类型转换语句的SLP发掘方法

赵 博 赵荣彩 李雁冰 高 伟
(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 多媒体技术的迅速发展使得越来越多的处理器集成了SIMD扩展,当前的编译器大多数都已实现了自动向量化功能。为了发掘迭代内并行,一些编译器在自动向量化模块中引入了SLP向量化方法。多媒体数据的密集存储和规则运算使得在处理多媒体数据时需要进行频繁的数据类型转换,而目前的SLP向量化方法对数据类型转换的处理能力还不完善。为了在存在大量数据类型转换语句的程序中发掘更多的SLP向量化机会,提出了一种类型转换语句的SLP发掘方法,它能够在SLP向量化框架下利用数据重组实现具有相同向量化因子和不同向量化因子的数据类型之间的转换。实验结果表明,该方法能够有效地对类型转换语句进行SLP向量化发掘,提高了程序的向量化执行效率。

关键词 类型转换,数据重组,SLP,SIMD,向量因子
中图法分类号 TP312 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2014.11.004

SLP Exploitation Method for Type Conversion Statements

ZHAO Bo ZHAO Rong-cai LI Yan-bing GAO Wei
(Information Engineering University, Zhengzhou 450001, China)
(State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China)

Abstract With the rapid development of multimedia technology, more and more processors are integrated with SIMD (Single Instruction Multiple Data) extensions. Almost all current compilers are equipped with automatic vectorization. SLP (Superword Level Parallelism) method is introduced to some compilers in order to exploit intra-iteration parallelism. Frequent data type conversions are required when handling the multimedia data because of its characteristics of intensive storage and regular computation. However, the processing capacity of current SLP technique is not sufficient enough. In order to exploit more opportunities of SLP vectorization in programs with large amounts of type conversion statements, an SLP vectorization method was proposed to deal with type conversion statements. This method is able to handle the type conversion statements with the same or different vector factors using data regrouping in the framework of SLP vectorization. The experiment results show that the proposed method is efficient to exploit the SLP vectorization of data type conversion statements and is effective to improve the performance of the vectorization programs.

Keywords Type conversion, Data regrouping, SLP, SIMD, Vector factor

1 引言

随着多媒体应用、图形学、嵌入式和数字计算等技术的飞速发展,SIMD扩展部件得到了越来越广泛的应用。很多的通用处理器集成了SIMD扩展,如Intel在其处理器上集成的SSE/AVX指令集系列、AltiVec等,都集成了SIMD扩展。同时,SIMD扩展单元被广泛应用到图像和音频文件处理、游戏开发和DSP处理器中。向量化属于细粒度并行,各体系结构的指令集较为复杂且不同平台的指令集差异较大,手工向量化工作量大且容易出错,难以实现,因而SIMD自动向量化技术成为代替手工编写向量化程序的一种重要途径。

超字级并行(SLP)识别出其中相邻且连续的语句,对其中的同构语句打包,然后根据定义-使用关系和使用-定义关系进行包扩展,从而进行向量化。理论上来说SLP向量化能够对所有的同构语句进行打包,在并行性一般或者较小的情况下也能够有效发掘向量化性能。它主要用于发掘迭代内的并行性,可以更有效地对科学计算领域的程序进行向量化,但相对于传统向量化算法,SLP算法研究起步较晚,因此其对很多特殊程序结构,比如类型转换、控制流、结构体运算等处理得还不够充分。

近些年来,研究人员面向SLP向量化方法进行了大量深入的研究[1-10],这些工作都是试图通过不同的方法提升程序的向量化能力,或通过结合SLP向量化和传统向量化方法进一步挖掘程序并行性。然而,在程序中存在数据类型转换问题时,这些方法无法有效挖掘程序的并行性,而当前多媒体文件处理程序中存在大量的数据类型转换问题,因此针对该问

到稿日期:2014-01-14 返修日期:2014-08-08 本文受"核高基"国家科技重大专项(2009ZX01036-001-001-2)资助。
赵 博(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:zhaobo197359@gmail.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;李雁冰(1989-),男,硕士生,主要研究方向为高性能计算、先进编译技术;高 伟(1989-),男,硕士生,主要研究方向为高性能计算、先进编译技术。

题的 SLP 向量化发掘还有待进一步研究。

数据类型的转换在多媒体相关的工作负载中十分普遍，因为多媒体的数据通常进行密集型存储，如 short 型或者 char 型，然而在运行时又使用定点等类型来减少舍入误差。例如在图 1 中，输入数组 b 是 16bit 的 short 型数据，但是在进行计算时要被强制转化为 32bit 的 int 型数据。在向量化时，有必要对数据类型转换进行相关处理。

```
...
#define M,N;
short b[N];
int I,j;
for (j=0;j<N;j++) {
  int a = 0;
  for (i=0;i<M-j;i++)
    a += ((int)b[i]*(int)b[i+j])<<2;
  ...
}
```

图 1　short 到 int 数据类型转换示例

针对当前多媒体扩展进行向量化代码生成时的数据类型转换问题，文献[11]提出了一种针对运行时对齐分析和数据长度变换的高效的 SIMD 代码生成方法，即通过流变换的方法对该运行时的对齐信息进行分析来进行向量化的发掘。文献[12]提出了一种带类型恢复的编译器源源翻译技术，即针对源源翻译变换时编译器优化带来的中间表示上表达式形式的改变进行类型恢复，解决由于中间表示类型变换导致结果错误的问题。本文提出了一种面向类型转换语句的 SLP 发掘方法来对程序中存在的类型转换语句进行向量化。

文献[11]主要通过计算数据流的等价变换和移位操作来实现对存在数据长度转换的运行时对齐信息的处理，并不是主要针对数据类型转换进行向量化发掘，而且文中所涉及的数据类型转换只是数据长度的转换，对数据的符号扩展变换等并未涉及。因此本文对数据类型转换的 SLP 发掘过程进行了详细描述，同时通过基本的数学运算对数据的符号变换进行了处理。

文献[12]的目标是如何提高源源翻译的健壮性，保证翻译出来的程序在语义和执行结果上的正确性和能够通过编译，同时不考虑与向量化相关的具体细节。而本文主要考虑的是从中间语言到生成向量化语句时 SLP 向量化发掘所必需的向量化因子、打包等因素，由于向量寄存器长度通常是固定的，因此数据大小的不同使得在向量化时每个向量寄存器中打包的数据元素个数是不同的，即向量化因子不同。例如一个长度为 128 位的向量寄存器能够装载 4 个 32 位的 int 类型的数据或者 2 个 64 位的 double 类型的数据，这样不同向量因子长度的数据进行类型转换时，由于每个向量包中元素个数不同，因此需要用到数据重组、交叉存取和向量混洗等操作，这在本文第 2 节有详细的叙述。

针对当前 SLP 向量化发掘中对类型转换语句的处理存在的问题，本文给出一种类型转换语句的 SLP 发掘方法，结合数据重组，给出了类型转换语句的 SLP 发掘框架和算法，从不同数据类型进行向量化时对向量因子的大小方面进行考虑，能够解决相同和不同向量因子长度之间数据类型的转换问题。

本文第 2 节对数据类型转换的分类进行了描述，并对在转换过程中涉及的数据重组进行了介绍；第 3 节详细地论述了本文所提出的类型转换语句的 SLP 发掘过程，并用示例的方式分类阐述了具体的向量化发掘过程，给出了相应的算法实现；第 4 节通过实验对本文提出的类型转换 SLP 向量化发掘方法进行了分析和支撑；最后是对本文工作的一个总结和未来研究方向的一些预想。

2　研究基础

2.1　数据类型转换

数据类型转换被广泛地用于多媒体应用中，比如图像处理、音频制作、数据压缩和游戏开发等，以 C 语言为例，有符号和无符号的各种数据类型之间的转换相当多，每两种数据类型之间定义一个转换规则，也会产生一个较大的规则集。由于各种体系结构对于各种类型的数据的支持也不尽相同，很多类型转换情况要根据具体情况进行分析。

现在要把一个 M 位的数据类型(值为 m)转换为一个 N 位的数据类型，可能的转换分类情况如表 1 所列。

表 1　数据类型转换的分类

待转换类型	如果 $M>N$	如果 $M=N$	如果 $M<N$
有符号定点到有符号定点	如果 m 的值在目标类型取值范围内，值不变；否则，implementation-defined	m 值不变	m 值不变
无符号定点到有符号定点	如果 m 的值在目标类型取值范围内，值不变；否则，implementation-defined	如果 m 的值在目标类型取值范围内，值不变；否则，implementation-defined	m 值不变
有符号定点到无符号定点	如果 $m\geqslant 0$，值不变；如果 $m<0$，2^N+m	如果 $m\geqslant 0$，值不变；如果 $m<0$，2^N+m	如果 $m\geqslant 0$，值不变；如果 $m<0$，2^N+m
无符号定点到有符号定点	如果 $m\geqslant 0$，值不变；如果 $m<0$，2^N+m	m 值不变	m 值不变
浮点到定点	按 0 截断，如果 m 的整数部分超出目标类型的取值范围，则进行边界值截断。		
定点到浮点	如果 m 在目标类型的取值范围，则其值不变，但是有可能损失精度；如果 m 的值超出目标类型的取值范围，则进行边界值截断。		
浮点到浮点	如果 m 在目标类型的取值范围，则其值不变，但是有可能损失精度；如果 m 的值超出目标类型的取值范围，则进行边界值截断。	m 值不变	m 值不变

较为常见的数据类型转换为 char 型和 short 型向 int 型的转换，主要用于多媒体图像或者音频视频等的处理，以及定点和浮点之间的转换，多用于科学计算等。本文针对 SLP 向量化，对向量因子长度进行分析，在引用点对齐与分析之后加入类型转换发掘模块，并结合向量混洗和交叉存取以及数据重组，以具有不同向量化因子的整型和浮点、浮点和双精度浮点数据为例，进行类型转换语句的 SLP 发掘。

2.2　数据重组

自动向量化中向量长度是固定的，不同的数据类型所占用的字节数不同，这决定了不同类型的数据在进行向量打包时每个向量中的存储情况也是不一样的。具有不同向量化因子长度的数据类型进行相互转换时，需要用到高位和低位的交叉存取和数据重组来保持变换前后数据的一致性。

当前的 SIMD 设备能够使用汇编指令直接运行或者通过自动 SIMD 变换间接执行。然而，这两种方式都面临着两个重要的约束：(1)SIMD 指令的寄存器到寄存器的属性使得向量操作必须被组织成适合于向量寄存器的形式来适应其执行；(2)SIMD 访存单元的限制，比如一些 SIMD 扩展设备只

能支持对齐的内存访问。尽管也有一些编译器能够对非对齐的内存访问进行向量化，如 Intel 的 SSE 系列编译器，但是也需要花费较大的性能开销。如果上述两种约束条件不能被满足的话，向量必须通过打包和拆包等操作来保证对齐，从而适应 SIMD 操作的应用，这增加了额外的开销。

(1)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：

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

一个更为简洁的混洗操作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：

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

(2)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打包成向量。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。

进行向量化发掘时，需要对复数数组的奇偶位进行交叉提取和打包，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结果向量中。本文中不同长度数据进行类型转换时，由于向量寄存器固定宽度的限制，不同数据长度的数据在打包时每个向量寄存器包含的元素个数不尽相同，需要在转换操作时交叉提取高位和低位并存储到目标向量寄存器中或者提取多个向量然后转换后交叉存储到目标寄存器的高位和低位。奇偶提取和高低位交叉存储的过程如图 2 所示。

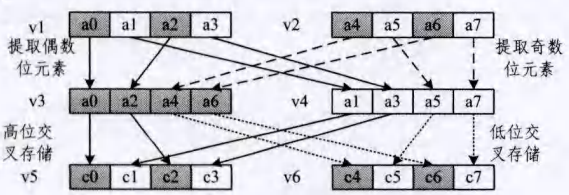

图 2　奇偶提取和交叉高位/低位存储操作

3　类型转换语句的 SLP 发掘

3.1　设计框架

在对程序进行自动向量化时，向量寄存器长度是固定的，不同数据类型所占用字节数的不同，决定了不同类型的数据在进行向量化时向量因子也不相同。本文在分析时设定向量寄存器长度为 128 位，以定点型(int，大小 32bits，向量化因子为 4)、浮点型(float，32bits，向量化因子为 4)和双精度浮点型(double，64bits，向量化因子为 2)的数据之间的转换为例进行分析，从循环展开、三地址化、SLP 发掘到代码生成进行分析，给出了相同向量因子长度和不同向量因子长度之间数据转换的 SLP 向量化发掘技术。

类型转换语句的 SLP 发掘过程如下：首先进行向量化识别和预优化工作；接着进行对齐分析和必要的对齐优化；然后遍历一个程序单元中的所有的循环套，对于每一个循环套，进行基本块识别，寻找出所有的可以对其进行优化的基本块；遍历各个基本块，对其进行一系列数据流预优化，建立语句依赖图，进行向量化发掘；处理完一个循环套中的所有基本块之后，进行向量化代码的生成。整体框架如图 3 所示。

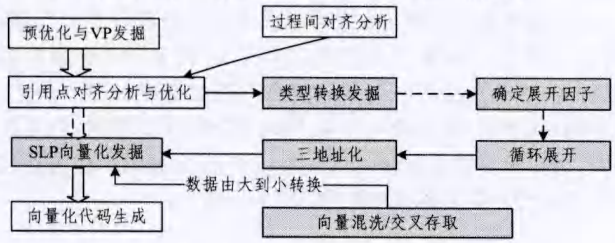

图 3　数据类型转换整体框架

3.2　相同向量因子

本小节阐述了相同向量因子的数据之间的类型转换的 SLP 发掘过程，在向量化因子确定之后，首先按照展开因子对循环进行展开，然后对展开的结果进行三地址化，接着进行 SLP 发掘。SLP 算法从数据相邻地址引用点开始发掘，最后进行代码生成。使用 SLP 对循环进行向量化时，需要一个固定的向量因子，向量因子通常在 SLP 向量化中作为展开因子使用。

下面以 int 型数据到 float 类型数据的转换为例详细叙述上述 SLP 向量化发掘过程，int 类型数据向量化因子为 4，float 类型数据向量化因子为 4，向量因子取为 4。

如图 4 中(a)所示的源程序，首先经过循环展开得到(b)所示的结果，由于展开因子为 4，在一次迭代中进行 4 次展开后的类型转换运算操作；然后对源程序进行三地址化，结果如图 4(c)所示；对三地址化后的结果进行 SLP 发掘，图 4 中所给的示例程序中数组 a 为 int 类型，int 类型的数组在一次向量读写操作中能处理 4 个元素，如图 5 所示的正向 SLP 发掘过程由 4 个 int 类型数据向两个 float 类型数据进行向下发掘。4 个 int 类型的数据(1,4,7,10)打成一个包，经过类型转换后得到由(2,5,8,11)组成的包，然后存储到目标寄存器中，完成整个 SLP 发掘过程；在 SLP 发掘和数据重组完成之后，进行代码生成，得到最终的向量化代码，如图 4(d)所示。

```
main()
{
  int a[512];
  float b[512];
  int i;
  for (i=0;i<512;i++)
    {
      b[i] = (float)a[i];
    }
}
```

(a) 源程序

```
main()
{
  int a[512];
  float b[512];
  int i;
  for (i=0;i<128;i++)
    {
      b[i] = (float)a[4i];
      b[i] = (float)a[4i+1];
      b[i] = (float)a[4i+2];
      b[i] = (float)a[4i+3];
    }
}
```

(b) 循环展开

```
main()
{
  int a[512];
  float b[512];
  int i;
  for (i=0;i<128;i++)
    {
      T1 = a[4i];
      T2 = (float)T1;
      b[4i] = T2;
      T3 = a[4i+1];
      T4 = (float)T3;
      b[4i+1] = T4;
      T5 = a[4i+2];
      T6 = (float)T5;
      b[4i+2] = T6;
      T7 = a[4i+3];
      T8 = (float)T7;
      b[4i+3] = T8;
    }
}
```

(c) 三地址化

```
main()
{
  int a[512];
  float b[512];
  vector.int va;
  vector.float vb;
  int i;
  for (i=0;i<128;i++)
  {
    simd_load(va,a+4i);
    vb=(vector.float)va;
    simd_store(vb,b+4i)
  }
}
```

(d) 代码生成

图 4　定点到浮点的类型转换

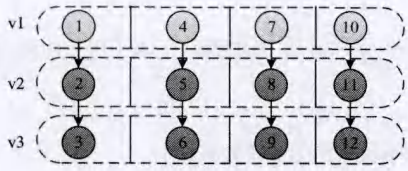

图 5　相同向量因子长度数据类型转换 SLP 发掘

3.3　不同向量因子

本小节阐述不同向量因子数据类型转换的 SLP 发掘过程，按照向量因子从小到大变换和向量因子从大到小变换分类进行讨论。整个的向量化发掘过程与 3.2 节中基本相同，基本过程同样分为循环展开、三地址化、SLP 发掘和代码生成。不同之处是由于涉及到数据长度不一致的数据类型的转换，在 SLP 发掘时需要利用数据重组对数据进行高低位交叉存取，进行 SLP 向量化打包和解包操作。

(1)向量因子从大到小变换

这部分以 float 类型数据向 double 类型数据转换为例进行详细描述，float 类型数据向量化因子为 4，double 类型数据向量化因子为 4，向量因子取 4。

对于数据类型由小到大变换，在向量化因子确定之后，其向量化代码生成发掘过程主要分为 5 步：循环展开和三地址化的过程与 3.2 节中相同向量因子长度类型转换过程类似。在 SLP 发掘阶段，对图 6 中所示的程序进行分析，数组 a 为 float 类型，float 类型的数组在一次向量读写操作中能处理 4 个元素，如图 7 所示的正向 SLP 发掘过程由 4 个 float 类型数据向两个 double 类型数据进行向下发掘。然后进行交叉存取和数据重组。图 7 所示的发掘过程中，4 个 float 类型的数据(1,4,7,10)打成一个包，通过交叉存取，把高位和低位分别取出打成两个 double 类型的数据包(2,5)和(8,11)，然后把这两个 double 类型的数据包存储到对应的 double 类型数组中，完成整个 SLP 发掘过程。最后进行代码生成，在 SLP 发掘和数据重组完成之后，得到最终的向量化代码，如图 6(d)所示，向量化代码中包含数据类型转换结果。

```
main()
{
  float a[512];
  double b[512];
  int i;
  for(i=0;i<512;i++)
    {
      b[i]=(double)a[i];
    }
}
```

(a) 源程序

```
main()
{
  float a[512];
  double b[512];
  int i;
  for (i=0;i<128;i++)
    {
      b[i] = (double)a[4i];
      b[i] = (double)a[4i+1];
      b[i] = (double)a[4i+2];
      b[i] = (double)a[4i+3];
    }
}
```

(b) 循环展开

```
main()
{
  float a[512];
  double b[512];
  int i;
  for (i=0;i<128;i++)
    {
      T1 = a[4i];
      T2 = (double)T1;
      b[4i] = T2;
      T3 = a[4i+1];
      T4 = (double)T3;
      b[4i+1] = T4;
      T5 = a[4i+2];
      T6 = (double)T5;
      b[4i+2] = T6;
      T7 = a[4i+3];
      T8 = (double)T7;
      b[4i+3] = T8;
    }
}
```

(c) 三地址化

```
main()
{
  float a[512];
  double b[512];
  vector.float va;
  vector.double vb1,vb2;
  int i;
  {
    simd_load(va,a+4i);
    vb1 = simd_getlow(va);
    vb2 = simd_gethigh(va);
    simd_store(vb1,b+4i);
    simd_store(vb2,b+4i+2);
  }
}
```

(d) 代码生成

图 6　float 到 double 的数据类型转换

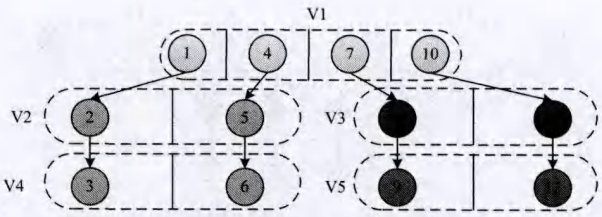

图 7　向量因子从大到小数据类型转换 SLP 发掘

(2)向量因子从小到大变换

这部分以 double 类型数据到 float 类型数据的转换为例进行详细的过程描述，double 类型数据向量化因子为 2，float 类型数据向量化因子为 4，向量因子取为 4。

对于数据类型由大到小变换，在向量化因子确定之后，其向量化代码生成发掘过程主要分为 5 步：循环展开和三地址化与 3.2 节类似。SLP 发掘阶段的难点在于 SLP 算法发掘是从数据相邻地址引用点开始的，对图 8 中所示的程序进行分析，数组 a 为 double 类型，double 类型的数组在一次向量读写操作中只能处理两个元素，这样导致如图 9 所示的正向 SLP 发掘过程无法正常地由两个 double 类型数据向 4 个 float 类型数据进行向下发掘。这里我们用图 10 所示的反向发掘方法解决这个问题。在 SLP 发掘之后进行数据重组和交叉存储。图 10 中所示的反向发掘过程中，由 V3 向前发掘时发现需要的数据为(1,4,7,10)，然而(1,4,7,10)中的数据是 double 类型，在进行读操作时并不能被打成一个 128 位的向量包，而是要分成两次向量读操作，然后进行类型转换，生成的两个向量通过交叉存储分别作为高位和低位存储到一个新的向量中。然后赋值给目标向量，完成整个 SLP 发掘过程。最后进行代码生成，在 SLP 发掘和数据重组之后，得到最终的向量化代码，如图 8(d)所示，向量化代码中包含数据类型转换结果。

```
main()
{
  double a[512];
  float b[512];
  int i;
  for(i=0;i<512;i++)
    {
      b[i]=(float)a[i];
    }
}
```

(a) 源程序

```
main()
{
  double a[512];
  float b[512];
  int i;
  for (i=0;i<128;i++)
    {
      b[i] = (float)a[4i];
      b[i] = (float)a[4i+1];
      b[i] = (float)a[4i+2];
      b[i] = (float)a[4i+3];
    }
}
```

(b) 循环展开

```
main()
{
  double a[512];
  float b[512];
  int i;
  for (i=0;i<128;i++)
    {
      T1 = a[4i];
      T2 = (float)T1;
      b[4i] = T2;
      T3 = a[4i+1];
      T4 = (float)T3;
      b[4i+1] = T4;
      T5 = a[4i+2];
      T6 = (float)T5;
      b[4i+2] = T6;
      T7 = a[4i+3];
      T8 = (float)T7;
      b[4i+3] = T8;
    }
}
```

(c) 三地址化

```
main()
{
  double a[512];
  float b[512];
  vector.double va1,va2;
  vector.float vb;
  int i;
  {
    simd_load(va1,a+4i);
    simd_load(va2,a_4i+2);
    vb=simd_vshuffle(va1,va2,mask)
    simd_store(vb,b+4i);
  }
}
```

(d) 代码生成

图 8　double 到 float 数据类型的转换

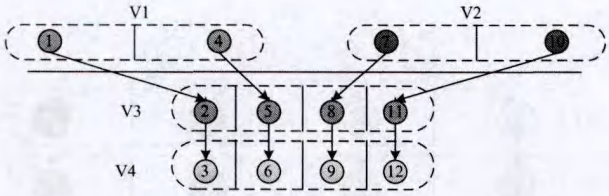

图 9　向量因子从小到大数据类型转换正向 SLP 发掘

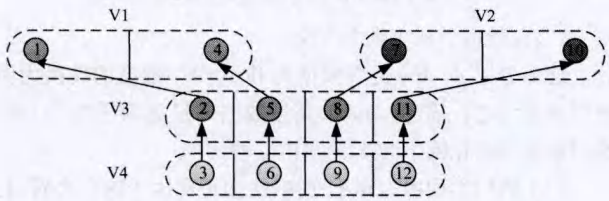

图 10　向量因子从小到大数据类型转换反向 SLP 发掘

3.4　有符号的类型转换

有符号和无符号数据类型进行相互转换时，需要考虑补集的情况，比如一个有符号的定点数值 $x=3$ 转换为无符号的数，其值为 $x'=3$；一个有符号的定点数值 $y=-3$，转换为无符号数，其值为 $y'=4294967293$。以 2.1 节中数据转换类型中的 signed integer to unsigned integer 转换为例，转换前数据类型位数为 M，数据值为 m，转换后数据类型位数为 N，需要分 3 种情况对转换结果进行考虑：

$M>N$ 时，如果 $m\geq 0$，值不变，如果 $m<0$，值为 2^N+m；

$M=N$ 时，如果 $m\geq 0$，值不变，如果 $m<0$，值为 2^N+m；

$M<N$ 时，如果 $m\geq 0$，值不变；如果 $m<0$，值为 2^N+m。

在不考虑数据溢出的情况下，上述 3 种情况其实可以归为一种情况考虑，即如果 $m\geq 0$，值不变；如果 $m<0$，值为 2^N+m。引入一个向量 $V=(2^N,2^N,\cdots,2^N)$，用这个向量和输入向量进行相加计算，得到的结果向量利用上面提到的相同向量因子和不同向量因子数据类型转换的方法进行变换，得到最终的数据类型转换结果。

3.5　算法实现

基于上文中的描述，本文用到的数据类型转换算法如下所示。

算法 1　数据类型转换算法

```
数据类型转换算法
输入:待转换数据位数 M,数据类型 T,数据值 X,转换后数据位数 N,
     转换后数据类型 P
输出:转换后数据值 Y
US={无符号数据类型集合};
SS={有符号数据类型集合};
If(T∈US and P∈US or T∈SS and P∈SS)
  If(M=N)
    //相同向量因子数据类型转换
    Y=X;
    Else if(M<N)
      //向量因子从大到小数据类型转换
    Y=X;
    Else
      //向量因子从小到大数据类型转换
    Y=X;
    End
End
Else
    Transform(有符号数,无符号数);//将有符号数统一转化为无符
    号数
    If(M=N)
      //相同向量因子数据类型转换
      If(m>=0)
        Y= X;
    Else
      Y=X+2^N;
      End
    Else if(M<N)
      //向量因子从大到小数据类型转换
      If(m>=0)
        Y=X;
      Else
        Y=X+2^N;
      End
    Else
      //向量因子从小到大数据类型转换
      If(m>=0)
        Y=X;
      Else
        Y=X+2^N;
      End
    End
End
```

4　实验结果及分析

4.1　实验平台

本文基于 Open64 实现了这种针对类型转换语句的 SLP 向量化算法，并在 IBM x3650 上进行实验，实验平台采用intel 至强处理器 5500，内存 4G。处理器的向量化寄存器长度为 128 位，可以同时处理 4 个 int 型数据或者 2 个 double 型数据。后端基础编译器采用 Intel C/C++ compiler (v11.0)，源程序首先经过基于 Open64 的向量化编译器将源程序转化为向量化程序，然后用基础编译器编译后在实验平台上运行。

本文提出的强制类型转换方法基于 Open64 开源编译器来实现，核心部分位于循环展开和三地址化之后代码生成之前的 SLP 发掘模块。

4.2　实验测试集

本文选择的 SPEC 2000 测试集中的数据压缩和面向对象数据库的程序包含有一定的数据类型转换工作。另外很多的多媒体音频或者图像处理程序以及通信中对校验和的计算，都需要用到频繁的数据类型转换，实验选取一个图像处理程序和一个通信中校验和计算的程序作为测试集的一部分。

Video. alphablending：图像处理程序，用于对可变透明度的图像进行混合操作。程序的内存流是 char 型数据，而一些处理操作用到的是 short 型，这样就要求从 char 到 short 类型数据的不断变换。

Tcp/ip. checksum：通信程序，用于检验接收到的网络数据包的完整性。对一个指针传来的长度为 4K 的地址中的 short 型的数据求其整型的校验和，需要用到 short 型到 int 型的数据类型转换。所选的测试程序如表 2 所列。

表 2　选取的测试程序

测试集	测试程序	程序功能
SPEC2000	bzip2	数据压缩
	vortex	面向对象数据库
	gzip	数据压缩
	Video. alphablending	图像处理
	Tcp/ip. checksum	通信校验和计算

4.3　实验结果及分析

(1)实验结果如表 3 和图 11 所示。

表 3　实验结果

测试程序	SLP 加速比	SLP+conversion 加速比	相对提升
Bzip2	1.03	1.07	4%
Vortex	1.02	1.05	3%
Gzip	1.12	1.24	11%
Video	1.56	2.53	62%
TCP/IP	1.83	3.49	91%

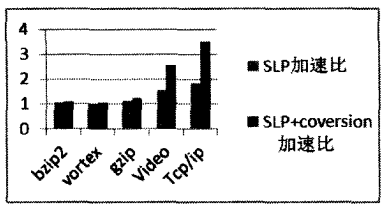

图 11　实验结果

(2)结果分析：

实验中 SLP 加速比表示只采用 SLP 向量化相对于源程序执行获得的结果，SLP+conversion 加速比表示采用本文的类型转换方法进行 SLP 向量化相对于源程序执行获得的结果，加速比的具体数值在表 3 中给出，其柱状图如图 11 所示。

SPEC 2000 中的图像处理程序 bzip2 程序以 ref 规模执行时，热点函数主要有 4 个，generateMTFValues 和 getAndMoveToFrontDecode 分别占到执行时间的18.13%和15.86%，sortIt 和 qSort3 的执行时间分别为12.42%和11.25%，其中包含类型转换操作的是函数 generateMTFValues 和 getAndMoveToFrontDecode。每个函数中有一个整型到无符号字符型的类型转换的循环，两个类型转换的总迭代执行次数约占这两个热点函数的 10%，另外两个热点函数中不包含数据类型转换，单独的数据类型转换向量化带来的加速比为 3，类型转换操作占整个程序执行的 3.4%，计算式为(18.13%+15.86%) * 10%=3.4%，实际测试的结果约为 4%。

Vortex 向量化后性能提升情况和 bzip2 类似，主要是因为向量化的循环在整个计算中所占的比例太低，对整个程序性能的提升不是特别明显。

Gzip 中的饱和算术循环基本被向量化了，其中包含类型转换的循环迭代执行次数约占整个程序执行总时间的 4%，相对于标量执行，单纯的类型转换语句向量化执行加速比大约为 3，因此对整个程序可以带来约 12%的性能提升，实际测试结果和理论结果基本吻合。

Video. alphablending 和 Tcp/ip. checksum 中数据类型转换占整个程序运行的比例很高，前者是典型的多面体音频处理程序，其中由于数字信号的密集存储和定点计算的特点，数据类型转换较为频繁，整个程序中核心函数部分包含大量的数据类型转换操作，程序整体向量化性能有着较大的提高。后者代码中核心语句就是一个数据类型转换操作，因此其加速效果十分明显。

结束语　本文针对程序中的类型转换语句进行了分析，归纳了数据类型转换的一些分类，结合向量混洗和交叉存取技术，提出了一种类型转换语句的 SLP 向量化发掘方法。文中以定点和浮点数据类型转换以及浮点和双精度浮点数据类型转换为例，详述了对具有不同向量化因子的数据类型之间进行转换的 SLP 发掘过程，实现了对不同长度的数据进行相互转换。实验中采用的文件压缩、图像处理和通信校验和计算等算例包含大量的数据类型转换，结果证明我们的方法能够有效地对其中的不同类型的数据进行转换，从而提高 SLP 向量化的效率。如表 3 所列，相对于单纯的 SLP 向量化，实验结果有 3%到 91%的加速提升，相对于源程序的加速比为 1.05到 3.49。

下一步的工作是准备对数据类型转换过程中的溢出处理和精度提升进行研究，并在本文的基础上丰富相应的框架，更进一步提升程序 SLP 向量化发掘的效率和向量化后程序运行结果的精确性。

参 考 文 献

[1] Allen J R, Kennedy K. Automatic Transformation of Fortran Programs to Vector Form[J]. ACM Transactions on Programming Languages and Systems, 1987(4): 491-542

[2] Zima H, Chapman B. Supercompilers for Parallel and Vector Computers[M]. ACMPress, 1990

[3] Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets[J]. Proceeding of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2000, 35(5): 145-156

[4] Rosen I, Nuzman D, Zaks A. Loop-Aware SLP in GCC [C]// Proceedings of the GCC Developers' Summit. Ottawa, Ontario, Canada, 2007

[5] Dai Y, Li Q, Zhang Q. SIMD-Efficient Loop Unrolling Design for Embedded Multimedia Applications[C]//IEEE International Conference on Multimedia and Expo. 2004: 1851-1854

[6] Wei Shuai, Zhao Rong-cai, Yao Yuan. Loop-Nest Auto-Vectorization Based on SLP[J]. Journal of Software, 2012, 23(7)

[7] Nuzman D, Rosen I, Zaks A. Auto-vectorization of interleaved data for SIMD[C]//PLDI. 2006

[8] Nuzman D, Zaks A. Outer-loop vectorization-revisited for short SIMD architectures[C]//PACT. 2008

[9] Barik R, Zhao J, Sarkar V. Efficient selection of vector instructions using dynamic programming[C]//MICRO. 2010

[10] Liu Jun, Zhang Yuan-rui, Ohyoung Jang, et al. A compiler Framework for Extracting Superword Level Parallelism[C]// PLDI'12. Beijing, China, 2012

[11] Wu Peng, Eichenberger A E, Wang A. Efficient SIMD Code Generation for Runtime Alignment and Length Conversion[C]// CGO. 2005

[12] Mi Wei, Li Yu-xiang, Chen li, et al. A Source to Source Translation Method with Type Restoration ib a Compiler[J]. Journal of Computer Research and Development, 2010(7): 1145-1155