计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.241100012

一种面向 SIMD 的控制流投机向量化方法

韩　林[1,2]　吴若枫[1]　刘浩浩[2]　聂　凯[2]　李浩然[2]　陈梦尧[2]
1 中原工学院网络空间安全学院　郑州 451191
2 国家超级计算郑州中心　郑州 450001
(674767963@qq.com)

摘　要　SIMD 自动向量化是充分发挥处理器计算能力、提升应用程序性能的重要手段,但是控制流的存在给自动向量化带来了极大的挑战。传统的控制流向量化方法依赖于 IF 转换技术,但此技术也带来了代码执行效率低的问题。因此,为了缓解这一问题,提出了一种面向 SIMD 的控制流投机向量化方法。该方法在向量代码中检测谓词相关区域,使用代价模型在区域内引导实施针对分支一致的投机变换,在运行时消除无用的谓词执行,从而消除冗余计算导致的代码效率低的问题。该方法基于当前主流的 GCC10.3 编译器实现,实验选取业界公认的 SPEC CPU 2006 测试集课题和测试向量化能力的 TSVC 测试集,结果显示 SPEC2006 测试集 481 课题在使用该方法后性能提升 10%,TSVC_2 测试部分典型用例的性能提升在 20%以上。在标准测试集上进行,结果表明,此方法能够有效提升 GCC 编译器的控制流向量化代码的执行效率。

关键词:SIMD;GCC;控制流;代价模型;投机向量化
中图分类号　TP314

Speculative Control Flow Vectorization Method for SIMD

HAN Lin[1,2],WU Ruofeng[1],LIU Haohao[2],NIE Kai[2],LI Haoran[2] and CHEN Mengyao[2]
1 College of Cyber Security,Zhongyuan University of Technology,Zhengzhou 451191,Chiha
2 National Supercomputing Center in Zhengzhou,Zhengzhou 450001,China

Abstract　SIMD automatic vectorization is an important means to give full play to the computing power of processors and improve the performance of applications,but the existence of control flow brings great challenges to automatic vectorization. The traditional control flow quantization method relies on IF transformation technology,but this technology also brings the problem of low efficiency of code execution. Therefore,in order to alleviate this problem,a speculative vectorization method of control flow for SIMD is proposed. The method detects the predicate-related region in vector code,uses the cost model to guide the implementation of the speculative transformation for branch consistency in the region,and eliminates the useless predicate execution at runtime,thus eliminating the problem of low code efficiency caused by redundant computation. The work of this method is based on the current mainstream GCC10.3 compiler. The experiment selected the industry-recognized SPEC CPU 2006 test set topic and the TSVC test set of testing vectorization ability. The results showed that the performance of SPEC2006 test set 481 topic was improved by 10% after using this method. The acceleration ratio of typical TSVC_2 test cases can reach more than 20%. Experimental results on standard test sets show that this method can effectively improve the execution efficiency of GCC compiler's control flow quantization code

Keywords　SIMD,GCC,Control flow,Cost model,Speculative vectorization

1　引言

随着科学技术的进步和多媒体产业的快速发展,人们对高性能计算的需求日益增长,因此越来越重视对数据并行性的利用。单指令多数据(Single Instruction Multiple Data,简称 SIMD)指令,即通常所说的向量化,通过对多个数据元素同时执行相同的操作,提高了处理器的计算效率。现代微处理器中 SIMD 扩展部件是广泛存在的,例如 x86 SSE/AVX,ARM NEON,POWERPC AltiVec/VMX 等[1]。随着 SIMD 技术的不断发展,向量化部件的寄存器长度也在不断扩大,如 Intel 从最初的 MMX 只支持 64 位,到后续推出的 AVX 支持 256 位,然后又推出了 512 位的 IMCI。除了支持的数据长度

基金项目:2024 河南省重大科技专项 1(241100210100);2024 河南省科技攻关项目(242102211094);2022 河南省重大科技专项 17(221100210600);2023 国家重点研发计划高性能计算专项(2023YFB3002505)
This work was supported by the 2024 Major Science and Technology Project of Henan Province 1(241100210100),2024 Henan Province Science and Technology Key Project(242102211094),2022 Major Science and Technology Project of Henan Province 17(221100210600) and 2023 National Key Research and Development Program of China-High Performance Computing Special Project(2023YFB3002505).
通信作者:聂凯(ieknie@zzu.edu.cn)

外，SIMD 支持的数据类型和操作指令也在不断地扩充丰富。基于 SIMD 扩展部件的向量化已经成为程序并行的重要手段之一[2]，已是当今科学应用获得高水平性能的关键。

当前利用 SIMD 扩展部件，实现高效并行化技术主要包括两种：1）基于程序中循环结构的 loop_based 向量化方法，即循环向量化[3]；2）针对程序中相邻语句间的 SLP 向量化方法，即超字并行技术[4]。前者是基于传统的数据依赖分析方法，发掘循环结构中迭代内语句间的并行性。后者则是通过将同构指令及其相关数据打包到超词中来实现并行化，旨在发掘程序中语句间的并行。

然而，高性能计算应用中控制流语句（如 if-statements）的存在严重影了向量化的发掘[5-6]，并且会导致生成的向量代码低效[7]。该问题产生的原因是控制流的存在，给编译过程中的数据依赖关系分析带来了诸多不确定性和复杂性[8]。最直观的表现是在对存在控制流的程序进行编译时，必须考虑其控制依赖，控制依赖指一条语句的执行取决于另一条语句的结果。

为发掘更多的向量化机会，已经有众多学者对降低控制依赖的影响进行了研究。例如，由 Allen 等提出的 IF 外提技术[9]，通过将控制流语句提至循环体外来避免控制流产生的分歧路径影响向量化优化；Liu 等提出的一种带有运行时检查的自动向量转换器技术 VecRc，通过在编译时使用运行时检查来检测动态均匀的控制流，从而在假设动态均匀性的前提下执行编译时分析[10]；Sujon 等提出的投机向量化技术，通过预测依赖分支，将预期在运行时频繁执行的代码路径进行向量化，并在预测失败时使用标量指令重新启动计算[11]；以及 Fung 等基于软硬件交互技术线程块压缩机制来实现的动态分支技术[12]。这些技术都为当前控制流向量化的发展做出了卓越贡献，并提供了十分有意义的参考价值，但是仍然存在一些问题，例如 IF 外提技术能够外提的控制流仅限于循环不变量，而动态分支技术则需要面向硬件进行优化，并且还要考虑分支同步产生的额外开销等问题，因此面对控制流的向量化难题，目前仍有很大的挖掘优化空间。

在编译器的实际应用中，目前普遍采用的向量化控制流实现方法是将控制依赖转换为数据依赖[13]，该技术被称为 IF 转换。为了能够将此方法与 SIMD 扩展部件进行适配，当前的指令集中普遍引入了向量条件选择 select[14] 以及向量掩码存储 MASK_STORE 等指令。select 指令的格式如图 1 所示，dst=select(src1, src2, mask)，指令有 3 个参数，其中 mask 为掩码，src1 和 src2 是两个源操作数。当掩码位置的值为 1 时，取 src2 的值赋给 dst，否则将 src1 的值赋给 dst。

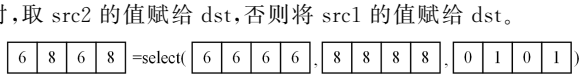

图 1 select 指令格式

Fig. 1 select instruction format

MASK_STORE 指令的格式如图 2 所示，MASK_STORE(dst, size, mask, src)指令有 4 个参数，其中 dst 是目标操作数，size 为参与运算的向量字节大小，mask 为掩码，src 为源操作数。当掩码位置的值为 1 时，将 src 的值存入 dst 的相应位置，否则保持 src 原有的值不变（如图中的 #，表示向量原有的值）。

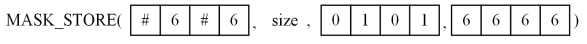

图 2 MASK_STORE 指令格式

Fig. 2 MASK_STORE instruction format

在 IF 转换的过程中，原有的分支路径被逐一剪除，那些依赖于控制的分支语句，在指令集架构中被诸如上文中所展示的这些支持掩码的指令所保护，以此来避免非法计算。通过这种技术，目前已经能够将部分具有控制流的循环进行自动向量化，但是它仍然存在两个十分重要的缺陷。一是软件预测需要额外的开销[15]，二是如果 IF 引导的分支条件在运行的过程中出现分支偏一化现象，那么大多数的 SIMD 通道都将处于空闲状态，从而导致冗余计算[16]。

为了解决上述问题，本文提出了一种面向 SIMD 的控制流投机向量化方法。该方法选择对控制流的一条“核心”路径进行投机，当程序在运行过程中出现多次的空分支一致现象，对其进行的“投机”操作，可以直接跳过一些不活跃的 SIMD 通道，从而达到降低冗余计算的目的，进而提高 SIMD 程序的执行效率。

本文第 2 章介绍了现有的控制流向量化技术；第 3 章展示了本文提出的面向 SIMD 的控制流投机向量化方法（SPvec）；第 4 章给出了实验数据和结果分析；最后总结全文。

2 控制流向量化技术

2.1 IF 转换

IF 转换的目的是将程序段内的控制依赖转换为数据依赖，这是一个整合程序内控制流分支的过程。但整合分支并不是简单的删除操作，为了确保程序能够正确执行，IF 转换引入了控制执行的概念。经过 IF 转换后，程序内原有的标量代码会被谓词化，即每一个语句都隐式地包含一个逻辑表达式来控制其执行。一条语句的逻辑表达式如果为假，则表示这条语句在此次迭代内无需执行；反之，其才能参与到此次迭代运算中。从结构上看，经过 IF 转换的代码已不存在控制分支，可以使用经典的自动向量化方法发掘数据级并行性。

当携带控制语句的循环代码因为存在依赖等原因而被拒绝向量化时，为了减少执行判断条件的次数，编译器采用了 IF 重构算法来对代码结构进行优化。IF 重构就是将那些拥有相同控制条件的语句合并到一个分支中，执行完此次优化后，这部分被控制执行但未被向量化的代码将被用一个最小的分支集合来代替，以此来规避 IF 转化后，那些未被向量化的循环所带来的性能降低风险。

通过使用条件选择和掩码指令，IF 转换在线性化控制流的同时，可以有效处理控制流向量化中可能出现的分支分歧，保证向量执行的正确性。与分支分歧相对应的是分支一致，它的存在为控制流向量化提供了新的思路。

2.2 投机向量化

携带控制流的程序在自动向量化的过程中之所以困难重重，是因为控制流语句带来的分支一致性问题。在 SIMD 向量化的过程中，关于分支一致性的分析引出了两个分类：分支分歧和分支一致。在程序中，习惯性把控制分支走向的控制条件结果称为“谓词”。在源程序向量化后，这些谓词根据程序中获取的向量类型来组成一组谓词向量。分支分歧指一个

谓词向量里存储的谓词结果是真假混合的，而分支一致指谓词向量中存储的结果是一致的，即“all false”或者“all true”。后者的出现，是能够使用投机向量化进行优化的基础。

传统的控制流向量化方法将代码中的控制分支统一成一条路径，然后挖掘向量化机会，即对控制流所在的整体进行向量化。投机向量化不改变程序原有的控制流结构，而是选取能够产生收益的高频路径进行向量化，其他路径按原有的标量形式进行处理，这便是投机向量化的“投机”所在。

```
for(i=1;i<=1024;i++)
{
  a=A[i] * scal; /* vectorizable */
  if(a <=MaxVal)
    B[i]=A[i]; /* vectorizable */
  else
    B[i]=B[i-1]; /* not vectorizable */
}
```

(a)源程序

```
for(i=1;i<=1024;i+=4)
{
Va[0:3]=A[i:i+3] * [scal,scal,scal,scal];
if(Va[0:3] <= [MaxVal,MaxVal,MaxVal,MaxVal]
  B[i:i+3]=A[i:i+3];
else
  {
    for(j=0;j<4;j++)
    {
      a =A[i+j] * scal;
      if(a <=MaxVal)
        B[i+j]=A[i+j];
    else
        B[i+j]=B[i-1+j];
    }
  }
}
```

(b)投机向量化优化后的程序

图 3　部分控制流投机向量化

Fig. 3　Part of the control flow is speculative vectorization

在实际的应用场景中，如果程序在 IF 转换的过程中无法实现对控制流的统一，即 IF 转换方法失效，那么便可以尝试开启部分控制流投机向量化[17]，如图 3 所示。而如果程序可以进行 IF 转换，则应当大力发掘程序中的分支一致机会，从而提升基于 IF 转换的控制流向量化收益。本文提出的控制流向量化方法着眼于后者，利用投机向量化的“投机”思想，关注其中某一条分支机会所带来的“投机收益”。其设计理念、具体实例以及如何实现将在第 3 章做详细说明。

3　基于 IF 转换的控制流投机向量化

3.1　SPvec 概述

基于 IF 转换的投机向量化方法 SPvec，是在现有 IF 转换技术之上，做出的一次向量代码优化。图 4(a)给出了一段携带控制流的循环代码。经过 IF 转换后，其原本存在控制依赖的标量语句被谓词化，完成了对代码中控制分支的整合，可以进一步发掘数据级并行性。指令向量化后的中间表示如图 4(b)所示。原有的标量算术操作被转换为向量操作。当涉及到对内存的写入时，条件执行的语义被映射为掩码存储指令 MASK_STORE。

```
for(i=0;i<n;i++)
if(c[i])
  {
p1[i] += 1;
p2[i]=p3[i] +2;
  }
```

(a)具有控制流的代码示例

```
header:
  vp1 =p1[i:i+3]
  vp3 =p3[i:i+3]
v_c=c[i:i+3]
mask=v_c ! = {0,0,0,0}
  v_1=vp1+{1,1,1,1}
  MASK_STORE(vp1[i],size,mask,v_1)
  v_2=vp3+{2,2,2,2}
  MASK_STORE(v_p2[i],size,mask,v_2)
i=i+4
latch:
if(i<n) goto header
exit:
  ...
```

(b)示例代码经过 ifcvt 优化遍后的中间表示

图 4　ifcvt 遍编译过程

Fig. 4　ifcvt pass compilation process

诸如掩码指令这种相对耗时的操作对代码性能的影响是巨大的。在向量执行的过程中，如果向量通道的一致性表现出较高占比，这意味着将产生大量无用的条件执行，从而严重降低程序的计算效率。对程序中存在的分支一致进行投机可以有效解决这一问题。以图 4(a)中的代码为例，其隐藏的 else 分支为空，这意味着向量通道对应的谓词为假时，程序不必做任何事，可以直接进入下一次循环。此时可以对分支条件为假的路径进行投机。在向量执行的过程中，如果投机成功，程序便能跳过掩码操作相关的指令流。这无疑会在一定程度上对代码的执行性能带来提升。

优化后的向量代码如图 5 所示。在示例中，对假分支进行投机，因此引入了对 mask 向量的“all false”判断作为投机控制条件。然后通过添加控制语句的方式，对现有的向量代码进行分区重组，使得那些原先被消除控制依赖的语句，被构建在一层崭新的控制依赖之中。由于这条新加入的控制依赖，使得程序中的部分代码由原来的必然执行，变成了现在的可能执行，即只有当投机失败的情况下代码才需要完整的执行。如此一来，在程序的执行过程中，如果“all false”的出现次数在所有执行数据中占比超过一定比例，即投机成功的次数超过一定比例，便能实现计算性能的提升。

SPvec 方法为程序新添加的投机操作，必然会对程序的计算性能产生一定的消耗，如果一个程序在执行过程中没有出现被投机成功的情况或者被投机成功产生的收益不足以抵消因为添加投机操作而带来的消耗，便会产生负优化的效果。为了避免这一情况，需要为 SPvec 专门设计一套代价模型，用于决定是否开启此优化方法。

```
header′:
v_c=c[i:i+3]
mask=v_c != {0,0,0,0}
if(mask == {0,0,0,0}) goto iter
pred:
  vp1 =p1[i:i+3]
  vp3 =p3[i:i+3]
  v_1=vp1+{1,1,1,1}
  MASK_STORE(vp1[i],size,mask,v_1)
  v_2=vp3+{2,2,2,2}
  MASK_STORE(v_p2[i],size,mask,v_2)
iter:
i=i+4
latch:
if(i < n) goto header
exit:
  …
```

图 5 SPvec 方法优化后的代码结构

Fig. 5 Code structure optimized by SPvec method

3.2 SPvec 编译框架

SPvec 向量化编译框架如图 6 所示。对于含有控制依赖的程序，其向量化处理流程大致可以分为 4 个阶段：预优化、向量化分析、向量代码生成以及向量代码调优。在预优化阶段，其主要工作便是排除那些不能向量化的循环形式，然后对含有控制流的循环进行优化。通用的处理流程是先进行循环形式检查，分析代码间的数据依赖，然后对满足条件的控制流区域进行谓词化处理，生成条件选择或者掩码指令[18]。

经过上述的预处理和优化后，程序进入到向量化分析阶段，这一阶段的主要工作便是依托循环向量化技术，检测程序中的数据级并行性。根据分析结果，可向量化的部分进入向量代码生成阶段，不可向量化的部分通过 IF 重构还原为控制分支的形式。向量代码生成便是将已经谓词化的标量指令转换成向量指令。

最后便是向量代码调优部分，SPvec 方法将应用于该阶段，对那些经过收益分析后有利可图的条件执行区域，选择合适的控制流路径生成投机代码，以减少冗余计算，提升向量程序的执行效率。

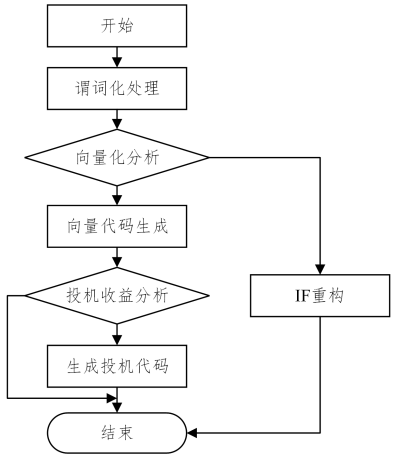

图 6 基于 IF 转换的投机向量化框架

Fig. 6 Speculative vectorization framework based on IF transformation

3.3 SPvec 算法设计

本小节将详细解释 SPvec 算法的具体实施过程。SPvec 是针对向量循环的优化变换。在使用 IF 转换线性化控制流并向量化循环之后，循环体中可能存在来自多个分支的谓词指令，它们具有不同的执行谓词。根据执行谓词的不同，循环体中的谓词执行区域可以被分为若干片，片内的向量指令属于同一分支。在应用投机变换时，可以逐片地为其重建控制流图并生成代码。

在进行变换之前，首先对代码中可能实施的投机行为进行分析。如算法 1 所示，栈列表 worklist 用于收集循环中所有的向量谓词操作。从 worklist 中选择一条语句作为投机候选，对其所在的谓词执行区域进行收益分析。如果代价模型判断对该区域进行投机是有收益的，则进入变换阶段。

算法 1 投机优化算法

```
输入：向量循环 loop
输出：投机变换后的 loop
1. worklist ← Ø
//遍历所有语句
2. for each basicblock bb ∈ loop
3.   for each statement stmt ∈ bb
     //将所有的掩码操作压入工作栈
4.     if(stmt is select or mask type) then
5.       worklist.add(stmt)
6. if(worklist=Ø) return
7. while(worklist ≠ Ø)
8.     last ← worklist.pop()
       //执行代价模型，决定是否开启投机优化
9.     if(speculation is profitable)
10.        Build Speculation
11. end while
```

图 7(a)、图 7(b)给出了变换前后的循环控制流图，对应的中间表示分别为图 4(a)和图 4(b)。从图 7(a)到图 7(b)的转化过程中，首先是对循环头 header 进行分裂。经过分裂，header 被分割成了两部分，分别是图 7(b)中的 header′和 iter。iter 继承了 header 的全部后继关系，如果投机成功将直接跳转到 iter。接着，创建了谓词操作块 pred，接入到投机失败的分支路径，其直接后继为 iter。

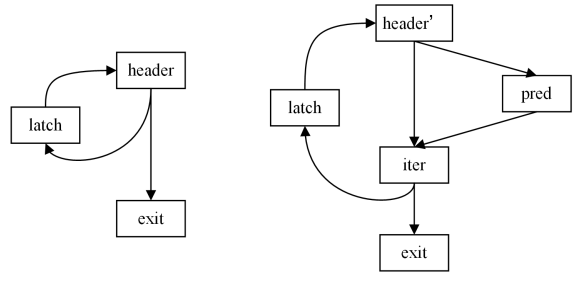

图 7 SPvec 优化前后的循环结构

Fig. 7 Loop structure before and after SPvec optimization

SPvec 的目的便是为了能够跳过部分掩码相关的操作，以减少不必要的向量化通道开启，降低冗余计算，因此 header 中的最后一条掩码操作语句便是分裂节点所在。此操作之后的语句，皆不受源程序中存在的控制依赖影响。因此，这一改

变不会影响程序最终的运行结果。

构建控制流图的具体过程如算法 2 所示。根据入栈顺序获取当前循环头中的最后一条掩码语句 last,将循环头以 last 为节点,分裂 last 语句之前(含 last 语句)的区域为 header′,之后则为 iter(步骤 3)。接着,创建谓词执行块 pred,并通过构建边(header′,pred)和(pred,iter),将其接入投机失败的执行路径。最后,依据取得的向量掩码 mask 的数据类型,创建具备相同数据类型的零向量和投机条件语句(步骤 8),并确保该投机语句位于 header′的末尾(步骤 9)。

算法 2　投机 CFG 建立算法(Build Speculation)

输入:向量循环 loop 的 CFG

输出:投机优化后 loop 的 CFG

```
Build Speculation:
1. mask ← get_mask(last)//获取掩码
2. header←basicblock(last)//获取 last 语句所在块
3. split_block(header,last)//于 last 处分割块
4. pred ← creat_empty_bb()//创建 pred 块
5. make_edge(header′,pred)
6. make_edge(pred,iter)//将 pred 块添加至循环
7. zero ← {0,0,0,…}//创建一个零向量
   //创建投机语句
8. cstmt ← build_cond_stmt(mask,zero)
   //于 header 块末尾添加投机语句
9. add_to_bb(cstmt,header)
10. Fill pred
```

经历上述操作后,已经成功将投机引入到了循环的控制流图中。接下来需要对中间表示进行处理,来完成最终的变换,主要由算法 3 Fill pred 来实施。该算法主要实现两个内容:一是对多余语句的剔除,二是对掩码相关向量语句的分流填充。关于前者,被删除的语句主要是那些不活跃的标量语句,而后者才是该算法关注的重点。这一分流填充的目的是为了将那些在逻辑上受掩码控制的语句集中到一个块中,以期投机操作在条件合适的情况下,能够跳过这些“非必要”语句。在实现语句分流的过程中,对一条语句能否加入到 pred 块,需要进行层层判断,其中最重要的便是对当前语句所定义变量的立即使用进行判断(步骤 10),因为如果该立即使用所在的语句在 pred 块之外,便表示该变量有了被二次修改的可能,这会导致程序的最终结果出错。而在此基础上,因为算法 2 新加入了一条投机语句,并且能够明确的是这条投机语句并不涉及对存储值的修改,因此在执行上述立即使用的冲突退出判断时应该提前剔除这一特殊情况(步骤 6)。最后,还需要设置该填充算法的退出条件,即当检测到控制流语句的第一个变量定义时,代表已经完成本次所有语句的分流填充操作(步骤 8)。

算法 3　语句分流填充算法(Fill pred)

输入:添加空 pred 块的向量循环 loop

输出:pred 块填充过后的向量循环 loop

```
1. while(true)
2.    last_mask ← last//将 last 语句的值赋给 last_mask
      //将语句 last_mask 移动到 pred 块块头
3.    stmt_move_bb_before(last_mask,pred)
      //遍历 last 语句之前的语句
4.    for each stmt1 before last do
5.       lhs=get_lhs(stmt1)//获取当前语句的左操作数
         //判断当前语句是否为掩码定义语句
6.       if(lhs=mask)
7.       maskdef ← rhs1(stmt1);continue
         //判断当前语句是否为控制变量定义语句
8.       if(lhs=maskdef) break
9.       res ← true
         //遍历当前语句左操作数的立即使用
10.       for each use_stmt of lhs do
          //判断当前的使用语句是否位于 pred 块中
11.          if(basicblock(use_stmt)) ≠ pred)
12.          res ← false;break
13.       if(!res) break
14.       stmt_move_bb_before(stmt1,pred)
15.   end for
      //判断当前工作栈是否为空以及当前语句是否为同一掩码控制下的语句
16.   if((worklist=Ø)
      ‖(get_mask(worklist.last()) ≠ mask))
17.      break
18.   last ← worklist.pop()
19. end while
```

综上,即为 SPvec 算法的全部构建过程。其中算法 1 为算法主体,执行顺序为算法 1 调用算法 2,算法 2 调用算法 3。其中算法 3 的时间复杂度为 $O(n^2)$,由于 3 种算法逐层嵌套,因此算法 2 的时间复杂度继承算法 3 为 $O(n^2)$,算法 1 在一个单层循环内调用算法 2,其时间复杂度为 $O(n^3)$。

3.4　代价模型

通过 3.3 节对 SPvec 方法的描述,可以认识到,当插入的投机条件判断成功时,即超字条件内的所有值为假时(为描述方便,本节后文中以 v_sc 代指超字条件),可以避免向量掩码指令的执行,但是只要 v_sc 出现一个真值,即代表投机失败,此时将不能跳过向量掩码指令实现优化。如果不考虑 v_sc 的真值比例而盲目投机,可能无法提高生成的向量代码执行效率,并且还会导致其执行效率的低效。因此,需要生成一个针对 SPvec 的代价模型,来指导是否开启该优化方法。

因为 SPvec 的收益与 v_sc 的值中“all false”的占比有关,所以需要收集 v_sc 值为“all false”时的次数和执行 v_sc 的总次数。通过插桩的方法可以获得上述两个数据。此外,还需要获取那些受源程序控制依赖影响的语句,即在 SPvec 优化后可跳过执行的语句数目 NBI,NBI 的值可在编译阶段确定。通过 NBI 的值,可以实现在向量化的过程中统计这些语句向量化后的开销总和,记为 $Cost_{NBI}$。同时,根据向量化时调用的数据类型,可以获得插入的投机语句开销,即一条相同数据类型的超字条件分支指令的开销,记为 $Cost_{SP}$。本文将获得的 v_sc 执行的总次数设为 N,v_sc 的值在执行时为“all false”的次数记为 T_{AF}。此时,可以构建如式(1)所示的一个代价评估模型。

$$FLAG\_SPvec=\frac{Cost_{SP}*T_{AF}+(Cost_{SP}+Cost_{NBI})*(N-T_{AF})}{Cost_{NBI}*N}<1 \tag{1}$$

其中，$Cost_{NBI} * N$ 表示不开启 SPvec 方法时，源程序经过所有循环迭代后，因控制流的存在所产生的不同分支内的所有语句开销总和，而 $Cost_{SP} * T_{AF} + (Cost_{SP} + Cost_{NBI}) * (N - T_{AF})$ 则表示开启 SPvec 方法后，该部分语句经过优化后的开销情况，用后者对比前者，若有利可图，即后者比前者小于 1 时，将 FLAG_SPvec 的值设为真，表示可以开启 SPvec 优化。如此，在该代价模型的指导下，可以尽可能地保障 SPvec 方法在程序编译过程中带来有利的优化。

4 实验与分析

本文的工作在开源编译器 GCC 10.3.0 中实现。编译环境为 Linux 操作系统，版本为 CentOS 7。处理器为 Intel(R) Xeon(R) Silver 4214R，核心数为 12，主频为 2.4 GHz，缓存为 16.5 MB，支持指令集 SSE4.2，AVX，AVX2，AVX-512，允许进行 SIMD 操作。

为了验证 SPvec 方法的有效性，从标准测试用例中挑选了具有代表性典型控制流程的程序作为测试对象，如表 1 所列，包括 SPEC CPU2006 测试集中的 481.wrf 和 TSVC_2 测试集中的 s1161，s253，s272，s273，s278，s279，s1279。

表 1 测试对象

Table 1 Test object

题目	所属测试集
481.wrf	SPEC CPU2006
s1161	TSVC_2
s253	TSVC_2
s272	TSVC_2
s273	TSVC_2
s278	TSVC_2
s279	TSVC_2
s1279	TSVC_2

这些程序中都存在因 IF 控制流引起的控制依赖，并且在向量化后的执行过程中，分支一致出现的情况较为频繁。

通过查看代码的编译过程，可以确保测试对象是否被 SPvec 优化。实验分别对不调用 SPvec 优化算法和调用 SPvec 优化算法两种情况下生成的可执行程序进行测试。编译时选项分别为“-O3-ftree-vectorize-fopt-info-mavx2”和“-O3-ftree-vectorize-fopt-info-mavx2-moptimize-mask-stores”。以未开启 SPvec 方法时的运行时间为基准，对比开启 SPvec 方法后程序的运行时间，计算加速比(基准运行时间/优化后运行时间 * 100%)。

测试结果如表 2 和图 8 所示。481.wrf 中分支一致的占比为 65%，在执行 SPvec 优化后整体性能提高了 10%。s1161 性能提升了 18.63%，s253 性能提升了 25.1%，s272 性能提升了 195.49%，s273 性能提升了 25.48%，s278 性能提升了 3.95%，s279 性能提升了 4.32%，s1279 性能提升了 82.44%。TSVC_2 测试集中有些程序的优化效果会明显优于其他项目，如 s272 和 s1279。这是因为 SPvec 方法的性能提升依赖于运行过程中投机成功的占比，占比越大，优化后的性能提升就越高。s272 和 s1279 被投机成功的次数接近 100%，而 s253 投机成功的次数占 67.4%，其余则更少。除此之外，在投机占比确定的情况下，跳过的指令开销 CostNBI 越大，程序执行 SPvec 优化算法后获得的性能提升也越多。如 s272 和 s1279，前者可跳过操作为 10，后者为 6，两者的投机成功概率都接近 100%，但前者经过 SPvec 算法优化后，性能提升效果要明显优于后者。

表 2 实验结果数据

Table 2 Experimental result data

题目	加速比/%
481.wrf	110
s1161	118
s253	125
s272	295
s273	125
s278	103
s279	104
s1279	182

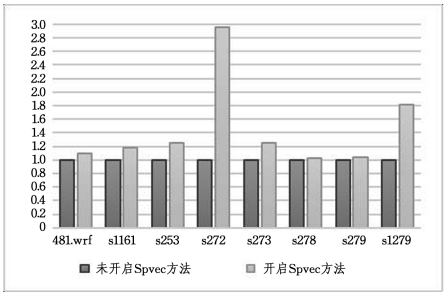

图 8 实验结果

Fig. 8 Experimental result

结束语 本文针对基于 IF 转换的控制流向量化中，冗余计算导致的代码效率低问题，提出了一种面向 SIMD 的控制流投机向量化方法，在向量代码中检测谓词相关区域，使用代价模型在区域内引导实施针对分支一致的投机变换，在运行时消除无用的谓词执行。从标准测试集中选取的 SPEC CPU 2006 中的 481.wrf 课题实验用例在优化后的性能提升达到 10%，TSVC_2 测试集中的 s1161 性能提升 18%，而 s253，s272，s273，s1279 几个用例的性能提升更是在 20%以上，这些实验数据证明了本文方法的有效性，可以为工业界优化 GCC 编译器中的控制流向量化提供参考。

参 考 文 献

[1] XIN N J, CHEN X C. Extending the vector instr-uction set for high-performance DSP matrix based on GCC[J]. Computer Engineering & Science, 2012, 34(1): 57-63.

[2] GAO W, LI Y Y, SUN H H, et al. An improved SIMD Vectorization method for Co-ntrol Flow [J]. Journal of Software, 2017, 28(8): 2046-2063.

[3] SRERAMAN N, GOVINDARAJAN R. A Vectorizing Compiler for Multi-media Extensions[J]. International Journal of Parallel Programming, 2000, 28(4): 363-400.

[4] LARSEN S, AMARASINGHE S. Exploiting Superword Level Parallelsm with Multimedia Inst ruction Sets[C]// Conference on Programming Language Design and Implementation. 2000: 145-156

[5] SUN H H, ZHAO R C, GAO W, et al. Quantification of control Flow Direction Based on Conditional Classification [J]. Computer Science, 2015, 42(11): 240-247.

[6] SUN H, FEY F, ZHAO J, et al. WCCV: Improvi-ng the vectorization of IF-statements with wa-rpcoherent conditions[C]// Proceedings of the ACM International Conference on Supercom-

puting. 2019：319-329.

[7] LANG H，KIPF A，PASSINGL，et al. Make the most out of your SIMD investments：counter control flow divergence in compiled query pipelines[C]// Proceedings of the 14th International Workshop on Data Management on New Hardware. 2018：1-8.

[8] KHORASANI F，GUPTA R，BHUYAN L N. Efficient warp execution in presence of divergence with collaborative context collection[C]// Proceedings of the 48th International Symposium on Microarchitecture. 2015：204-215.

[9] ALLEN F E，COCKE J. A Catalogue of Optimizing Transformations [M]// Rustin R，ed. Design and Optimization of Compilers. Prentice-Hall，Englewood Cliffs，1972：1-30.

[10] LIU B，LAIRD A，TSANG W H，et al. Combining Run-time Checks and Compile-time Analysis to Improve Control Flow Auto-Vectorization[C]// Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. 2022：439-450.

[11] SUJON M H，WHALEY R C，YI Q. Vectorization past dependent branches through speculation[C]// Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques. IEEE，2013：353-362.

[12] FUNG W W L，AAMODT T M. Thread block compaction for efficient SIMT control flow[C]// 2011 IEEE 17th International Symposium on High Performance Computer Architecture. IEEE，2011：25-36.

[13] ALLEN J，KENNEDY K，PORTERFIELD C，et al. Conversion of Control Dependence to Data Dependence[C]// Annual Symposium on Principles of Programming Languages. 1983：177-189.

[14] SHIN J，HALL M，CHAME J. Superword-level parallelism in the presence of control flow [C]// International Symposium on Code Generation and Optimization. IEEE，2005：165-175.

[15] SHIN J，HALL M W，CHAME J. Evaluating compiler technology for control-flow optimizations for multimedia extension architectures[J]. Microprocessors and Microsystems，2009，33(4)：235-243.

[16] PRAHARENKA W，PANKRATZ D，DE CARVALHO J P L，et al. Vectorizing divergent control flow with active-lane consolidation on long-vector architectures[J]. The Journal of Supercomputing，2022，78(10)：12553-12588.

[17] MOLL S，HACk S. Partial control-flow linearization[J]. ACM SIGPLAN Notices，2018，53(4)：543-556.

[18] SHIN J. Introducing control flow intovectoriz-ed code[C]// 16th International Conference on Parallel Architecture and Compilation Techniques(PACT 2007). IEEE，2007：280-291.

HAN Lin，born in 1978，doctor，professor，doctoral supervisor，is a special committee member of high performance computing of china computer federation (CCF)(No. 16416M). His main research interests include high per-formance computing，advanced compilation，program optimization，and domestic autonomous control.

NIE Kai，born in 1987，Ph.D，senior engineer，graduate tutor. His main research interests include ad-vanced compiler technology and high performance computing.