外一个约束$t_j \leq t_k + d$，它意味着$t_i \leq t_k + c + d$。我们定义图中一条路径的值为其所有边的值的和。在上面的例子中，从节点i到节点k的路径的值为c + d。所以路径的值也想边一样约束它的两个端点。所以如果有值为v路径从n1到n2，那么$n_1 \leq n_2 + v$。只有一个变量的约束也是可以接受的。我们创建一个特殊的节点n0，约束$t_i \leq c$可以表示为从ti到n0的边，值为c。同样的，$t_i \geq c$可以表示为n0到ti的边，值为-c。图中的一个环表示$i - i \leq c$，也就是0<=c 。我们检查图中的每一个环。如果某个环存在负的值，那么系统不存在依赖，否则就存在依赖。

Shostak [15]首次将这一算法扩展到处理$at_i \leq bt_j + c$ 的形式，并进一步扩展到处理多个变量的形式。不幸的是，这些扩展使得算法不再精确。然而，该算法可以扩展到处理$at_i \leq at_j + c$的形式而不丧失精确性。它等价于$a(t_i - t_j) \leq c$。假设d是下于c的最大整数并且d是a的倍数。我们可以将不等式替换为 $t_i - t_j \leq d/a$，其中d/a是整数。

作为循环留数测试的例子，我们考虑下面的约束：

$$1 \leq t_1, t_2 \leq 10$$
$$0 \leq t_3 \leq 4$$
$$t_2 \leq t_1$$
$$2t_1 \leq 2t_3 - 7$$

图1是将约束转换后的结果图

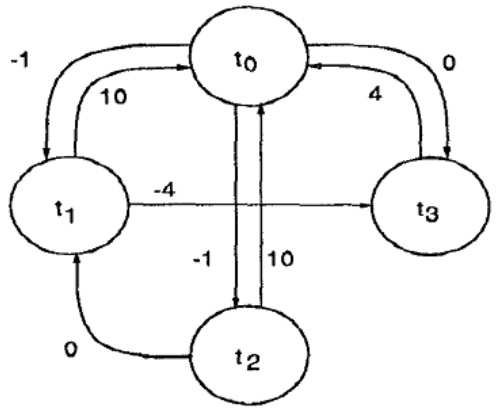

图 1: 循环留数测试的例子图

从t1到t3到t0再到 t1的循环的值是–4 + 4 – 1 = – 1。所以该系统不存在依赖。

3.5 Fourier-Motzkin

我们最后的算法， Fourier-Motzkin [6], 是一个后备的不精确测试。它精确解通用的非整数线性规划。如果返回无依赖，我们知道整数也是无依赖的。如果它返回依赖，它也会返回一个采样解。如果采样解是整数，那么整数情况下存在依赖。否则我们是不确定的。我们将这一算法扩展到处理特殊例子，其只有实数解，没有整数解。即使存在非整数采样解，我们有时也能证明不存在依赖。当我们前面四个算法不能处理的时，需要调用Fourier-Motzkin, 本算法是精确的(i.e., 它要么返回无依赖要么返回的采样解是整数)。本算法的复杂性值得讨论。理论上它是指数的。实验上，Triolet实现了该方法并发现它的效率可以满意。

[17] ，但是 Li, Yew 和 Zhu 认为Triolet的数字太昂贵的 [11]. 不管怎样，我们只需要在很少的情况下调用它，所以它的实际开销不会太高。

第一步，我们从约束条件中消除第一个变量x_1。所有的约束都规范化成x_1系数为0，1 和-1的形式。根据系数，约束集可以分为三类如下：

$$x_1 \geq D_1(\vec{x}_{2,\ldots,n}), \ldots, x_1 \geq D_p(\vec{x}_{2,\ldots,n})$$
$$x_1 \leq E_1(\vec{x}_{2,\ldots,n}), \ldots, x_1 \leq E_q(\vec{x}_{2,\ldots,n})$$
$$0 \leq F_1(\vec{x}_{2,\ldots,n}), \ldots, 0 \leq F_r(\vec{x}_{2,\ldots,n})$$

这个系统有解当且仅当

$$\exists \vec{x}_{2,\ldots,n} \text{ such that}$$
$$D_i(\vec{x}_{2,\ldots,n}) \leq E_j(\vec{x}_{2,\ldots,n}),\ i = (1,\ldots,p),\ j = (1,\ldots,q)$$
$$0 \leq F_k(\vec{x}_{2,\ldots,n}),\ k = (1,\ldots,r)$$

证明见 [6]。所以我们可以每次消除一个变量直到最后没有变量剩余。在每一步，约束个个数增长pq-p-q。最坏情况下这会导致指数增长。但p和q都很小的时候，每一步也许会减少约束的数量。如果新的系统有解$\vec{x}_{2,\ldots,n}$ ，那么原系统在x_1满足约束$\max(D(\vec{x}_{2,\ldots,n})) \leq x_1 \leq \min(E(\vec{x}_{2,\ldots,n}))$的情况下也有解。所以我们可以通过反向替换找出所有变量x的采样解。作为一个启发式策略，我们设置x_1为允许区域中间的那个整数。

在每一步i的反向替换过程中，我们已经替换了$\vec{x}_{i+1,\ldots,n}$的值，我们知道$\max(D(\vec{x}_{i+1,\ldots,n})) \leq x_i \leq \min(E(\vec{x}_{i+1,\ldots,n}))$下$x_i$存在实数解。在xi的解域中不存在整数解并不意味着原系统无整数解。如果我们为$x_k, i+1 \leq k \leq n$选择不同的采样值，我们也许能找到整数解。有一种特殊情况被证明是非常有用的。如果x_n没有整数解，那么原系统一定没有整数解。

一般情况下，如果采样解不是整数，我们可以使用分支定界法。比如说x_i= 3.5，那么我们可以两个相伴的系统。一个增加约束x_i<= 3，另外一个增加x_i>=4。如果两个中任一个存在解，那么原系统存在解。也有可能在这种情况下仍然得到非整数解。我们可以重复这一步骤。可以设想到，我们也许需要多次采用分支定界法，正比于区域的大小。

在这种情况下，我们需要在一定次数的分支定界以后截断操作，并假设系统是存在依赖的。我们还没发现有那种情况下需要明确使用分支定界法。在四种情况下，我们需要隐私的调用它（见第6节）。

4 数据依赖测试的有效性

我们已经在SUIF系统中实现了上述算法。SUIF是斯坦福大学开发的一个通用编译器。我们在其上运行PERFECT Club 基准测试。该测试有13个Fortran语言的科学计算程序，大小从500到18,000行，由Itlinois大学Urbana-Champaign校区收集[7]。我们觉得这是一组合理的测试。非可续计算代码并没有很多的循环级并行性，而且库例程一般比全程序简单。

表1显示了每个依赖测试在每个程序中各被调用了多少次。第一列代表数组常数，比如a[3]对比a[4]。这些情况下无需依赖测试。我们列出他们仅仅是为了显示它们的频率会使统计结果发生偏移，如果我们针对它们使用一般的依赖测试的话。第二列代表GCD测试返回无依赖的情况。在这种情况下，我们不需要调用后面的精确测试。其它的列代表各测试成功的应用程序数。很明显，绝大多数情况下应用的是单变量测试。

5 存储提高效率

我们已经显示了在基准测试中的每个实例我们的算法都是精确的。现在我们考虑效率。很多引用很简单因而倾向于经常性的重复。边界常常是从i= 1 到 n，其中n要么是一个常数，要么未知。虽然n可能在不同的程序中变化，它在同一个程序中常常是一样的。我们不需要在同一个数据上多次调用依赖测试程序。通过存储上次的结果，我们可以消除很多次调用。使用哈希表可以让我们快速找到重复的调用。

如果输入精确匹配，一个简单的记忆策略不会重复测试。我们可以在无用的循环索引上忽略循环界限约束以提高算法的效率。下面的两个程序：

```
(a)  for i = 1 to 10 do
       for j = 1 to 10 do
         a[i+10] = a[i]+3
       end for
     end for
```

```
(b)  for i = 1 to 10 do
       for j = 1 to 10 do
         a[j+10] = a[j]+3
       end for
     end for
```

都可以简化成下面的

```
for i = 1 to 10 do
  a[i+10] = a[i]+3
end for
```

上面的两个例子看起来有差别，现在等价了。

表2中显示了我们的简单方案以及改进方案在PERFECT Club中的效果。我们使用两个哈希表：一个使用循环界限，一个不使用。GCD测试不使用循环界限。所以，如果某个应用在忽略界限的情况下匹配，我们不需要重复GCD测试。

我们使用简单的开放表哈希策略。它对我们的目的很有效。如果需要，我们也可以找到更有效的策略。将输入数据、数组索引方程和循环界限都表示为一个长的整数向量$\vec{x}$，我们的哈希函数$\mathrm{h}(\vec{x}) = \mathrm{size}(\vec{x}) + \sum_i 2^i x_i$。选择该哈希函数是因为它对于对称或者部分对称的引用不会冲突。因为独立案例的个数较少，随机冲突不是一个大的问题。

表3显示了存储如何在表1的基础上提高。总案例列给出了从表1来的精确测试的总数。其余列显示在存储以后还有多少精确测试剩下来。存储将测试的总数从5,679降低到332！

更多的优化也是有可能的。比如我们可以消除对称的情况。假设在一个循环中有两个引用r1和r2。我们想知道它是否等价于表中的一对索引r1’和r2’，假设边界都是一样的。我们的简单策略的判断当且仅当r1=r1’和r2=r2’时，它们才是等价的。但是当r1=r2’和r2=r1’时，它们其实也是等价的。比如比较a[i]和a[i-1]其实等价于比较a[j-1]和a[j]。这可以更进一步。比如a[i][j]与a[i+1][j+1]其实等价于a[j][i]和a[j+1][i+1]。

我们另一个可能的改进是跨编译存储哈希表。这会减少增量编译的数据依赖测试的代价。

Dependence Test Frequency							
Program	#Lines	Constant	GCD	SVPC	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	229	91	613	0	0	0
CS	18,520	50	0	127	15	0	0
LG	2,327	6,961	0	73	0	0	0
LW	1,237	54	0	34	43	0	0
MT	3,785	49	0	326	0	0	0
NA	3,976	45	0	679	202	1	2
OC	2,739	2	7	36	0	0	0
SD	7,607	949	0	526	17	5	12
SM	2,759	1,004	98	264	0	0	0
SR	3,970	1,679	0	1,290	0	0	0
TF	2,020	801	6	826	0	0	0
TI	484	0	0	4	42	0	0
WS	3,884	36	182	378	4	0	160
TOTAL	59,412	11,859	384	5,176	323	6	174

表1 PERFECT Club 基准测试中每个依赖测试算法在每个程序中各被调用的次数

Percentage of Unique Cases						
Program	Without Bounds (GCD)			With Bounds		
	Total	Unique		Total	Unique	
		Simple	Improved		Simple	Improved
AP	704	7.0%	4.4%	613	6.4%	4.4%
CS	142	7.7%	7.0%	142	16.2%	14.1%
LG	73	32.9%	13.7%	73	47.9%	31.5%
LW	77	11.7%	10.4%	77	23.4%	22.1%
MT	326	3.4%	2.5%	326	6.4%	4.3%
NA	884	4.2%	3.4%	884	7.9%	6.9%
OC	43	27.9%	20.9%	36	19.4%	13.9%
SD	560	6.6%	6.1%	560	9.5%	8.8%
SM	362	5.5%	3.6%	264	4.9%	3.0%
SR	1,290	1.1%	0.9%	1,290	1.6%	1.1%
TF	832	2.2%	1.7%	826	2.9%	2.4%
TI	46	30.4%	19.6%	46	34.8%	23.9%
WS	724	11.9%	11.0%	542	14.2%	11.6%
TOT	6,063	5.7%	4.4%	5,679	7.3%	5.8%

表2 使用简单存储策略并消除无用变量以后，每个程序中单独测试的百分比

Dependence Test Frequency For Unique Cases						
Program	#Lines	Total Cases	SVPC	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	613	27	0	0	0
CS	18,520	142	14	6	0	0
LG	2,327	73	23	0	0	0
LW	1,237	77	15	2	0	0
MT	3,785	326	14	0	0	0
NA	3,976	884	48	11	1	1
OC	2,739	36	5	0	0	0
SD	7,607	560	36	6	3	4
SM	2,759	264	8	0	0	0
SR	3,970	1,290	14	0	0	0
TF	2,020	826	20	0	0	0
TI	484	46	3	8	0	0
WS	3,884	542	35	1	0	27
TOTAL	59,412	5,679	262	34	4	32

表3 单独的依赖测试的频率

此外，如果不同程序中有相似性，我们可以建立一个通用的哈希表供所有的程序使用。

6 方向和距离向量

典型的，我们不仅仅希望知道两个引用是否存在依赖[19]。如果存在依赖，我们常常想知道依赖的类型。比如：

```
for i = 1 to 10 do
   a[i+1] = a[i]+7
end for
for i = 1 to 10 do
   a[i] = a[i]+7
end for
```

在两种情况下，两个数组引用是依赖的。但是第二个循环可以并行化，而第一个不行。确切描述依赖的信息量是很大的。我们需要枚举每一个存在依赖的迭代对 (i,i')。第一种情况，但(i,i')为$(1,2),(2,3),\ldots,(10,11)$时存在依赖。对于第二种情况，当(i,i')为$(1,1),(2,2),\ldots,(10,10)$时存在依赖。区分这两种情况不需要我们使用所有的信息。这两种情况的差别是，对于第一种情况，i<i'；对于第二种情况，i=i'。方向向量是一种总结这种差别的常用方法。我们定义方向为'>'、'='和'<'三者之一。

对于有依赖的引用，如果i<i'，那么方向为'<'。我们也允许组合，比如方向'<=’代表'<'或者'='。我们使用'*'代表任意的方向。一个方向向量$\vec{\psi}$是方向$\psi_1,\psi_2,\ldots,\psi_n$组成的向量。我们说循环中的索引向量为$\vec{i}$ and $\vec{i'}$的两个引用之间是依赖的且方向向量是$\vec{\psi}$当且仅当

$$
\begin{array}{l}
i_1\ \psi_1\ i'_1 \\
i_2\ \psi_2\ i'_2 \\
\quad\cdots \\
i_n\ \psi_n\ i'_n
\end{array}
$$

对两个引用可能有多个方向向量。比如：

```
for i = 0 to 10 do
   for j = 0 to 10 do
      a[i][j] = a[2i][j]+7
   end for
end for
```

两个引用是依赖的，且方向向量包括(<,=)和(=,=)。因此，依赖测试需要返回所有存在依赖时的方向向量。

另外一个更详细地描述依赖信息的是距离向量。两个引用是依赖的并且距离为$\vec{d}$ if $\vec{i}-\vec{i'}=\vec{d}$。

```
for i = 0 to 10 do
   a[i] = a[i-3]+7
end for
```

在这个例子中，我们不仅仅知道i < i'，我们也知道i – i'= –3。扩展GCD测试提供给我们一个简单的方法来计算距离向量。在上面的例子中，GCD告诉我们i=t1和i' = t1 +3。我们简单的将两个表达式相减。既然GCD不使用边界条件，这一方法在距离仅仅依赖于边界条件时不起作用。比如

```
for i = 1 to 8 do
   for j = 1 to 10 do
      a[10i+j] = a[10(i+2)+j]+7
   end for
end for
```

我们不能发现距离向量是(2,0)。处理枚举所有可能的依赖，我们不知道有那种方法可以计算任意情况下的距离向量。即便如此，GCD在通常的恒定距离的情况下可以工作。

另一方面，距离向量可以在任何情况下计算。一个简单的方法是枚举所有可能的距离向量然后判断在此情况下引用是否存在依赖。每一个距离向量是一组循环变量的简单线性约束。我们简单的将这些约束加入到系统中并和前面一样的解它。基于Burke and Cytron [5] 的标准方法使用层次系统剪枝。不是测试所有可能的向量，它先测试(*, *,...,*)。如果返回的结果是无依赖，我们知道系统无存在依赖的方向向量。如果它返回依赖，我们于是测试下列向量 (<, *, ..., *), (=, *,.. ., *) 和 (>, *,..., *)。如果，比如说，第一个向量返回无依赖。我们知道对于第一组是<的任何向量都不存在约束。我们可以剪枝任何这样的向量。如果任何的向量返回依赖，我们继续扩展它的'*'。

增加方向向量的约束明显会限制我们的测试的可用性，比如：

$$1 \leq t_1, t_2, \leq 10$$
$$t_1 \leq t_2$$

可以用无环测试来解。如果增加方向向量约束t2 < t1，那么无环测试将不再有效，而我们必须使用循环留数测试。类似的 方向向量有时会迫使我们使用 Fourier-Motzkin测试。实践中，我们发现大量需要无环测试和循环留数测试的例子，而没有发现因为增加了约束而需要Fourier-Motzkin测试的例子。

一个更严重的问题是方向向量需要依赖测试在同一对应用上多次测试。向量的可能数量在循环内呈指数增长。即使使用层次剪枝，如果没有其它优化，在实际中我们仍然会有问题。表4中我们重复了表3并加上了方向向量，计算每一个方向测试。它过高估计了花费因为特定的固定测试比如GCD测试以及将循环边界转换为变量t的形式的开销并不依赖于每次测试有多少个向量。

即使允许过量的估计，计算方向向量仍然大大增加了测试的次数。在此之前，编译器调用了332次，大部分是SVPC测试。现在，它需要调用12500次，大部分是无环测试和循环留数测试。使用Fourier-Motzkin测试的次数从32升高到157 (注意这仅仅是因为为同一个引用检查多个向量)。

一些简单的剪枝方法能将花费降低下来。在讨论存储策略时，我们提到不需要包含未使用的变量。比如：

```
for i = 1 to 10 do
   for j = 1 to 10 do
      a[j] = a[j+1]
   end for
end for
```

既然i没有在数组表达式中出现也不再循环边界中，我们知道i – i'的方向是 "*"。所以我们对j运行测试，然后在测试的结果向量前添加一个"*"。

计算距离向量也会有帮助。比如：

```
for i = ...
   a[i+1] = a[i]
end for
```

从GCD测试我们知道i – i' = –1。所以我们知道i < i'，于是不需要i的其它方向。

表5是采用了无用变量消除和方向向量剪枝以后的结果。我们现在要调用大约900次。如果需要更好的结果，Burke 和 Cytron建议好的情况可以被看作是维到维的，而不是一个系统。比如：

Dependence Test Frequency For Direction Vectors					
Program	#Lines	SVPC	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	363	104	100	0
CS	18,520	127	48	34	0
LG	2,327	1,067	1,138	4,619	0
LW	1,237	132	73	59	0
MT	3,785	120	32	16	0
NA	3,976	295	124	172	23
OC	2,739	37	8	4	0
SD	7,607	309	106	120	28
SM	2,759	355	110	169	0
SR	3,970	130	30	18	0
TF	2,020	169	16	11	0
TI	484	780	267	703	0
WS	3,884	303	105	52	106
TOTAL	59,412	4,187	2,161	6,077	157

Table 4: Number of times each test was called only looking at unique cases and computing direction vectors

Unused Variables and Distance Vector Pruning					
Program	#Lines	SVPE	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	27	6	6	0
CS	18,520	14	16	14	0
LG	2,327	44	6	6	0
LW	1,237	15	12	5	0
MT	3,785	14	0	0	0
NA	3,976	48	59	118	7
OC	2,739	5	0	0	0
SD	7,607	54	20	55	28
SM	2,759	8	0	0	0
SR	3,970	14	0	0	0
TF	2,020	23	0	0	0
TI	484	3	38	72	0
WS	3,88 4	35	15	0	106
TOTAL	59,412	304	172	276	141

Table 5: Number of times each test was called using distance vector pruning and pruning away all unused variables

```
for i = 1 to 10 do
   a[i + 1][j] = a[i][j]
end for
```

i 和 j 是不相关的。我们可以独立计算每一个方向向量的每一维。

方向向量也引入了一个暗含的分支定界步骤。有可能在不计算方向向量时，测试返回 “unknown”，但是对每一个可能的方向向量都返回无依赖。在这种情况下，我们可以很明显的将引用设置为无依赖。这一情况在我们的测试套件中出现了四次。在每种情况下，在距离向量大于0而下于1时系统存在真的依赖。

7 讨论

我们已经展示了本算法实践中是精确的。我们没有展示同其它的非精确算法相比如何。查看引用对并不是全部。在一个有着上千个引用对的循环中，只要有一个不精确的就会大大影响整个循环的并行性。理想情况下，人们希望为并行性发现建立一个标准的模型。这样人们就可以说因为精确的依赖分析，一个程序加速了多少。不幸的是不存在这样的系统。I

即便如此，我们实现了简单的GCD测试 (算法 5.4.1 in [4]) 和 Trapezoidat Banerjee 测试 (算法 4.3.1 in [4])。没有计算方向向量，这些算法找出了482个无依赖对中的415个，遗漏了16%。对于方向向量，我们使用简单的GCD测试并接着Wolfe的扩展Banerjee矩形测试 (2.5.2 in [19])。我们消除无用变量。这样a[i] 和 a[i-1] 将会返回一个方向向量(* <) 而不是三个方向向量((< <) (= <) (> <)) ，如果这个引用在另外一个未使用的外层循环里。 该算法返回了8,314方向向量，比精确的6,828个多了22%。

我们并不相信我们特定的测试是很重要的。关键的概念是使用一组针对特例的精确测试。有可能增加其它测试并减少某些测试而不会显著更改我们的结果。

选择扩展GCD测试是因为它增加了其它测试的可适用性。选择其它几个测试有下面一些原因。它们都接受同一种数据形式 $\mathbf{A}\vec{x} \leq \vec{b}$。所以不需要在测试之间转换数据格式。某些测试需要不同形式的数据，比如lambda 测试[11] 。

实践中，所有的测试都成功的发现了无依赖。我们检查了表5中每个测试返回无依赖的次数(计算每一个方向向量) 。

SVPC在308个案例中返回了40个无依赖结果。无环测试在172个案例中返回了14个。循环留数测试在276个案例中返回了131个。Fourier-Motzkin在141个案例中返回了82个。

测试按花费排序。在一个MIPS R2000 机器(a 12 MIPS machine)上对它们计时。SVPC平均大约0.1 msec/test，无环测试大约0.5 msec/test，循环留数测试大约0.9 msec/test，而Fourier-Motzkin大约3 msec/test。

最后，在表6中我们对我们的依赖测试计时并和标准的标量优化编译器 (f77 -03)对比。我们希望展示精确测试只对编译时间增加了很少的开销。计时没有包括建立依赖测试环境的开销，比如用循环变量表示一个应用。这一建立时间，虽然可能较为显著，对所有的方法都一样。这一计时因此应当被视为我们方法所需要的时间的上限。标准编译器使用全标量优化。我们的方法在每次编译时增加了大约3% 的开销。

8 扩展到符号测试

我们的测试期望所有的引用和循环边界都是归纳变量的线性函数。我们前面提到了我们使用优化技术 (常量传播，归纳变量和前向替换) 来增加可行性。比如：

```
n = 100
i2 = 0
...
for i = 1 to 10 do
   i2 = i2 + 2
   a[i2+n] = a[i2+2n+1]+3
end for
```

将会被优化为：

```
for i = 1 to 10 do
   a[2i+100] = a[2i+201]+3
end for
```

这符合我们的分析条件。即便如此，有时未知变量不能表示为归纳变量的函数形式。比如：

```
read(n)
...
for i = 1 to 10 do
   a[i+n] = a[i+2n+1]+3
end for
```

Dependence Testing Cost		
Program	Dep. Test Cost (in secs)	f77 -O3 (in secs)
AP	2.2	151.4
CS	*	485.0
LG	4.0	65.4
LW	1.1	33.0
MT	1.0	45.0
NA	3.6	136.3
OC	0.3	38.2
SD	2.7	62.1
SM	3.5	102.5
SR	3.8	118.5
TF	2.6	116.6
TI	0.7	12.6
WS	3.6	110.0

Table 6: Total Cost of Dependence Testing. * too small to measure

只要我们知道n在循环中不变，我们可以将它加入到我们的系统中，就好像一个没有界限的归纳变量。对于这个例子，我们的系统可以问下面的问题：

does there exist integers i, i' and n such that
$1 \leq i, i' \leq 10$ and
$i + n = i' + 2 * n + 1$

表7显示了在增加符号测试以后的结果。我们的测试现在被调用了1600次，而之前是900多次。这应该只会增加很少一些开销。我们这样推测是因为我们的预优化技术非常强大。

9 结论

数据依赖测试是并行编译器的基础部件。以前的技术使用近似的方法。我们给出了该问题的一个确定性的方法。通过使用一组简单易实现的方法。串级特例精确测试，存储和更好的方向向量剪枝方法。实践中，我们的方法发现了一些以前的方法无法发现的无依赖应用。通过运行大型基准测试，我们通过实验证实了我们的方法是精确的和高效率的。

参考文献 （略）