SIMD数据置换操作的自动生成和优化

陈 向 沈 立 李家文
(国防科学技术大学计算机学院 长沙 410073)

摘 要 SIMD指令能够高效开发数据级并行,因此当前绝大多数通用微处理器都支持这种机制。但是应用程序和算法的一些固有特性,如访存地址不对齐、非连续存储访问以及控制流等,使得编译器或程序员必须借助置换指令重新组合向量的各个元素,才能得到符合SIMD指令要求的操作数。这些冗余的置换指令已成为当前挖掘数据级并行的主要性能瓶颈。提出一种自动的数据置换指令生成和优化算法,以有效地减少置换指令带来的性能损失。该算法基于提出的一种新中间表示形式,其中包含有足够的操作数地址信息,因此可以将置换指令的生成转换为数据流图中冲突边的识别问题,而将置换指令的优化转化为用最少的置换指令来删除所有冲突边的问题。面向一组典型多媒体程序进行测试的结果表明,提出的算法可平均获得7%的性能加速。

关键词 数据置换,中间表示,冲突边
中图法分类号 TP314 **文献标识码** A

Automatic Data Permutation Generation and Optimization for SIMD Devices

CHEN Xiang SHEN Li LI Jia-wen
(School of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract Nowadays, more and more general-purpose microprocessors provide enhanced SIMD instruction-set extensions to exploit data level parallelism. However, some inherent characteristics of applications and algorithms, such as memory address nonalignment, inconsecutive memory access and control flow, etc., make compilers or programmers have to use permutation instruction to reorganize the element of vectors to get correct operands for SIMD instructions. And these redundant permutation instructions had become the performance bottleneck of exploiting data level parallelism. This paper proposed an automatic data permutation generation and optimization algorithm. It can effectively reduce the performance loss caused by permutation instruction. The algorithm is based on a new intermediate representation, which contains enough address message of the operand, with which the problem of data permutation generation and optimization can be solved via identifying and eliminating all conflict edges in data flow graphs with minimal costs. The test result to a group of typical multimedia program shows that the algorithm can achieve performance acceleration up to 7% on the average.

Keywords Data permutation, Intermediate representation, Conflict edge

1 引言

为了更好地开发数据级并行,提高多媒体、信号处理等数据密集型应用的处理速度,几乎所有的现代高性能微处理器都提供了SIMD指令集扩展。而且随着集成度的提高,SIMD指令所能处理的数据宽度已经增加到了128位、256位[1],甚至512位[2]。

尽管很多研究工作表明SIMD扩展具有巨大的性能潜力[3-12],但是编译器自动生成的SIMD代码的实际效果仍然无法令人满意,主要原因是[4]:(1)SIMD指令都是寄存器-寄存器型的,操作数的宽度必须与向量寄存器的一致;(2)绝大多数SIMD访存单元仅支持地址连续、对齐(aligned)的存储访问,若实际应用无法满足这两个条件,则必须插入数据置换指令(Permutation),以生成符合SIMD指令要求的操作数[5]。

不同SIMD指令集提供的置换指令形式也不相同,本文以AltiVec指令集中的vperm[6]指令为例介绍我们的工作。指令vperm($V1$,$V2$,p)可以按照模式p从两个128位(4元素)的向量$V1$和$V2$中选取元素组成一个新的128位向量。Vperm指令通用性很高,其他置换指令可视作它的特例。

数据置换指令已成为当前SIMD应用的主要性能开销[7],表1给出了SIMD应用中各类指令的执行时间百分比。

从中可以看出,SIMD处理器在处理时有12%~18%的时间是用于数据访存,还有10%~17%的时间是用于数据的重新组织,而用于数据计算的时间仅有14%~28%。数据访

到稿日期:2010-06-13 返修日期:2010-10-29 本文受国家"973"重点基础研究发展计划(2007CB310901),国家自然科学基金项目(60803041)资助。

陈 向(1984—),女,硕士生,主要研究方向为高性能微处理器体系结构,E-mail:cxfenfei66@163.com;沈 立(1976—),男,博士,副教授,主要研究方向为高性能微处理器体系结构、高级编译技术等;李家文(1981—),男,硕士生,主要研究方向为高性能微处理器体系结构。

存和数据置换已成为整个 SIMD 应用的性能瓶颈。虽然人们已经提出了一些置换指令的生成策略，例如文献[8]中提出的4种流拼接方式，文献[9]在文献[8]的基础上对对齐分析提出了动态置换策略和流拼接策略等，但是这些方法都会产生冗余的置换指令，并且现有的优化策略不能删除这些冗余。因此，如何减少甚至消除这种由于冗余置换带来的开销，是一个值得研究的问题[7,10-12]。

表1 典型 SIMD 应用中各类指令的执行时间[13]

	VIS	MMX/SSE	AltiVec
数据存储指令	11.7%	21.0%	17.9%
数据组织指令	9.7%	12.6%	17.0%
数据运算指令	13.6%	18.8%	11.8%
浮点运算指令	/	9.3%	6.9%

图1(a)给出了一段标量C代码，图1(b)则是其向量化和优化后的结果。向量化后会产生大量置换指令 vperm(参见文献[4])，经过优化仍有6条剩余。但只要按照指令9和10中的模式操作数重新组织向量 T4 中的元素，就可以优化掉这两条指令，这说明现在的编译优化策略还有改进的空间。而且，冗余置换指令的存在也增加了编译器的复杂度。

```
// 标量C代码
1. for (i=0; i<4; i++) {
2.   t0[2i] = x[i] + x[i+4];
3.   t0[2i+1] = x[i] - x[i+4];
4. }
5. for (i=0; i<8; i++)
6.   t1[i] = T8[i] * t0[i];
7. for (i=0; i<2; i++) {
8.   for (j=0; j<2; j++) {
9.     t2[2j] = t1[2j+i] + t1[2j+i+4];
10.    t2[2j+1] = t1[2j+i] - t1[2j+i+4];
11.  }
12.  for (j=0; j<4; j++)
13.    t3[j] = T4[j] * t2[j];
14.  for (j=0; j<2; j++) {
15.    y[i+2j] = t3[j] + t3[j+2];
16.    y[i+4+2j] = t3[j] - t3[j+2];
17.  }
18. }
```

(a)

```
// 优化后的向量C代码
1. v1 = x[0:3] + x[4:7];
2. v2 = x[0:3] - x[4:7];
3. v3 = vperm(v1, v2, <0,4,1,5>);
4. v4 = vperm(v1, v2, <2,6,3,7>);
5. v5 = v3 * T8[0:3];
6. v6 = v4 * T8[4:7];
7. v7 = v5 + v6;
8. v8 = v5 - v6;
9. v9 = vperm(v7, v8, <0,4,2,6>);
10. v10 = vperm(v7, v8, <1,5,3,7>);
11. v11 = T4[0:3] * v9;
12. v12 = T4[0:3] * v10;
13. v13 = vperm(v11, v12, <0,4,1,5>);
14. v14 = vperm(v11, v12, <2,6,3,7>);
15. y[0:3] = v13 + v14;
16. y[4:7] = v13 - v14;
```

(b)

图1 包含冗余指令的向量代码

为了解决这个问题，提出了一种新的置换指令自动生成和优化算法。算法可以在当前任何一个编译框架下实现。我们的工作有以下3个特色：

1)提出了一种新的中间表示，其中包含有足够的操作数地址信息，无需再通过置换指令说明向量操作数各元素的地址，因而可以避免冗余置换指令的产生。第2节将对该中间表示进行介绍。

2)基于这种新的中间表示，提出了一种自动向量化算法，适用于循环和非循环的代码段。向量化过程中不会产生冗余的置换指令。第3.1节将介绍这种向量化算法。

3)将冗余置换指令的生成问题转换为检测向量代码数据流图(DFG)中是否存在冲突边，而将置换指令的优化问题转换为用最少的代价消除 DFG 中所有冲突边，提出并实现了相应的算法。第3.2节和第3.3节将详细介绍这些算法。

2 中间表示

向量化的过程中之所以会产生冗余的置换指令，是因为现有的中间表示只记录了标量操作数的存储地址，无法记录向量操作数中每个元素的存储地址。当不同向量中的元素被拼接成一个新向量后，这些元素的地址信息不会保存在新向量中，而只能通过产生该向量的置换指令的操作数重新计算出来。这既增加了置换指令优化的复杂度，也限制了优化的效果。

为此，我们向传统的中间表示中增加操作数的地址模式(address mode)，以记录操作数的地址信息。例如一个2输入、1输出的指令(标量指令或向量指令)可以表示为下面的七元组：

〈opcode,va,vb,vc,ma,mb,mc〉

其中，opcode 代表操作码，va 和 vb 是源操作数，vc 是目的操作数，ma,mb 和 mc 分别是这3个操作数的地址模式。

地址模式是一个位向量，指出一条指令要处理的元素在源向量中的下标，这里“源向量”表示被向量化的代码段中所有操作数组成的向量。图1中的 T8[0:7]就是一个源向量。

8位地址模式“11110000”表示一个128位的向量操作数中的4个元素来自256位源向量的前4个字，而模式“01010101”则表示128位向量操作数中的4个元素分别是256位向量的第2,4,6,8个字。图1(a)中标量指令6和图1(b)中的 SIMD 指令5可以分别表示为：

〈mul,t0[i:i+7],T8[i:i+7],t1[i:i+7],00010000,00010000,00010000〉

〈vmul,v3[i:i+7],T8[i:i+7],v5[i:i+7],11110000,11110000,11110000〉

有一些操作数的地址模式必须为空(none)，例如 vload(向量加载)和 vstore(向量保存)指令的操作数，以及另外一些特殊的操作数(移位操作数和置换模式)，因为编译器不会生成针对这些操作数的重组指令。

图2给出了图1(b)中C代码对应的 DFG，其中标出了每个结点中所有操作数的地址模式。由于这些操作数的元素都来自256位源向量(包括变量 x[0:7]和 y[0:7]，常量 T4[0:7]和 T8[0:7]，临时变量 t1[0:7]、t2[0:7]和 t3[0:7])，因此地址模式用8位位向量来表示即可。地址模式记录了指令中操作数的定义和使用情况，足以保证语义的正确性，因此在 DFG 中没有表示置换指令的结点。

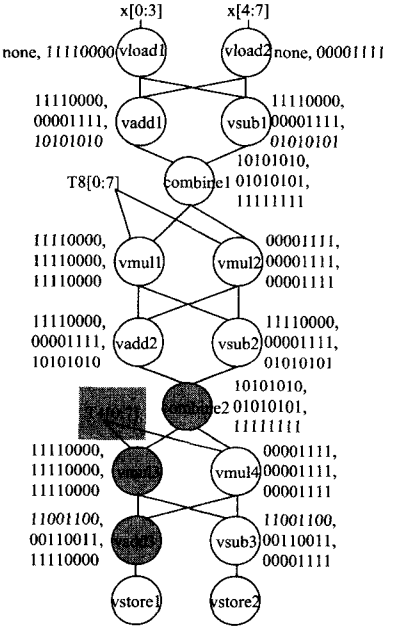

图2 包含完整地址模式的数据流图

通过对地址模式的分析很容易地检测出所有可能的数据置换。例如，图2中结点 vmul3 和 vadd3 之间存在真数据依

赖，vmul3 的目的操作数的地址模式是 11110000，而 vadd3 的第一个源操作数的地址模式为 11001100，不同的地址模式意味着这里需要插入一个置换指令，以便为 vadd3 准备源操作数。引入地址模式后，数据置换指令的生成和优化问题就被转换成如何在数据流图中识别并且消除所有冲突边。冲突边的引入使得编译器可以推迟一些置换指令的生成，从而避免了冗余置换指令的产生，但这绝非意味着编译过程中不会产生任何置换指令。

图 2 中的 Combine 结点是冲突边的一个特例。引进 Combine 结点的目的是产生源向量，如结点 Combine1 的目的操作数是源向量 t0[0:7]，Combine2 的是源向量 t2[0:7]。在代码生成时，每个 Combine 结点将被转换为一条或多条置换指令，置换指令的实际个数取决于 Combine 结点出边的条数。

冲突边和 Combine 结点的引入在向量化过程中也会产生置换结点。例如标量代码 y[2i]=b * c[2i+1]在向量化之后会变成如下形式 y[2i:2i+6:2]=vb * c[2i+1:2i+7:2]，常量 b 经过标量扩展得到 vb，该操作需要通过一条置换指令实现。不难发现，可以被描述为冲突边或 Combine 结点的置换指令都不会破坏存储器的数据一致性，而像标量扩展那样的置换指令是不可能被优化掉的。

3 置换指令的生成和优化

基于上节描述的中间表示，按照以下步骤生成和优化数据置换指令：首先进行向量化，主要是合并程序段中的同构语句，并根据需要进行循环展开，直至向量指令无法再容纳新的同构指令；然后标出 DFG 中所有的冲突边，并用最少的通用 permu 结点删除这些冲突边和 Combine 结点；最后进行代码生成，将 permu 结点转换为最少的 vperm 指令。接下来将详细描述这一过程。

3.1 向量化

以图 3(a)中的标量 C 循环为例说明向量化过程。向量化以基本块为单位进行，这些基本块可以是一段循环代码中最内层的循环体，或者是一段非循环代码中任何一个基本块。若基本块内含有条件分支，可以先通过条件转换(if-conversion)[14]将它们删除，然后进行向量化，当然其前提是处理器支持条件执行机制。

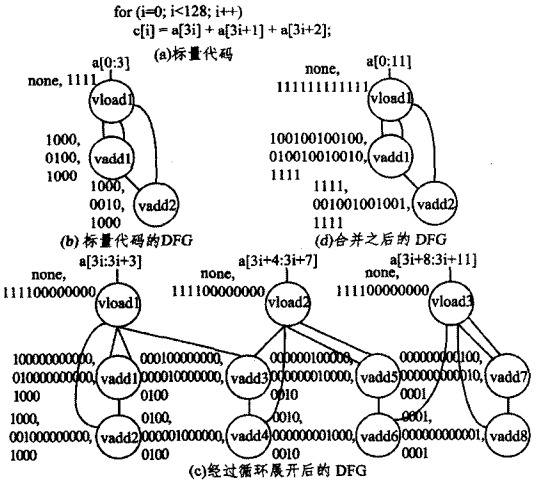

图 3 自动向量化

向量化的第一步是为基本块构建 DFG。除了一些特殊指令(例如修改循环索引变量的指令，因为它们不会被转换为向量指令)外，所有的标量指令都按照表 2 中的模板转换成 SIMD 形式。图 3(a)中循环体对应的 DFG 如图 3(b)所示，其中没有列出表征标量指令(例如循环索引增加的指令)的结点和边。

表 2 构建 DFG 模板

Instr. Type	Example	Templates	Description
Load/Store	Load R1, a[0]	⟨vload, V1, a[0:3], 1000, none⟩	Address mode 1000 means a[0]
	store R1, b[1]	⟨vstore, V1, b[0:3], 0100, none⟩	Address mode 0100 means b[1]
ALU	Add R1, R2, R3	⟨vadd, V1, b[0:3], c[0:3], 1000, 0100, 1000⟩	V1[0]=V2[1]+V3[0]
Logic	sll R1, R2, c	⟨vsll, V1, V2, c, 1000, 0100, none⟩	V1[0]=V2[1]<<c
	cmp R1, R2, 0	⟨vcmp V1, V2, 0, 1000, 1000, none⟩	V1[0]=V2[0]>0

注：寄存器 V1，V2 和 V3 分别保存向量 a[0:3]，b[0:3] 和⟨R3, R3, R3, R3⟩的内容，并假设这 3 个向量的存储地址是边界对齐的。

接下来，将 DFG 中的同构结点打包在一起，构成一个新的结点，即将循环体中同构的指令合并为一条 SIMD 指令。这里同构结点(或同构指令)是指那些有相同操作数并且完成同样操作的结点(或指令)。对于 AltiVec 指令集来说，至多 4 条同构的标量指令可以被打包在一起。两个同构结点能否被打包在一起，取决于它们操作的地址模式。例如指令⟨vadd, v1, v2, v3, 1000, 1000, 1000⟩和⟨vadd, v1, v2, v3, 0010, 0010, 0100⟩是同构指令，可以打包形成一条新的 SIMD 指令⟨vadd, v1, v2, v3, 1010, 1010, 1100⟩。

第三步，进行循环展开并重复上述两个步骤。若无法进行循环展开，则向量化结束。这一步应考虑的一个重要问题是地址模式的扩展。例如图 3(c)是将循环体展开 3 次后得到的 DFG，其中 3 个 vload 结点分别加载向量 a[3i:3i+3]，a[3i+4:3i+7]和 a[3i+8:3i+11]。为了将这 3 个地址连续的向量合并为一个源向量，必须将这 3 个操作数的地址模式从最初的 4 位扩展为 12 位，即将 1111 分别扩展成 111100000000，000011110000，000000001111，表示它们在源向量 a[3i:3i+11]中的位置。这 3 个同构的 vload 结点可以合并在一起，得到一个新的 vload 结点⟨vload, v1, a[3i:3i+11], 111111111111, none⟩。

在图 3(c)中还有另外两个同构的结点组，即{vadd1, vadd3, vadd5, vadd7}和{vadd2, vadd4, vadd6, vadd8}。图 3(d)展示了它们合并后的结果。通过这个例子可以看出这种新的中间表示能够很容易表示出同构语句组，因为在建立数据流图的过程中相同组中的同构语句已经被标识出来了。

在图 3(d)中可以发现一些冲突边。这里，边 e⟨I1, I2⟩是冲突边当且仅当结点 I1 的目的操作数和结点 I2 的至少一个源操作数有不一致的地址模式。按照传统的中间表示是不可能出现这种冲突的，不过这些冲突边都可以通过向数据流图中插入置换结点来消除。引入冲突边的目的是尽可能地推迟置换指令的生成。

3.2 冲突边的删除

为了以最少的置换指令删除 DFG 中所有冲突边，定义一

种通用的置换指令 permu。语句 $v_d = permu(v_1, v_2, \cdots, v_{n,p})$ 表示输入向量 $v_1, v_2, \cdots, v_n$ 中的元素按照模式 p 组合得到一个新向量 v_d。删除冲突边和 Combine 结点时，首先要确保引入最少的 permu 结点。具体步骤如下：

首先，删除 DFG 中的冲突边和 Combine 结点，将其转换成 permu 结点。permu 结点的所有输入向量可以通过其源操作数的地址模式获得，例如图 4(a)是图 2 中灰色子图转换之后形成的，其中子图“vector，permu→SIMD→permu”被转换为子图“permu，SIMD→SIMD→SIMD”。

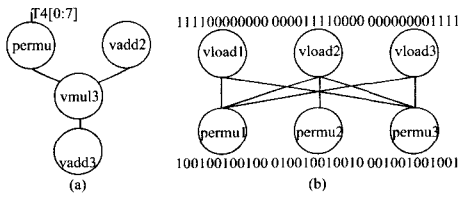

图 4　置换指令的产生和优化

其次，根据 load-store 功能单元的限制将原来加载或存储源向量的访存结点转换为实际的 vload 或 vstore 结点。例如，AltiVec 指令集提供 vload 和 vstore 指令来仅支持 128 位向量加载或存储，因此图 3(d)中的结点 vload1 应该被分成 3 个 vload 结点，并且所有从 vload1 出发的边应该用来自新结点的边进行替换。看起来，这一步只是简单地将向量化算法时合并在一起的同构结点 vload 或 vstore 结点重新拆分开，但这没有什么必要。实际上之前产生的源向量大大简化了向量化算法的实现和冲突边的识别。

图 4(b)给出了图 3(d)中所有冲突边被删除后的结果 DFG，新引入的 permu 结点数量是最少的。

3.3　代码生成

这一小节将详细描述代码的生成过程。首先根据上述 3.2 节提出的方法将 DFG 中的 permu 结点优化到最少，然后根据本文描述的算法(文献[7])拆分 permu 结点，并用 vperm 结点进行替换，最终使得 DFG 中包含的 vperm 结点最少。下面以图 5(a)中的标量 C 循环为例来说明。

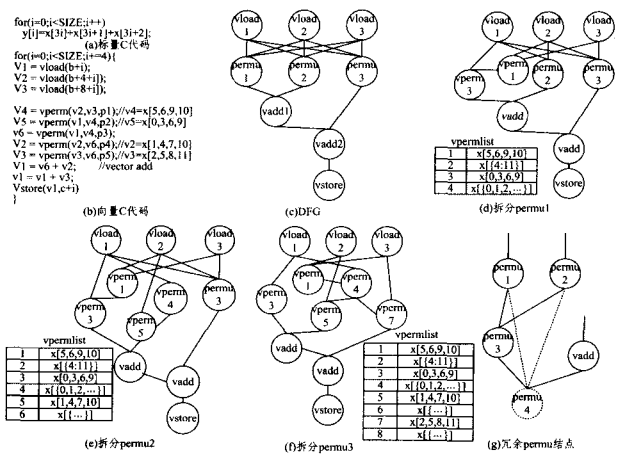

图 5　代码生成

首先对输入的每个基本块数据流图进行优化，消除其中的冗余 permu 结点。若一个 permu 结点的所有父结点都是 permu 结点，则它将作为被优化的对象。例如，图 5(g)中的 permu3 就是一个这样的结点。很容易看出，permu1 和 permu2 可直接作为 permu4 的输入，因此边 permu3－permu4 将被替换为两条新边 permu1-permu4 和 permu2-permu4。此外，还应根据 permu3 和 permu4 的模式得到 permu4 的新模式。接下来，结点 permu3 将被删除，因为它已没有任何出边。这一步将 DFG 中的 permu 结点数降到了最低。如果输入的程序中已经进行了优化且没有任何冗余的 permu 结点，这一步就可以不用进行。图 5(c)中没有任何冗余的 permu 结点。

接下来将每个 permu 结点拆分为一组 2 输入的 vperm 结点。这一步生成的 vperm 结点数应该尽可能地少，这是必须考虑的一个重要问题。在图 5(b)中，第三个 vperm 结点的结果将被作为最后两个 vperm 结点的输入，因此第二步结束后，图中应该只有 5 个 vperm 结点。

为了用最少的置换指令来替换 permu，引入一个 vperm-list 表格来记录每一步生成的 vperm 操作。显然，这一步开始时 vpermlist 为空。在拆分 permu 结点时，首先选出含有最少被打包元素数的两个输入结点，作为新生成 vperm 结点的输入。然后搜索 vpermlist，查找是否存在能为这两个输入产生相应输出的项。若有，则确定该项的输出并将其加入 DFG。否则，根据输入结点生成两个新项并加入 vpermlist。但这两个新项中，只有一个的输入是确定的，另一个的输入只能在之后被搜索到时确定。最后，从输入列表中删除这两个输入，并将新生成的 vperm 结点的输出加入 vpermlist。对于一个 n 输入的 permu 结点，上述过程应迭代进行，直到输入列表中只剩一项且已生成 $n-1$ 个 vperm 结点。

图 5(d)，5(e)和 5(f)演示了这一过程。结点 permu1 的输入列表为$\{x[0:3], x[4:7], x[8:11]\}$，在拆分 permu1 之前 vpermlist 为空。首先，输入列表的最后两个输入被选中，因为它们分别只含有一个被打包的元素。根据它们可以得到两个新 vperm 结点(vperm1 和 vperm2)并加入 vpermlist，其中 vperm1 的输出是 $x[5, 6, 9, 10]$，可以被直接加入 DFG。vperm2 的输入并不确定，将其表示为 $x[\{4:11\}]$，这里向量 $x[i, j, \{m, n, \cdots\}]$表示该向量的前两个元素分别为 $x[i]$和 $x[j]$，而它的后两个元素不确定。此时 permu1 的输入列表变为$\{x[0:3], x[5, 6, 9, 10]\}$。接下来，这两项都被选中，由于 vperm1 和 vperm2 都无法产生它们所需要的输出，因此得到两个新的结点 vperm3 和 vperm4。vperm3 被加入 DFG，vperm4 的表示变为 $x[\{0, 1, 2, 3, 5, 6, 9, 10\}]$。

permu2 和 permu3 与 permu1 的输入列表相同。为了拆分 permu2，首先选中输入 $x[0:3]$和 $x[8:11]$，vpermlist 中的 vperm4 可以为它们产生输出，因此可被直接加入 DFG，而 permu2 的输入变为$\{x[4:7], x[1, 10, \{0, 2, 3, 5, 6, 9\}]\}$。每当一个 vperm 结点被更新时，新的输出应该沿着该结点的出边被广播出去。接下来，又有两个新的结点产生，其中 vperm5 被加入 DFG，且其输出为 $x[1, 4, 7, 10]$。至于 permu3，输入 $x[0:3]$与 $x[4:7]$首先被选中，vpermlist 中的 vperm4 可以为它们产生输出。由于 vperm4 已经在 DFG 中，因此其输出可以被确定为 $x[1, 2, 5, 10]$。最后，vperm7 结点被生成并加入 DFG 中，就像 vperm3 与 vperm5 那样。

4　实验结果和分析

我们基于 SimpleScalar 3.0 工具链建立了模拟测试环境。该模拟环境与文献[15]中所用的相同，只不过去掉了其中关

于隐式数据置换(Implicit Data Permutation)的支持。模拟环境提供了32个128位的向量寄存器,支持AltiVec SIMD指令集,该指令集支持128位浮点和整数SIMD操作以及128位数据的访存操作。模拟器还为标量运算提供了32个标量寄存器。此外,假设SIMD操作与标量操作的延迟相同。

我们选择了7个多媒体基准程序进行性能测试,包括5个核心程序(C-Dot,C-Saxpy,R-FIR,FFT以及MT)以及两个完整的应用(adpcm,epic)。为了更好地进行比较,输入数据的地址是不对齐的。这些程序的实现算法中都包含由于非对齐存储访问和非连续存储访问引起的数据置换,前5个向量化的主体是循环,后2个既有循环也有非循环。

我们用3种方法完成这些基准程序的向量化和优化:(1)基于置换模式传播的优化;(2)本文提出的方法;(3)对方法(2)在编译过程中标识出的可向量化部分进行手工优化,前两种方法都在SUIF2.0上实现。这3种方法都会向代码中插入代表AltiVec SIMD操作的宏,最后用支持AltiVec的gcc编译为可在SimpleScalar上模拟的二进制代码。

表3对比了采用这3种方法后置换指令的数量,其中方法(3)可被视作最优的情况。因为经过仔细地手工优化,置换指令的数量已被减到最少。可以看出,对于除MM(矩阵乘)以外的基准程序,方法(2)均能够取得与方法(3)近似的效果。这主要是因为只有对矩阵进行合理的分块才能将MT所需的置换指令减到最少,而方法(2)没有涉及到对矩阵分块的处理。对于分块后的MM程序(MMB),方法(2)的效果也与方法(3)近似。而方法(1)没有专门针对置换指令的优化,不过由于它是在循环展开后再进行向量化,因此循环代码中的部分冗余置换指令也可以被优化掉。不过,对于adpcm和epic这两个应用,方法(1)生成的非循环向量代码中仍会产生较多的置换指令,这是它和方法(2)的主要差距。方法(2)和方法(3)对adpcm和epic的优化结果不一致,主要原因在于方法不能完全消除非循环代码向量化引起的冗余。表4给出了采用前两种方法得到的SIMD代码相对于初始标量代码的性能加速比,相对于方法(1),方法(2)能够得到平均7%的性能提升。

表3 3种方法生成的置换指令数量

	C-Dot	C-Saxpy	R-FIR	FFT	MM	MMB	adpcm	Epic
方法(1)	96	96	316	192	96	36	203	574
方法(2)	80	80	252	48	72	24	94	307
方法(3)	80	80	252	48	24	24	78	293

表4 前两种方法的性能加速比比较

	C-Dot	C-Saxpy	R-FIR	FFT	MMB	adpcm	Epic
方法(1)	2.57	3.25	2.08	2.99	2.10	1.08	1.07
方法(2)	2.63	3.33	2.16	3.11	2.19	1.12	1.09

结束语 SIMD扩展为有效开发数据级并行提供了很好的支持,但数据置换指令却限制了SIMD扩展能够带来的实际性能提升。为解除这一问题,提出了一种新的数据置换指令自动生成和优化算法。算法的基本思想是推迟冗余置换指令的产生,并将数据置换指令的产生和优化问题转换成如何识别并以最小的代价消除数据流图中的冲突边。我们提出的中间表示形式记录了指令操作数的地址信息,因而可以推迟冗余置换指令的生成;而引入了通用置换操作permu后使得最终得到的vperm结点数最少。测试结果表明,所提出的算法能够获得接近手工优化算法的性能加速比,效果十分理想。

参考文献

[1] Firasta N, Buxton M, Jinbo P, et al. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency[R]. Intel White paper, 2008

[2] Seiler L, Carmean D, Sprangle E, et al. Larrabee: a many-core x86 architecture for visual computing[J]. ACM Trans Graph., 2008, 27(3): 1-15

[3] Larsen S, Amarasinghe S. Exploiting superword level pararllelism with Multimedia Instruction Sets[C]// Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation(PLDI'00). 2000: 145-156

[4] Crescent Bay Software. VAST-F/AltiVec: Automatic Fortran Vectorizer for PowerPC Vector Unit[EB/OL]. http://www.psrv.com/vast altivec.html, 2004

[5] Lee R. Multimedia Extensions for General-purpose Processors [C]// 1997: Signal Processing Systems(SIPS). SIPS 97-Design and Implementation, 1997: 9-23

[6] IBM Corporation. PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual, version 2.06[S]. 2005

[7] Ren Gang, Wu Peng, Padua D. Optimizing Data Permutations for SIMD Devices[C]// Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation(PLDI '06). 2004: 118-131

[8] Eichenberger A E, Wu Peng, O'Brien K. Vectorization for SIMD architectures with alignment constraints[C]// Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation(PLDI'04). 2004: 82-93

[9] Wu Peng, Eichenberger A E, Wang A. Efficient SIMD Code Generation for Runtime Alignment and Length Conversion[C]// Proceedings of the International Symposium on Code Generation and Optimization(CGO'05). 2005: 153-154

[10] Shahbahrami A, Juurlink B H H, Vsaailiadis S, et al. Matrix register file and extended subwords: Two Techniques for Embedded Media Processors[C]// Proceedings of the 2nd ACM Int. Conf. on Computing Frontiers. 2005: 171-180

[11] Lee R B. Subword Permutation Instructions for Two-dimensional Multimedia Processing in MicroSIMD Architectures[C]// IEEE / Application-Specific Systems, Architectures, and Processors(ASAP'00). 2000: 3-14

[12] Naishlos D, Biberstein M, Ben-David S, et al. Vectorizing for a SIMD DSP architecture[C]// International Conference on Compilers, Architecture and Synthesis for Embedded Systems(CASES'03). 2003: 2-11

[13] Slingerland N T, Smith A J. Design and characterization of the Berkeley multimedia workload[J]. Multimedia Systems, 2002, 8(4): 315-327

[14] Shin J, Hall M, Chame J. Superword-level Parallelism in the Presence of Control Flow[C]// Proceedings of the International Symposium on Code Generation and Optimization(CGO'05). 2005: 165-175

[15] Huang Li-bo, Shen Li, Wang Zhi-ying, et al. SIF: Overcoming the Limitations of SIMD Devices via Implicit Permutation[C]// 16th International Symposium on High Performance Computer Architecture(HPCA'10). 2010: 355-366