一种支持跨幅访存的向量化代码生成方法

李朋远 赵荣彩 高 伟 张庆花
(信息工程大学 郑州 450001) (数学工程与先进计算国家重点实验室 郑州 450001)

摘 要 随着SIMD扩展部件的迅速发展,自动向量化工具已逐渐成熟。现阶段的工具能对连续访存程序进行较好的处理,然而,大部分非连续访存的多媒体程序并不能被转换为高效的向量化代码。提出并实现了一种支持跨幅访存的向量化代码生成方法,其利用目标系统已有的基本数据处理指令实现多个向量间的任意重组来解决含有非连续访存语句的向量化代码生成问题。经过实验分析和验证,提出的代码生成方法能够将含有跨幅访存的语句转化为面向目标系统的高效向量化代码,以提高程序执行效率。

关键词 代码生成,跨幅访存,向量化,数据重组
中图法分类号 TP311.5 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.5.039

Effective Vectorization Technique for Interleaved Data with Constant Strides

LI Peng-yuan ZHAO Rong-cai GAO Wei ZHANG Qing-hua
(Information Engineering University, Zhengzhou 450001, China)
(State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China)

Abstract Due to the development of the SIMD extensions in general processors, automatic vectorizing compilers are widely used in various fields, especially in scientific and engineering computing area. Conventional vectorizing compilers can parallelize applications with continuous access successfully, but most irregular multimedia applications which access interleaved data cannot be vectorized correctly. To address this issue, this paper presented an effective vectorization technique for interleaved data with constant strides. We achieved any form of data regroupings with the help of the data processing instructions provided by targeted platforms. As a result, programs with interleaved data access are vectorized and vector codes are generated. The experimental results show that the proposed method can translate irregular applications with interleaved data access into high-performance targeted vectorized codes, thereby advancing the execution efficiency adequately.

Keywords Code generation, Stride access, Vectorization, Data regrouping

1 引言

微处理器中的SIMD扩展是指采用SIMD寄存器、SIMD扩展指令和SIMD扩展技术,由SIMD执行部件通过一条指令实现多个数据的同时操作,以充分发挥对媒体程序中的数据并行性的特点[1]。SIMD功能部件能够对多媒体程序中的数据进行并行处理,极大地提升了多媒体程序的运行速度。随着对浮点运算支持的完善,SIMD扩展部件的应用越来越广泛。

目前,SIMD扩展部件主要通过串行程序自动向量化的方式来提升程序的性能。当前自动向量化编译的方法主要有两种:1)借鉴向量机上的传统向量化方法[2,3](Classic Vectorzation, VP);2)由Samuel Larsen提出的超字并行(Superword Level Parallelism, SLP)向量化方法[4]。

传统向量化方法是出现最早也是最成熟的一种向量化技术,采用Allen-Kennedy算法[2,16],以数据间的依赖关系为基础,通过循环变换技术,针对最内层循环进行向量化。SLP向量化方法是针对循环语句级别的向量化,对循环的规整性要求较高,通过将基本块内不存在依赖环的一组同构操作打包成一个向量单元,寻求同一基本块内指令的并行性。

用户希望用最优的SIMD编译器将用高级语言编写的应用程序自动地转换为相应高效的使用SIMD指令的向量化代码。目前许多技术已经被提出以自动生成向量化代码[4,17,18]和解决SIMD代码生成过程中遇到的问题。然而,不连续内存访问仍然是阻碍SIMD程序性能提升的一个重要因素。SIMD装载和存储指令只能处理一个向量因子大小的连续内存单元中的数据,这样就导致SIMD编译器不能够自动地处理存储在非连续内存中的数据单元,本文针对此类问题设计了一种支持跨幅访存的向量化代码生成方法,对非连续内存中的数据首先进行数据重组,其次进行SIMD指令生成,最后经过重组使得目标数据依然存储在原来的内存位置。

到稿日期:2014-06-29 返修日期:2014-11-23 本文受"核高基"国家科技重大专项(2009ZX01036)资助。
李朋远(1989-),男,硕士生,主要研究方向为高性能计算、先进编译技术,E-mail:pengyuan_li@163.com;赵荣彩(1957-),男,博士,教授,博士生导师,主要研究方向为先进编译技术、软件逆向工程等;高 伟(1988-),男,博士生,主要研究方向为高性能计算、先进编译技术;张庆花(1991-),女,硕士生,主要研究方向为高性能计算、先进编译技术。

本文第 2 节对该领域的相关研究进行了描述；第 3 节主要描述了本文设计的代码生成方法的整体流程，对 SIMD 代码生成和数据重组进行了简单介绍；第 4 节详细论述了本文提出的支持跨幅访问的向量化代码生成方法算法实现过程；第 5 节通过实验对本文提出方法进行了分析和验证；最后是对本文工作的总结和对未来研究方向的展望。

2 相关研究

当前 GCC[5]、LLVM[6]、Open64[7] 等主流的编译器均采用传统向量化技术发掘循环内语句的并行性，文献[8,9]阐述了 GCC 中的自动向量化优化技术，Dorit Nuzman 等人于 2006 年在 GCC 中间表示 GIMPLE 中加入新的操作符和描述符，使其能够面向多平台自动生成向量化代码[10]。文献[11]阐述了一种跨基本块边界的代码生成优化技术，文献[12]阐述了基于多面体模型的代码生成方法，文献[13]阐述了 Open64 编译器中基于同构树模式匹配和动态划分的部分向量化实现方法。文献[14]以数据是否对齐为出发点，利用移位和提取插入方法分别提出了针对整型和浮点型数据的向量重组方法，解决了非对齐访存语句的向量化代码生成问题。文献[15]分别针对运行时的非对齐访存和不同数据类型间强制类型转换向量化代码生成问题，使用数据重组的方法将其解决。但现有文献均没有针对不连续数据的代码生成问题提出高效的解决方法。本文提出并实现了一种支持跨幅访存的向量化代码生成方法，通过传统向量化方法发掘程序中的并行性，利用数据重组实现不同向量间的重组，从而解决了跨幅访存的代码生成问题。

3 支持跨幅访存的代码生成整体流程

3.1 代码生成

向量化代码生成指的是将可以向量化的串行程序经过向量化发掘后变换为目标系统 SIMD 扩展部件可执行的向量化程序，从而可以提升程序的执行效率。在向量化代码生成时，文献[11]按照其中语句操作的不同，将所有的语句分为装载操作、计算操作、存储操作 3 类，分别进行向量化代码生成。在进行向量化代码生成时存在一个问题就是如何在新生成的向量寄存器中维护原来的标量所携带的依赖关系，本文通过一个数据结构来记录向量中的什么位置存放了哪些变量的方式来解决这个问题。

数组的装载操作就是将操作类型为 load 的语句转换为相应的向量化装载语句。比如 $t1=a[i]$ 转换为相应的向量化装载语句，即 $simd\_load(v\_1,a[i])$，其中 v_1 是向量寄存器，该操作的意思是将 $a[i]$ 从当前地址算起，连续存储一个向量长度的数据到向量寄存器 v_1 中，假设当前 $i=0$，向量长度为 128，数组类型为整型，则 v_1 中存储的数据为 $\{a[0],a[1],a[2],a[3]\}$。

操作类型不属于 load 也不属于 store 的语句统称为计算语句。比如 $r1=s1+t1$ 转换为相应的向量化计算语句，即 $v\_r1=simd\_add(v\_s1,v\_t1)$。该向量化语句将 v_s1 和 v_ti 中装载的一个向量长度的数据进行相加，将结果存储在 v_r1 向量寄存器中。

数组的存储操作就是将那些操作符类型为 store 的语句转换为相应的向量存储语句。比如 $b[i]=t$ 转换为相应的向量化存储指令，即 $simd\_store(v\_t,b[i])$。该向量化语句将 v_t 中装载一个向量长度的数据存储到以当前数组 $b[i]$ 地址算起一个向量长度的连续数组中。假设当前 $i=0$，向量长度为 128，数组类型为整型，v_t 中存储的数据为 $\{a[0],a[1],a[2],a[3]\}$，则该语句一次实现 $b[0]=a[0],b[1]=a[1],b[2]=a[2],b[3]=a[3]$。

3.2 数据重组

本文所描述的数据重组指的是通过移位、插入和提取等方法得到目标向量。形式化地提出这个问题，即现在有一个数组 $a[M*i]$，i 为数组索引，M 为数组索引系数，N 为向量因子，M 和 N 均是 2 的幂次方，假设 $i=0$，则要实现该数组的装载或存储，经过判断可知目标向量为 $v\_0=\{a[0],a[M],a[M*2],\cdots,a[M*(N-1)]\}$，为了便于描述，假设 M 与 N 相等，得到该向量的向量重组模型如图 1 所示。

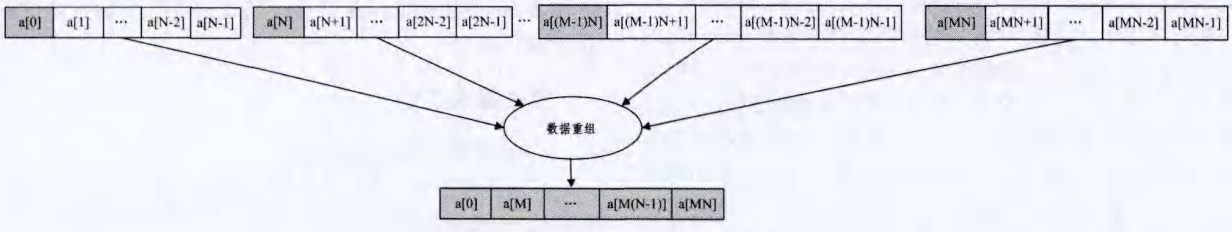

图 1 面向跨幅访存的向量重组模型

(1)目标系统的数据重组指令

目前许多处理器都提供了向量化重组指令，比如 intel 提供了混洗指令 pshuf，可以对两个向量中的数据进行一定限度的重新组合，得到新的向量。Altivec 指令集提供了更加强大的向量重组指令 vperm，该指令可以提取两个向量中的任何位置的元素来组成一个新的向量，多次使用该指令可以实现较为复杂的重组操作，如交叉存取。进行这些操作可能需要更多的指令周期，但是由于是由硬件实现，因此用软件实现某些重组比较容易，并且耗时较少。

(2)混洗操作

一个混洗操作 $Y_n \leftarrow Permute(X_n,P)$ 表示一个向量 X_n 根据模式 P 进行重排列操作，得到一个长度和 X_n 相同的向量 Y_n。模式 P 也可以表示为一个混洗矩阵 $P_{n\times n}$，即一个经过行重排列的等价方阵。一个混洗操作可以被认为是一个矩阵向量乘法，表达式为：

$$Permute(X_n,P_{n\times n})\equiv P_{n\times n}\cdot X_n$$

一个对混洗操作更为简洁的描述是把混洗的模式 P 看作一个索引向量，它的第 i 个元素是输入向量到输出向量进行重排列操作的一个索引值，即把输入向量的相应位置的元素移动到输出向量第 i 个元素的位置。例如：

$$Permute(\langle x_0,x_1,x_2,x_3\rangle,\langle 1,2,0,3\rangle)=\langle x_1,x_2,x_0,x_3\rangle$$

(3)交叉存取

通常 SIMD 扩展中的向量指令访问的数据都在向量寄存器中存储为向量。在计算中，一个相同的操作可能要被用于

对非连续的或者任意组织的数据的操作。比如 FIR(Finite Impulse Response)滤波函数和 FFT(Fast Fourier Transformation)傅里叶变换函数中存在的复数运算需要对实部和虚部进行交叉的存取和计算，一个简单的复数乘法源程序如图 2 所示。

```
for(i=0;i<256;i++){
    c[2i]=a[2i]*b[2i]-a[2i+1]*b[2i+1];
    c[2i+1]=a[2i]*b[2i+1]+a[2i+1]*b[2i];
}
```

图 2　标量复数乘法

进行向量化代码生成时，需要对复数数组的奇偶位进行交叉提取然后进行向量计算，计算得到的结果通过高位交叉和低位交叉存储到目标内存单元中。奇偶提取和高低位交叉存储的过程如图 3 所示。

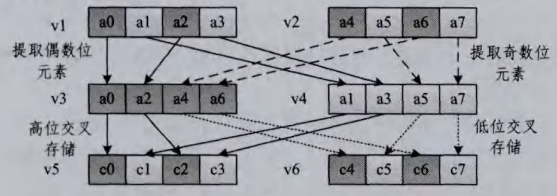

图 3　奇偶提取和交叉高位/低位存储操作

如果一个向量操作的输入是由另外的向量操作的输出经过向量混洗得到的，或者是由另外几个向量操作的输出结果经过拼凑而来，那么就需要进行向量重组。在这种情况下，从一个或者多个向量寄存器得到的数据需要经过混洗或者提取来匹配相应的 SIMD 指令。

3.3　实现流程

在对程序进行自动向量化变换时，向量寄存器的长度是不变的，数据类型的长度决定了向量因子的大小，不同的程序中跨幅访问系数可能存在着差异，本文在分析时设定向量寄存器长度为 256 位，数据类型为 float 类型，以面向跨幅访存的向量重组模型为基础，分析相应的向量化代码生成方法。支持跨幅访存的向量化代码生成是在向量化分析阶段和向量化发掘阶段通过的前提下完成的，具体过程可以归结如下：首先进行向量化预分析，分析当前循环是否存在阻止向量化的语句，如 goto 语句，并根据依赖关系判定当前循环中语句是否可以进行向量化；接着对整个循环进行向量化发掘，将所有的标量操作符类型转换为向量操作符类型；最后进行代码生成，将循环中的语句分为存在跨幅访存和不存在跨幅访存两类。对于不存在跨幅访问的语句，直接采用第 2 小节所描述的常规代码生成方法生成向量化代码；对于存在跨幅访问的语句，需要通过本文提出的向量化代码生成方法进行代码生成。整体的流程如图 4 所示，其中虚线框中部分为本文提出的代码生成部分流程图。

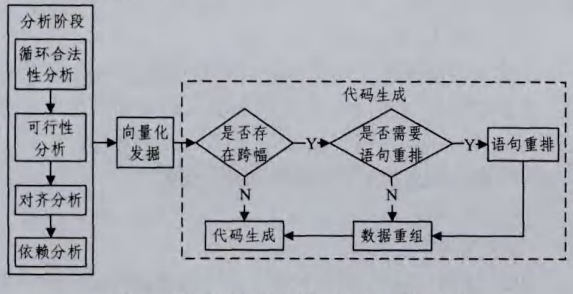

图 4　支持跨幅访存的向量化整体流程图

向量化代码生成技术对生成面向目标系统的包含 SIMD 指令的高效代码具有重要的作用，该框架仅在原有的代码生成框架下增加了向量重组和语句重排优化部分，在延续向量化编译工具原有的优点的同时，提升了工具向量化跨幅访存语句的能力。算法实现如图 5 所示。

```
1. //功能：支持跨幅访存的代码生成；
2. //输入：基本块 bb，SIMD_VL_List，v_space；
3. //输出：newbb；
4. algorithm code_generation_of_stride(bb, SIMD_VL_List，v_space)
5.   for all stmt in bb
6.     if (stride_coff > 1)
7.       if (stmt_resort_is_need(stmt，bb))
8.         stmt_resort(bb)；
9.       endif
10.      vector_regroup_improve(v_space，SIMD_VL_List)；
11.    else
12.      code_generation(bb)；
13.    endif
14. end
15. end algorithm
```

图 5　支持跨幅访存的向量化代码生成算法

图 5 中，*stride_coff* 表示当前循环索引下的数组系数，*stmt_resort_is_need*()函数根据基本块 bb 中的语句判断当前语句是否可以进行语句重排，*smt_resort*()调用语句重排优化子模块，在不进行语句重排且数组系数大于 1 的条件下进行向量重组，由 *vector_regroup_improve*()子模块完成。当前 bb 中若数组访问系数为 1，直接调用编译器现有的 *code_generation*()实现代码生成。该算法的优势可以总结为以下几点：

- 该方法按照目标 SIMD 功能部件的指令特性，分为有重组指令和无重组指令两种情况进行代码生成。
- 目标 SIMD 平台无重组指令时，该方法能够对已有向量进行任意形式的数据重组，从而得到目标向量。
- 当程序中同时访问了数组的奇数位和偶数位时，直接进行数据重组的开销比较大，本文提出了语句重排序的方法，在不改变程序的语义的前提下，更改循环中语句顺序，降低数据重组开销。

4　相关算法实现

4.1　支持数据重组的代码生成

多媒体程序中大量应用不连续地进行内存访问，如复数的加法、乘法等。这些语句的操作数不能直接从内存中装载，而且也不能直接将计算结果从向量寄存器中存储到数据内存中。以图 6 中的程序段为例，由于只对数组的偶数位进行装载和存储，而 SIMD 扩展部件仅仅支持数组连续装载和存储，因此要对该程序进行向量化，首先需要在预分析阶段对跨幅的数组访问进行支持，然后在代码生成阶段进行数据重组优化。

```
for(int I=0;i<N;i++){
  c[2*i]=a[2*i]+b[2*i]
}
```

图 6　跨幅访存示例

假设数组类型为 float 类型，则向量化因子为 4，根据图 1 可知，图 6 中程序的访存模型如图 7 所示。为了拼凑出向量 {*a*[0]，*a*[2]，*a*[4]，*a*[6]}，首先从连续的内存空间装载两次，然后利用一次重组，才能得到目标向量。

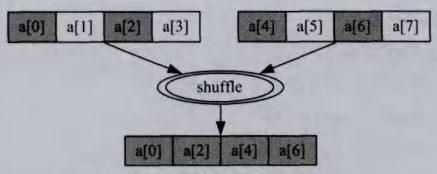

图 7　图 6 中程序的访存模型

算法实现如图 8 所示，其中 *load_number* 表示重组出目标向量需要装载的次数，根据数组 *array* 的跨幅系数可以得到。假如至多通过 N 次重组得到的结果是有收益的，函数 *check_is_benefit*()检测当前重组情况是否有收益，则若有收益，则根据第一个操作数获得初始装载地址 *address*0，从 *address*0 进行向量装载，获得向量 $V0$，将 *address*0 偏移向量长度获得 *address*1，从 *address*1 进行向量装载，获得向量 $V1$，按照此方法装载 *load_number* 个连续的向量，存储在数组 *v_load*[*load_number*]中；若重组无收益，则返回 NULL。最后按照数组 *array* 的跨幅系数提取装载向量中的元素插入到目标向量中。

```
1. //功能:向量重组;提取非连续的数组元素到目标向量;
2. //输入:数组 array;
3. //输出:目标向量 v_des;
4. algorithm vector_regroup(array)
5.   load_number=compute_load_number(array);
6.   if (check_is_benefit(load_number));
7.     address0=get_start_address(array);
8.     v_load[load_number]=get_vector(address0,array);
9.   else
10.     return NULL;
11.  endif
12.  根据 array 的跨幅系数提取 v_load[load_number]中的每个向量
       的相应位置的元素;
       将提取出的元素依次插入到目标 v_des 中;
13.  return v_des;
14. end algorithm
```

图 8　面向跨幅访存的向量重组算法

通过计算得到图 6 中程序需要装载的次数 *load_number* =2，首先装载向量 $v0=\{a[2*i],a[2*i+1],a[2*i+2],a[2*i+3]\}$ 和 $v1=\{a[2*i+4],a[2*i+5],a[2*i+6],a[2*i+7]\}$，通过重组获得的 $v\_des=shuffle(shuffle\_mode,v0,v1)$ 即为目标向量。

向量化重组的目标是对任意形式的重组要求实现数据重组。我们可以形式化地提出这个问题，即现在有一个需要进行向量重组才能得到的目标向量 *v_des*，其中需要装载的标量为{*e*1,*e*2…,*en*}，n 为向量化因子。

对于一个变量 t，经过多次数据重组后，它的变量存放位置可能有多个，图 9 所示类 *SIMD_VL* 可以用来存放其中的一个位置，使用一个 *SIMD_VL* 的链表来存放变量 t 所有的存放位置。进行数据重组的时候，可以从多个存有这个变量值的位置中读取操作数。

```
class SIMD_VL {
  WN_OFFSET_vpreg; //所在向量寄存器
  INT32_slot; //向量中的槽位
}
```

图 9　变量位置信息类

假设 $v1=\langle t0,t1,t2,t3\rangle$，$v2=\langle t4,t5,t6,t7\rangle$，如果后面的代码中需要重组出 $v\_des=\langle t0,t2,t4,t6\rangle$，通过查找各操作数的所在向量和槽位信息，就可以利用 $v1$ 和 $v2$ 重组生成。

对于一个需要重组生成的向量 v，能够从两个向量中重组出来是最理想的情况；否则就需要递归地调用向量重组，从而从多个向量中重组出目标向量。可以对向量重组算法进行改进以实现该重组方法。

向量重组改进算法的实现如图 10 所示：

```
1. //功能:向量重组,从已有向量寄存器中提取出目标向量所需要的
     数据单元,存储到目标向量中;
2. //输入:当前基本块中所有的向量 v_space;SIMD_VL_List;
3. //输出:目标向量 v_des;
4.  algorithm vector_regroup_improve(v_space,SIMD_VL_List)
5.    for v in v_space
6.      if v==v_des
7.        return v;
8.      else
9.        continue;
10.     endif
11.   end
12.   for element in v_des
13.       if (SIMD_VL_List ! =NULL)
14.         提取出该 element,插入到 vec_tmp;
15.       else
16.         continue;
17.       endif
18.     end
19.   end
20.  if(vec_tmp==v_des)
21.    return vec_tmp;
22.  else
23.    for element in vec_des
24.      if elememt not in vec_tmp
25.        装载含有该元素的数组到向量 v_load;
26.        从 v_load 中提取出该元素,插入到 vec_tmp;
27.      else
28.        continue;
29.      endif
30.    end
31.    return vec_tmp;
32.  endif
33. end algorithm
```

图 10　向量重组改进算法

对于不连续内存的向量存储，为了不覆盖原有的奇数位的数据，需先将存储位置的数据加载进来，获得向量 $v0$、$v1$，具体过程同向量装载中的 a、b。然后通过数据重组，最后再对重组后的向量进行向量存储。

4.2　语句重排序

当一个程序段中需要对同一数组的奇数位和偶数位同时装载或存储时，程序段如图 11(a)所示，按照上述方法生成的向量化代码如图 11(b)所示。该向量化代码中同时存在对数组 *b*[*i*]和 *c*[*i*]的偶数位和奇数位的装载和存储，很明显，其中存在重复的内存装载访问(称为冗余装载)，这在一定程度上影响了向量化代码的执行效率。由此本文提出一种新的优化技术语句重排，该优化技术可以解决上述冗余装载以及冗余存储问题，在一定程度上能提高向量化代码的执行效率。语句重排优化技术在不改变原来程序段语义的前提下，通过发掘基本块中针对数组的交叉存取位置，调整语句顺序，生成更

加高效简洁的向量化代码。按图 11(a)所示程序段进行语句重排后生成的向量化代码如图 11(c)所示，与图 11(b)进行比较，语句规模约减少 46%。

```
for(i=0; i<256; i++){
  a[2*i]=b[2*I] - c[2*i];
  a[2*i+1]=b[2*I+1] + c[2*I+1]
}
```

(a)

```
for(i=0; i≤255;i+=8){
  simd_laod(p_6, c[2*i]);
  simd_load(p_7, c{2*I + 16});
  p_4=vec_vshufh(1, p_6, p_7);
  simd_load(p_8, b[28i]);
  simd_load(p_9, b[28i + 16]);
  p_5=vec_vshufh(1, p_8, p_9);
  p_3=p_5 - p_4;
  simd_laod(p_10, a[2*i]);
  simd_load(p_11, a[2*I + 16]);
  p_12=vec_vshufh(3, p_3, p_10);
  p_13=vec_vshufh(4, p_3, p_11);
  simd_store(p_12, a[2*i]);
  simd_store(p_13, a[2*I + 16]);
  simd_load(p_14, c{2*i});
  simd_load(p_15, c[2*I + 16]);
  p_1=vec_vshufh(2, p_14, p_15);
  simd_laod(p_16, b[2*i]);
  simd_load(p_17, b[2*I + 16]);
  p_2=vec_vshufh(2, p_16, p_17);
  p_0=p_2 + p_1;
  simd_load(p_18, a[28i]);
  simd_load(p_19, a[2*I + 16]);
  p_20=vec_vshufh(5, p_0, p_18);
  p_21=vec_vshufh(6, p_0, p_19);
  simd_store(p_20, a[2*i]);
  simd_store(p_21, a[2*I +16]);
}
```

(b)

```
for(i=0; i≤255; i+=8){
  simd_laod(p_6, c[2*i]);
  simd_load(p_7, c{2*I + 16});
  p_4=vec_vshufh(0, p_6, p_7);
  p_1=vec_vshufh(1, p_6, p_7);
  simd_load(p_8, b[28i]);
  simd_load(p_9, b[28i + 16]);
  p_5=vec_vshufh(0, p_8, p_9);
  p_2=vec_vshufh(1, p_8, p_9);
  p_3=p_5-p_4;
  p_0=p_2 +p_1;
  p_10=vec_vshufh(3, p_0, p_3);
  p_11=vec_vshufh(2, p_0, p_3);
  simd_store(p_10, a[2*i]);
  simd_store(p_11, a[2*I + 16]);
}
```

(c)

图 11　数组奇数位和偶数位同时访问例子

语句重排算法如图 12 所示。

```
1. //功能:语句重排
2. //输入:基本块 bb
3. //输出:经过语句重排的基本块 newbb
4.   algorithm stmt_resort(bb)
5.     foreach 语句 s in bb do
6.       if (s 是对某一数组的装载(存储) && 该语句仅访问数组的
           奇数位(偶数位))
7.       向下遍历整个 bb,查找是否存在语句 s1 对同一数组的装载
         (存储) && 仅访问该数组的偶数位(奇数位);
8.       if(存在 s1)
9.         check_dependece_stmt(s1);
10.          resort_stmt(s1);
11.       else
12.         放弃重排
13.       endif
14.     else
15.       放弃重排
16.     endif
17.   end
18. end algorithm
```

图 12　语句重排序算法

其中，bb 是需要进行分析的循环体，newbb 是满足条件进行语句重排后的基本块，函数 *check_dependence_stmt*(*s*)用于检测调整语句顺序后的语句依赖关系，如果语句 *s*1 被调整后语句间依赖关系被破坏，则不进行语句重排；若语句间依赖关系不发生变化则进行语句调整，由函数 *resort_stmt*(*s*)函数完成。

5　实验结果与分析

5.1　实验平台

本文基于开源编译器 open64 实现了这种支持跨幅的代码生成方法，编译环境为 Linux 操作系统，版本为 Readhat Enterprise 5。实验平台 CPU 主频为 2.0GHz，内存为 2GB，L1 数据 cache 为 32kB，L2 cache 为 256kB，基本页面为 8kB。所使用的测试平台是国产 CPU SW1600 处理器，支持 256 位的 SIMD 扩展指令。

源程序首先经过 open64 的向量化编译器转化为向量化程序，然后再用基础编译器编译成二进制代码并在实验平台上运行。最后用串行程序的运行时间除以向量化程序的运行时间便得到向量化的加速比。

5.2　实验测试集

本文选择的测试程序是数字信号处理中经常遇到的 FIR 和 FFT 程序以及 gcc-vec 测试集中的部分程序，FIR 和 FFT 程序的核心循环如图 13 所示，其中数组类型为 float 类型，*real*、*imag*、*rtemp*、*itemp* 为 float 型变量，*i*、*j*、*m*、*ia*、*n2*、*nh*、*ny* 为整型变量。gcc-vect 测试集所选程序列表如表 1 所列，由于所选程序的核心循环中迭代规模较小，导致执行时间几乎可以忽略不计，因此增加核心循环迭代次数，使得得到的实验数据更加具有说服力。

表 1　选自 gcc-vect 测试集的程序列表

程序名称	扩大规模	数据类型
vect-strided-float. c	16×100×1024	float
vect-107. c	16×100×1024	float
vect-1. c	16×100×1024	float
vect-10. c	16×100×1024	short
fast-math-slp-27. c	16×100×1024	float
vect-outer-3b. c	16×100×1024	float

```
for (i=0; i < 2 * ny; i +=2)
  for (j=0; j < 2 * nh; j +=2)
  {
    real +=h[j] * x[i-j] - h[j+1] * x[i+1-j];
    imag +=h[j] * x[i+1-j] + h[j+1] * x[i-j];
}
```

(a)FIR 核心循环

```
for(i=0; i < n2; i++)
{
  m=ia + n2;
  rtemp=c * x[2 * m] + s * x[2 * m+1];
  itemp=c * x[2 * m+1] - s * x[2 * m];
  x[2 * m]=x[2 * ia] - rtemp;
  x[2 * m+1]=x[2 * ia+1] - itemp;
  x[2 * ia]=x[2 * ia] + rtemp;
  x[2 * ia+1]=x[2 * ia+1] + itemp;
  ia++;
}
```

(b)FFT 核心循环

图 13　FIR 与 FFT 程序核心循环

5.3　实验数据及分析

(1)实验结果如图 14 和图 15 所示。

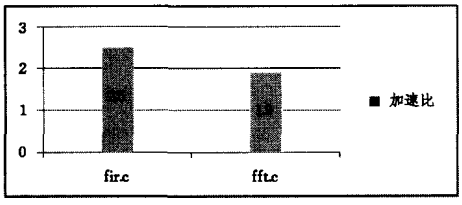

图 14　FIR 与 FFT 程序加速比

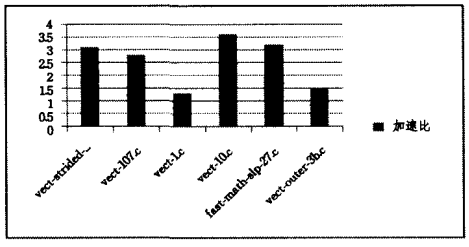

图 15　选自 gcc-vect 测试集程序加速比

(2)结果分析

本文提出的支持跨幅访存的代码生成方法已经在开源编译器 open64 中实现，该方法能够将绝大部分的存在跨幅的数组装载和存储语句转化为面向目标系统的向量化代码。

FIR 和 FFT 程序在通信、图像处理、模式识别等领域有着广泛的应用，FIR 程序规模为 4096×2048×32，串行执行时间为 0.528s，向量化代码执行时间为 0.212s，执行效率提升近 60%，加速比达到 2.5。FFT 程序规模为 1024×1024×100，串行执行时间为 0.874s，向量化代码执行时间为0.513s，执行效率提升近 40%，加速比达到 1.9。

从 gcc-vect 测试集中选择的 6 个含有跨幅访存语句的程序，平均加速比能达到近 2.6，其中 vect-10.c 的加速比最高为 3.6，分析可知，其程序中的数组类型为 short 类型，一次可以处理 16 个数组元素，并且满足语句重排优化条件，因此加速效果比较明显。vect-1.c 的加速比最小只有 1.3，分析原因可知，其程序中能够被向量化的循环只占到了整个程序的 25%，因而加速效果并不明显。

结束语　本文工作可总结为以下几点：

• 本文在 open64 编译器原有的代码生成框架下，针对程序中的跨幅访存语句不能生成向量化代码的问题，提出了该问题的解决方法。该方法主要是通过利用目标系统已有的数据重组指令，实现多个向量间的重组，从而生成面向目标系统的高效向量化代码。

• 本文给出了针对跨幅访存的向量重组算法和改进算法，向量重组算法首先需要对数组按照一定的装载次数进行装载，再进行重组；向量重组改进算法能够从已有的向量中提取目标向量中的元素，从而实现任意向量间的重组。

• 实验中采用的 FIR 和 FFT 程序在通信和图像处理等领域有着十分广泛的应用，结果证明本文提出的支持跨幅访存的代码生成方法能够较好地提高源程序的执行效率。

下一步的工作准备对跨幅访存语句代码生成后的优化工作进行进一步的研究，提升自动向量化发掘的效率，使得生成的向量化代码高效、简洁。

参 考 文 献

[1] Hassaballah M, Omran S, Mahdy Y B. A review of SIMD multimedia extensions and their usage in scientific and engineering applications[J]. The Computer Journal, 2008, 51(6): 630-649

[2] Allen R, Kennedy K. Automatic translation of Fortran programs to vector form[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1987, 9(4): 491-542

[3] Zima H, Chapman B. Supercompilers for parallel and vector computers[M]. ACM, 1990

[4] Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets[M]. ACM, 2000

[5] Griffith A. GCC: the complete reference [M]. McGraw-Hill, Inc., 2002

[6] Lattner C, Adve V. The LLVM compiler framework and infrastructure tutorial[M]// Languages and Compilers for High Performance Computing. Springer Berlin Heidelberg, 2005: 15-16

[7] Lin M, Yu Z, Zhang D, et al. Retargeting the open64 compiler to powerpc processor[C]// International Conference on Embedded Software and Systems Symposia, 2008 (ICESS Symposia'08). IEEE, 2008: 152-157

[8] Naishlos D. Autovectorization in GCC[C]// Proceedings of the 2004 GCC Developers Summit. 2004: 105-118

[9] Nuzman D, Zaks A. Autovectorization in GCC-two years later [C]// Proceedings of the 2006 GCC Developers Summit. 2006: 145-158

[10] Nuzman D, Henderson R. Multi-platform auto-vectorization [C]// Proceedings of the International Symposium on Code Generation and Optimization. IEEE Computer Society, 2006: 281-294

[11] Tanaka H, Ota Y, Matsumoto N, et al. A new compilation technique for SIMD code generation across basic block boundaries [C]// Proceedings of the 2010 Asia and South Pacific Design Automation Conference. IEEE Press, 2010: 101-106

[12] Bastoul C. Code generation in the polyhedral model is easier than you think[C]// Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques. IEEE Computer Society, 2004: 7-16

(下转第 203 页)

全率和查准率都有不同程度的改善，特别是查准率有着明显的优化。这说明，基于本体的多约束服务发现能够提供更高精度的服务发现，这对于提高整个服务系统的性能有重要的意义。

3.2.4 实验结果解释

实验结果表明基于本体的多约束服务发现机制在查准率方面有很明显的改善，同时我们在实验过程中发现了导致这种结果产生的两类比较典型的数据，如表1和表2所列。

表1 第1类数据

说明	请求者需求	多约束服务发现结果	一般服务发现结果
打印机类型	喷墨式	喷墨式	喷墨式
纸张类型	A5	A3	A5
待服务时间	无要求	0	0
距离	无要求	2.2	4.4
打印机/请求者位置	5_5	7_7	1_1

表2 第2类数据

说明	请求者需求	多约束服务发现结果	一般服务发现结果
打印机类型	A4	B3	A4
纸张类型	无要求	0	1
待服务时间	无要求	2.2	4.4
距离	7_7	9_9	3_3
打印机/请求者位置	A4	B3	A4

注：①实验基于一个假设：A3、B3、A4、B4、A5、B5能打印纸张类型为前者的打印机就能打印纸张类型为后者的纸张，比如，能打印B3类型纸的打印机就能打印A4、B4、A5、B5类型的纸张；②实验实例x_y中x表示楼层号，y表示房间号，如7_7表示7楼7号房间对应的打印机。

观察发现，基于本体的多约束服务发现机制的返回结果受两个方面的影响：1)该机制能够发现在语义上满足服务请求者的服务，这导致有较高的召回数量；2)该机制能从多方面综合考察某个服务是否为优质服务导致较少的召回数量。两个因素综合导致基于本体的多约束服务发现机制相对于一般约束的服务发现机制在查全率上相当，而在查准率上更有优势。

结束语 从Web服务发现的角度看，如何从大量满足功能的服务中选择出最佳的服务，是提高服务质量的关键问题。本文提出的基于本体的多约束服务发现机制从不同于关键词匹配的语义匹配角度出发引入多约束的概念来发现最佳的服务，使服务发现精度提高，大大提高了服务系统的性能。如何结合已有方法实现查全率的进一步提升将成为后续的研究内容。

参考文献

[1] Berners-Lee T, Hendler J, Lassila O. The semantic Web [J]. Scientific American, 2001, 284(5): 35-43

[2] Corby O, Dieng-Kuntz R, Faron-Zucker C. Searching the semantic web: approximate query processing based on ontologies[J]. IEEE Intelligent Systems, 2006, 21(1): 20-27

[3] Ji J, Bu F, Cai H, et al. Ontology model for semantic web service matching[M] // Information Computing and Applications. Springer Berlin Heidelberg, 2010: 181-188

[4] Yang Z, Chen J, Wu B. A new ontology-based service matching algorithm[C] // The 6th World Congress on Services. IEEE, 2010: 170-171

[5] 叶蕾，张斌. 基于功能语义的Web服务发现方法[J]. 计算机研究与发展，2007，44(008)：1357-1364

[6] Chakraborty D, Joshi A, Yesha Y, et al. Toward distributed service discovery in pervasive computing environments [J]. Transactions on Mobile Computing, IEEE, 2006, 5(2): 97-112

[7] Caceres C, Fernández A, Ossowski S, et al. Agent-based semantic service discovery for healthcare: an organizational approach [J]. Intelligent Systems, IEEE, 2006, 21(6): 11-20

[8] Kritikos K, Plexousakis D. Requirements for QoS-based Web service description and discovery[J]. Transactions on Services Computing, IEEE, 2009, 2(4): 320-337

[9] Zhou C, Chia L T, Lee B S. Semantics in service discovery and QoS measurement[J]. IT professional, 2005, 7(2): 29-34

[10] 刘志忠. 多本体环境下服务发现的相关技术研究[D]. 武汉：国防科学技术大学研究生院，2007

[11] Meditskos G, Bassiliades N. Structural and role-oriented Web service discovery with taxonomies in OWL-S[J]. Transactions on Knowledge and Data Engineering, IEEE, 2010, 22(2): 278-290

[12] Segev A, Sheng Q Z. Bootstrapping ontologies for web services [J]. Transactions on Services Computing, IEEE, 2012, 5(1): 33-44

[13] Lacasta J, Nogueras-Iso J, Béjar R, et al. A Web Ontology Service to facilitate interoperability within a Spatial Data Infrastructure: Applicability to discovery[J]. Data & Knowledge Engineering, 2007, 63(3): 947-971

(上接第199页)

[13] Ramanarayanan R, Gupta M, Chakraborty S S, et al. Harnessing partial vectorization in Open64 compiler[C] // 2014 IEEE International Advance Computing Conference (IACC). IEEE, 2014: 813-824

[14] 魏帅，赵荣彩，姚远. 面向国产CPUSW-1600的向量重组[J]. 计算机应用与软件，2012，28(11)：230-233

[15] Wu P, Eichenberger A E, Wang A. Efficient simd code generation for runtime alignment and length conversion[C] // International Symposium on Code Generation and Optimization, 2005 (CGO 2005). IEEE, 2005: 153-164

[16] Allen R, Kennedy K. Optimizing compilers for modern architectures: a dependence-based approach[M]. San Francisco: Morgan Kaufmann, 2002

[17] Leupers R. Code selection for media processors with SIMD instructions[C] // Proceedings of the Conference on Design, Automation and Test in Europe. ACM, 2000: 4-8

[18] Sreraman N, Govindarajan R. A vectorizing compiler for multimedia extensions[J]. International Journal of Parallel Programming, 2000, 28(4): 363-400