面向SIMD扩展部件的循环优化研究

侯永生 赵荣彩 黄 磊 韩 林
(数字工程与先进计算国家重点实验室 郑州 450002)

摘 要 高性能微处理器中普遍采用SIMD向量扩展作为计算加速部件。在深入研究SIMD扩展部件数据依赖关系约束条件的基础上,提出一种基于依赖关系逆向图的Tarjan扩展算法,提高了SIMD并行性识别率,并结合传统向量化方法,实现了面向SIMD扩展部件的循环优化技术,消除了不可向量化语句对可向量化语句在数据重组中不必要的开销。实际程序测试结果显示,其在基于依赖关系的SIMD并行性判定方面优于ICC编译器,经过循环优化后,最终生成的SIMD代码其执行效率平均提高了12%。

关键词 SIMD,依赖关系,循环优化,Tarjan
中图法分类号 TP314 **文献标识码** A

Research on SIMD-oriented Loop Optimizations

HOU Yong-sheng ZHAO Rong-cai HUANG Lei HAN Lin
(State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450002, China)

Abstract Accelerating program performance via SIMD vector is very common in modern processors. Based on SIMD data dependency constraints, we presented the design and implementation of the improved Tarjan algorithm with reverse dependency graph to increase recognition rate of SIMD parallelization. And in combination with traditional vectorization method, the SIMD-oriented loop optimization technology was proposed that avoids unnecessary overhead in data reorganization. Our experimental results demonstrate that the performance of generated SIMD using the technology code is increased by 12% compared with ICC.

Keywords SIMD, Dependence analysis, Loop optimization, Tarjan

1 引言

随着高性能微处理中SIMD扩展指令功能日益完善,SIMD扩展部件逐渐从多媒体专用加速部件[4]转变为通用计算加速部件[3],在科学计算[12]、数字信号[11]、密码破译[10]等非多媒体应用领域得到广泛使用。SIMD扩展部件利用128或256位的SIMD寄存器对多个字符型、整型、浮点型数据同时进行相同操作,实现一种细粒度的数据级并行。

SIMD短向量编译优化与传统向量化技术[7,8]所解决的主要矛盾不同,前者主要解决由于SIMD扩展部件体系结构的访存限制造成程序性能下降的问题,例如数据引用的对齐分析[5]、内存访问[6]的连续性以及特殊SIMD指令优化等;后者关注点主要集中在利用各种循环变换使循环所携带的向量并行性最大化,这主要是由一次向量流水线操作代价很大所决定的。在实际工作中发现:传统向量化技术中利用各种循环优化充分挖掘循环并行性的优化策略对提高SIMD短向量编译优化效率具有重要借鉴意义。

本文融合传统向量化循环优化方法,提出一种在复杂依赖关系条件下面向SIMD扩展部件的循环优化技术。该技术通过引入语句依赖关系逆向图,并与Tarjan算法巧妙结合,将判定数据依赖关系环、语句重排、循环分布和循环融合有机地融合在一起,实现了语句SIMD向量化判断、SIMD语义等价转换和循环变换,并最终生成优化的循环分布策略的一体化优化流程,在充分挖掘循环中SIMD成份的基础上,实现有效降低不可向量化语句对可向量化语句的负面影响的优化目的。

2 问题提出

Intel编译器在优化图1(a)循环J时会输出"loop was not vectorized:existence of vector dependence"诊断信息,表明循环J含有阻碍向量化的依赖关系,无法进行SIMD向量化。我们通过仔细分析循环J的语句数据依赖关系图1(b),发现循环J虽然存在依赖关系,但可以通过循环优化将其消除掉,从而实现循环的完全SIMD向量化,具体过程如下:

(1)依赖环($S1$,$S2$,$S3$):可引入临时数组$h1$,将$S3$分裂为$S0$和$S3'$,从而消除阻碍向量化的依赖环;

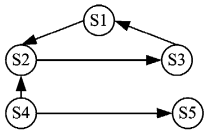
(2)反向依赖($S4->S2$):可将$S4$放置在$S2$前,将后向依赖转换为前向依赖。

到稿日期:2013-07-17 返修日期:2013-10-21 本文受"核高基"重大专项"支持国产CPU的编译系统及工具链"分课题"自动并行化与二进制翻译系统"(2009ZX10036-001-001-2)资助。

侯永生(1978-),男,博士生,主要研究方向为并行编译,E-mail:magichhou45@gmail.com;赵荣彩(1957-),男,教授,博士生导师,主要研究方向为体系结构和先进编译技术;黄 磊(1985-),男,硕士生,主要研究方向为并行编译;韩 林(1978-),男,副教授,主要研究方向为并行编译。

```
for(j=1; j<N; j++){
S1   h[j+1] = g[j] + 10;
S2   e[j-1] = h[j] + g[j];
S3   g[j] = h[j+2] + c[j] & 0x1100;
S4   f[j+1] = e[j] +20;
S5   b[j+4] = f[j] + 10;
}
```

(a)循环J



(b) 语句依赖关系图

```
for(j=1; j<N; j++){
S0   h1[j] = h[j+2];
S1   h[j+1] = g[j] + 10;
S4   f[j+1] = e[j] +20;
S2   e[j-1] = h[j] + g[j];
S3'  g[j] = h1[j] + c[j] & 0x1100;
S5   b[j+4] = f[j] + 10;
}
```

(c)循环优化后的循环J

图 1　ICC 无法向量的循环

上例说明现有的并行性判定方法并不适应 SIMD 扩展部件体系结构特点，本文在对 SIMD 功能部件依赖关系约束条件深入研究的基础上，借鉴传统向量循环优化方法，研究面向 SIMD 扩展部件的循环优化技术，解决了复杂依赖关系条件下的语句 SIMD 并行性判定和循环级优化问题，实现了消除不可向量化语句对可向量化语句的负面影响的优化目的。

本文第 3 节介绍 SIMD 扩展部件数据依赖关系约束条件；第 4 节介绍基于数据依赖关系逆向图的 Tarjan 算法；第 5 节对面向 SIMD 扩展部件的循环优化技术进行详细阐述；第 6 节对实验结果进行讨论；最后介绍进一步的工作。

3　SIMD 扩展部件数据依赖关系约束条件

SIMD 扩展部件由 SIMD 寄存器和 SIMD 扩展指令集组成，SIMD 寄存器的数据宽度（W_{simd}）一般是 128 比特或 256 比特，可以存放 4 或 8 个整型，2 或 4 个浮点型数据；SIMD 指令可以对 SIMD 寄存器中的多个标量数据进行相同的并行操作。由于 SIMD 寄存器数据宽度有限，需要将全局数据空间按照 SIMD 寄存器的宽度划分成多个长度相等的子数据块，我们将该数据块形成的数据空间称为短向量空间 SVS(Short Vector Space)。

目前大部分处理器芯片内部只设一条 SIMD 运算流水线，SIMD 指令以串行方式执行。如果将短向量空间作为基本数据类型，那么 SIMD 扩展部件在全局数据空间中的执行模式是串行；如果将整型、浮点型作为基本数据类型，SIMD 扩展部件在短向量空间内的执行模式是并行。我们将 SIMD 扩展部件这种全局串行、局部并行的执行方式称为“受限并行”(Limited Parallel)。这种特殊的执行方式使得 SIMD 扩展部件在数据依赖关系上呈现出两个特征，一是因为在全局数据空间内串行执行，SIMD 扩展部件能保留串行执行时所产生的前向数据依赖关系。二是短向量空间形成一个局部数据依赖空间，在该空间内数据之间不能存在任何数据依赖关系。根据上面两个特征可以得出结论：如果语句满足依赖关系是前向依赖或在局部数据依赖空间内无数据依赖这两者中的任意一个条件，该语句就具备 SIMD 并行性。

传统向量化技术只根据依赖环判断语句是否可向量化，这对于 SIMD 功能部件来说过于简单。本节将从依赖距离 D_d、依赖方向 D_r、依赖性质 D_w 和依赖环 D_c 4 个方面，按照上节得到的结论，研究 SIMD 扩展部件依赖关系约束条件。首先对依赖距离进行分析，如果依赖距离大于或等于 SIMD 寄存器宽度，可以保证在短向量空间内无数据依赖；如果依赖距离小于 SIMD 寄存器宽度，需要对依赖环进行进一步判断。在非依赖环情况下，前向依赖符合 SIMD 向量化要求，后向依赖可以通过语句重排转化为前向依赖；在依赖环情况下，需要对依赖方向和依赖性质进行综合分析。如果依赖环中后向依赖是读依赖(WAR)，可以通过节点分裂(Node Split)将后向依赖转化为前向依赖，从而消除依赖环实现 SIMD 向量化；如果依赖环中后向依赖是写依赖(RAW)，则无法消除依赖环实现 SIMD 向量化。

上节在分析 SIMD 扩展部件依赖关系判断标准的同时，也给出多个依赖关系条件的判断顺序：依赖距离→依赖环→依赖方向→依赖性质，图 2 是该判断过程示意图。在图 2 中判断标准将依赖关系状态空间分成 4 个部分，其中斜线部分表示在当前状态下，需要更多的条件进一步判断依赖关系的 SIMD 向量化特性，灰色部分表示不可向量化，白色部分表示可以向量化，图 2 中只有最右上角是灰色区域，我们将该区域的依赖关系性质称为 SIMD 扩展部件依赖关系约束条件。按照判断流程，将阴影部分依赖关系状态条件条件逐个进行与运算，即可计算出 SIMD 扩展部件依赖关系约束条件。

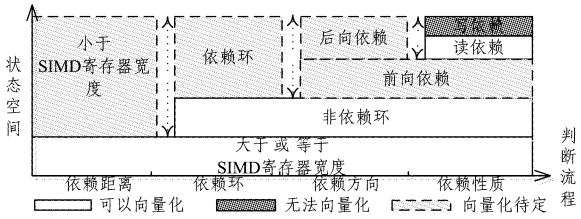


图 2　SIMD 依赖关系分析流程图

定义 1　如果一个依赖关系为 Dep，其中 Dep_c 表示与其它依赖关系形成依赖环，Dep_{cb} 表示 Dep_c 中的后向依赖，Dep_w 表示真依赖，Dep_d 表示依赖距离，则 SIMD 扩展部件依赖关系约束条件表示为 $RDep_{simd}$：

$$RDep_{simd}=(Dep_d<L_{SVS})\&Dep_c\&(Dep_b\&Dep_w) \quad (1)$$

4　基于数据依赖关系逆向图的 Tarjan 算法

4.1　Tarjan 算法

Tarjan 强连通分量算法[9]是一种基于图的深度优先搜索算法，它能够快速寻找并确定非强连通有向图中的最大连通分量(SCC)。该算法的基本思路是将一个未访问的节点压入栈中，然后判断它的后继节点是否在栈中，如果在栈中，表明发现一个强连通分量。在退栈过程中，通过最小编号来识别同一个 SCC 中的节点。由于栈先进后出的特性，导致节点的退栈顺序与拓扑顺序无法保持一致，这是传统循环优化技术中需要在此基础上构建凝聚图并对之拓扑排序的根本原因。

本文通过引入数据依赖关系逆向图，在原算法的基础上保留面向 SIMD 扩展部件循环优化所需的拓扑信息和依赖关系信息，形成基于依赖关系逆向图的 Tarjan 算法。

4.2　数据依赖关系逆向图

语句间的数据依赖关系是对同一内存位置 M 进行读写而产生的，假设语句 S1 先于 S2 对 M 同一内存位置进行读写，那么称 S1 为源点(Source)，S2 为汇点(Sink)。数据依赖关系图[7] $G=(S,\vec{D})$ 是表示循环中语句间数据依赖关系的有向图，$S=\{s_1,s_2,\cdots,s_n\}$ 是语句集合，其中节点下标值是对应语句在程序内的词典顺序。$\vec{D}=\{\vec{d}_1,\cdots,\vec{d}_m\}$ 是依赖关系集

合，$\vec{d}=(s_i \rightarrow s_j)$的方向是源点 s_i 指向汇点 s_j。

语句依赖关系逆向图 $\widetilde{G}$ 是将语句依赖关系图 G 中依赖关系 $\vec{d}$ 的方向由源点指向汇点改为汇点指向源点，即 $s_i \rightarrow s_j$ 转变为 $s_j \rightarrow s_i$。为了与语句依赖关系逆向图相对应，我们将原先依赖关系图称为语句依赖关系正向图，如图 3 中(b)是(a)依赖关系正向图所对应的依赖关系逆向图。

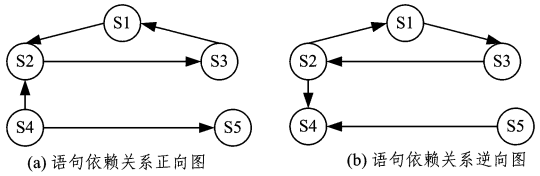


图 3　依赖关系逆向图

4.3　算法描述

该算法的基本思想是：利用基于依赖关系逆向图使 Tarjan 算法中节点压栈顺序与语句拓扑顺序相反，从而保证节点出栈顺序与语句拓扑顺序一致。图 4 展示了在图 3 中的正向图和逆向图下 Tarjan 算法中栈的变化情况和由退栈形成最终的语句序列。从图 4(b)中可以看出，由于遍历的是依赖关系逆向图，节点 S_5 先入栈而节点 S_4 后入栈，出栈时 S_4 先入 S_5 出栈，从而使节点出栈顺序与语句拓扑顺序一致，而图 4(a)最终生成的语句序列明显与语句的拓扑排序不一致。

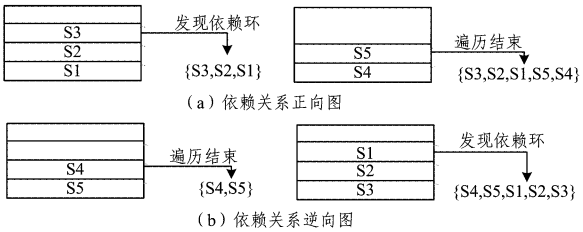


图 4　Tarjan 算法中栈变化情况示意图

根据依赖关系逆向图的特点和后续循环优化的需要，基于依赖关系逆向图的 Tarjan 算法对原 Tarjan 算法进行了如下改进：

(1)使用字典序对语句节点进行编号。原 Tarjan 算法按照压栈顺序对节点进行编号，这样做的优点是使算法更具有通用性，缺点是无法为后续循环优化提供必要的字典序信息。改进后的算法采用节点对应语句的字典序对节点进行编号，在节点中保留了后续循环优化所需的字典序信息。

(2)选择字典序最大的节点作为遍历起点。原 Tarjan 算法中没有字典序的概念，任选任意节点作为起始节点。选取字典序最大的节点作为起始节点有两个目的，1)保证字典序大的节点最先压入栈，最后出栈，使出栈顺序和字典序一致；2)可使后向依赖的源点先于汇点出栈，从而将后向依赖转化为前向依赖。

(3)确定强连通分量成员。通过分析栈中强连通分量节点分布特点可知，不同强连通分量中节点的 lowindex 值不相同，同一强连通分量中节点的 lowindex 值相同，大小等于根节点的字典序，且该值是唯一的。所以采用根节点的子典序作为强连通分量组成节点的判断标准。

(4)序列化强连通分量。原 Tarjan 算法没有字典序的概念，只能确定强连通分量的组成。在改进算法中对强连通分量遍历的起点是随机的，所以强连通分量的节点在栈内被根节点分为两个部分：字典序的后半部和字典序的前半部，需要将字典序的后半部分移到字典序的前半部分后面，生成一个正确的字典序序列，该过程称为强连通分量序列化。

基于逆向依赖关系图的 Tarjan 算法如图 5 所示。

```
//Input:  N in reservse dependence graph G=(N,D)
Func  RankNode(N)
  forall n in N do
        n.index   =     n.LexicalOrder;
        n.lowlink =     n.LexicalOrder;
  endfor
//input:  N in reservse dependence graph G=(N,D)
//ouput:  Start node v
Func  FindStartNode(N)
        v = max(s.LexicalOrder);
//input:  node list L
Func  Serialize(L)
  forall s in L do
        while (s.index != s.lowlink)
                L.Dequeue(s);
                L.Enqueue(s);
        endwhile
  endfor
//input:  start node v
//output:  node list L
Func  SerializeNode (v)
  St.push(v);
  forall (v, v') in  do
        If  v'. unvisited   then
                SerializeNode (v')
                v.lowlink = min(v.lowlink, v'.lowlink)
        elseif v' in St   then
                v.lowlink = min(v.lowlink, v'.index)
                If v == v' then v. LexicalOrder = 0;
      endif
  endfor
   // Is v the root of an SCC?
  if (v.lowlink == v' .index)
    repeat
      v' = S.pop
        L.add(v')
   until (v'.lowlink == v.index)
   Serialize(L);
  endif
// input:  reservse dependence graph G=(N,D)
//output:  node list L
Func  RTarjan()
        L.empty();
        Ranknode(N);
        while n. unvisited() in N do
                v = FindStratNode(N);
                L' =SerializeNode(v);
                L+=L';
                S=S-L;
        endwhile
```

图 5　基于逆向依赖关系图的 Tarjan 算法

1)初始化节点(RankNode)：计算节点 s 的字典序 LexicalOder，使 s. Index = LexicalOrder，s. Lowlink = LexicalOrder，其中 LexicalOrder 的值是节点的字典序。

2)确定遍历起始节点(FindStartNode)：从未遍历节点集合 S 中选取字典序最大的节点作为遍历起始节点。

3)序列化强连通分量 (SerializeNodes)：按照 Tarjan 算法对语句依赖关系逆向图进行遍历，并对强连通分量中的节点进行序列化，并将结果添加到队列 L'。在遍历过程中，如果强连通分量的根节点与结束节点，说明是自身构成强连通分量，将 LexicalOrder 置为 0，和其他强连通分量加以区别。

4)如果存在未访问的节点，返回(2)。

5　面向 SIMD 扩展部件的循环优化技术

面向 SIMD 扩展部件的循环优化技术根据 SIMD 功能部件数据依赖关系的特性，对循环依赖关系图进行优化，在此基础上，按照 SIMD 功能部件依赖关系约束条件，对基于依赖关系逆向图 Tarjan 算法输出的语句进行 SIMD 并行性判定，根据判断结果，在不改变串行语义的前提下，将可向量化语句与

不可向量化语句划分到不同的语句集合中，并根据分组结果进行循环分布。该技术不仅可以将后向依赖自动转换为前向依赖，还可消除不可向量化语句对可向量化语句在数据重组上不必要的程序开销。

根据 SIMD 扩展部件“受限并行”的执行方式可知，当数据依赖距离大于 SIMD 向量长度时，不会违背 SIMD 指令内并行执行语义，所以该类依赖关系可以安全地从依赖关系图中去掉，从而减少依赖关系的数量，实现依赖关系图优化。SIMD 向量长度(L_{simd})与 SIMD 寄存器数据宽度(W_{simd})和标量数据类型(D_{scalar})紧密相关，它们 3 者之间的关系是：$L_{simd}=W_{simd}/D_{scalar}$。根据上述公式计算出 L_{simd}，并与依赖距离 D_d 进行比较，若 $L_{vec}>D_d$ 或 $L_{vec}==D_d$，将该依赖关系从依赖关系图删除掉；若 $L_{vec}<D_d$，则保留该依赖关系。

按照优化后的循环依赖关系图生成循环依赖关系逆向图，把它作为基于依赖关系逆向图 Tarjan 算法的输入，寻找其中的依赖环。在该算法的输出队列中存在 3 类语句节点：自然节点、自身依赖节点和依赖环，其中自然节点是不与任何节点构成依赖环的节点，自身依赖节点是与自身构成依赖环的节点，依赖环是多个节点形成的强连通分量。通过节点 LexicalOrder 字段判断是否是自身依赖节点，利用 lowlink 是否相同来识别节点是否位于依赖环中，以下是上述 3 类节点是否可向量化的判断标准：

(1)自然节点：因为不与任何节点构成依赖环，该类节点可以向量化，所以该类节点是否可向量化的判断标准为：(s. LexicalOrder!=0)&&(s. lowlink!=s′. lowlink)，其中 s' 是 s 在语句节点队列中的后继节点。

(2)自身依赖节点：从 SIMD 扩展部件依赖关系约束条件可知，只有自身读依赖的节点才可以 SIMD 向量化，所以该类节点可向量化的判断标准是：(s. LexicalOrder==0) &&(D(s)is read))，其中 $D(s)$ 表示语句 s 所携带的依赖关系。

(3)依赖环：通过 SIMD 扩展部件依赖关系约束条件可知，当依赖距离小于向量长度时，只有后向依赖是读依赖的依赖环可以向量化。所以首先要确定环中的语句节点，从而定位依赖环的结束语句，再判断结束语句所携带的后向依赖的依赖关系是否是读依赖。若依赖环 DepCircle={sroot,…,send}，则该类节点可向量化的判断标准是：D(send) is read。

使用上述判断标准，节点被分为可向量化节点和不可向量化节点，isVecSt=1 表示可向量化，isVecSt=0 表示不可向量化。按照相邻且 isVecSt 相同的标准，将语句节点队列划分成多个语句集合，每一集合对应生成一个循环，将可向量化语句和不可向量化语句放置在不同的循环内。

在实际优化中，为了缩短 SIMD 优化的编译时间，利用 Annotation 机制为新生成的循环标示上可向量化循环和不可向量化标签，这样后端 SIMD 代码生成模块只对可向量化的循环进行分析。下面是面向 SIMD 扩展部件的循环优化步骤：

第 1 步　SimplifyDep()：对循环依赖关系图进行优化，删除不影响 SIMD 并行性的依赖距离大于向量化因子长度的依赖关系；

第 2 步　ConstructReverseDep()：根据第一步的优化结果，构建循环的逆向依赖关系图；

第 3 步　RTarjan()：寻找逆向依赖关系图中的依赖环，并生成语句节点队列；

第 4 步　JudgeVectoriable()：判断节点语句队列中语句是否可向量化，并对节点的 isVecSt 进行赋值；

第 5 步　PartitionNode()：对语句节点队列进行分组，将相邻的且具有相同 isVecSt 属性的节点划分到同一分组中。按照分组结果对循环进行循环分布，并根据子循环内语句 isVecSt 的值，将循环划分为可向量化循环和不可向量化循环。

面向 SIMD 扩展部件的循环优化算法如图 6 所示。

```
//Input:Dependenc Graph G=(N,D)
        vector length veclength
Func SimplifyDep(G)
  forall d ∈ D do
    if d.dis >= veclength  then
      G.delete(d);
//input:  Dependenc Graph G=(N,D)
//output: Reverse Dependence Graph G=(N,D)
Func  ConstructReverseDep(G,G)
        forall d ∈ D in G do
        d=(n,n’) --> d’=(n’,n);
        G.add(d’);
//input:state node list L
//output: state node list with annote La
Func JudgeVectoriable(L)
    p = L.firstnode;
    while p != NULL do
      judge p whether it could be vectorized using Table 4 ;
      if p is vectoriabe then p.isVec = 1;
        p=L.next;
    endwhile
//input:state node list with annote La
//ouput:Fors with the isVecFor annote
Func PartationNode(La)
  p = p’ = L.firstnode;
  while p != NULL do
    St.empty();
        do
        St.push(p’);
        p’ =p;
        p = L.next;
        while ( p.isVec != p’.isVec)
      generate For with node in St;
      if p’.isVec then annote For with isVecFor;
  endwhile
//Input:Dependenc Graph G=(N,D)
Func OptForSimd(G)
    SimplifyDep(G);
    ConstructReverseDep(G,G);
    L=Rtarjan(G);
    JudgeVectoriable(L);
    PartationNode(L);
```

图 6　面向 SIMD 扩展部件的循环优化算法

6　测试和结果分析

在开源编译器 Open64 基础上实现了本文描述的面向 SIMD 扩展部件的循环优化技术，并将其作为 SZ-VEC 向量化编译器的前端优化模块，实现语句的 SIMD 并行性判定和面向 SIMD 功能部件的循环优化，通过 Annotation 机制实现了与 SZ-VEC 后端 SIMD 代码生成模块连接。本节将从 SIMD 并行性判定和 SIMD 代码效率两个方面分别进行测试，在 SIMD 并行性识别率方面，采用 Intel 的 ICC 编译器[3]作为参照进行比较；在 SIMD 代码效率方面，以 SLP[1]算法为基础，测试采用面向 SIMD 扩展部件的循环优化技术后 SIMD 代码性能是否得到了提升。

测试用例采用开源 GCC 编译器自带的 SIMD 向量化能力测试包 GCC-VECT，该测试包主要用于测试编译器在非正规循环、数据类型、依赖关系、连续/跨步内存操作等情况下的向量化能力。测试环境：Xeon(TM) CPU 2.40GHz，2G 内存，向量寄存器宽度为 256 位，操作系统：Red Hat Enterprise V5，编译环境：ICC v11.0 编译器[2]。

根据测试目的，采用 GCC-VECT 测试包中依赖关系比较复杂的 237 个循环来测试 SIMD 并行性识别率，这些循环基本上涵盖了依赖环、前/后向依赖、读/写依赖等单独或组合的情况，能够较为全面地测试出编译器的 SIMD 并行性识别能力。在分析结果中，使用单层循环为计数单位，用识别出的 SIMD 并行性循环个数与测试循环总数的比值计算出 SIMD 识别率。识别出 SIMD 并行性的循环存在两种情况，1）被 ICC 和 SZ-VEC 同时识别出 SIMD 并行性的循环，称为一类循环；2）只被 ICC 或 SZ-VEC 识别出 SIMD 并行性的循环，称为二类循环。

从表 1 测试结果可知，ICC 识别出的二类循环为 23 个，SZ-VEC 识别出的二类循环为 38 个。通过仔细分析 ICC 和 SZ-VEC 二类循环，发现 ICC 能够实现结构体、指针、多维数组等复杂数据结构情况下的 SIMD 并行性判定，同时对部分非结构循环，例如 while、goto、if 语句，也能判断其 SIMD 并行性，而 SZ-VEC 在这两方面的分析能力要弱于 ICC。但在真依赖环判定、后向依赖分析、依赖距离计算等方面要优于 ICC。综上所述，ICC 的分析能力在全面性上要强于 SZ-VEC，而 SZ-VEC 在面向依赖关系的 SIMD 并行性判定方面要优于 ICC。由于在我们选择的测试用例中数组和正规循环占大多数，因此在总的 SIMD 并行识别率上，SZ-VEC 要高于 ICC。

表 1　SIMD 识别率对比表

测试结果 / 编译器	已识别		未识别	识别率
	一类	二类		
Intel C++	105	23	109	54%
SZ-VEC		38	94	60.3%

下面对自动生成的 SIMD 代码的性能进行测试。测试用例是 SZ-VEC 识别出 SIMD 并行性的 143 个循环，测试方法通过屏蔽和开启前端循环优化模块，在后端采用 SLP 算法[1]完成 SIMD 代码生成，图 7 是测试结果，其中性能未变化的情况占 39%，共 69 个循环；性能提高的情况占 57%，共 70 个循环；性能下降的情况占 4%，共 4 个循环。图 7(b)是程序平均运行时间，其中性能提高情况下比原来提高了 12%，性能下降情况下比原来下降了 18%。

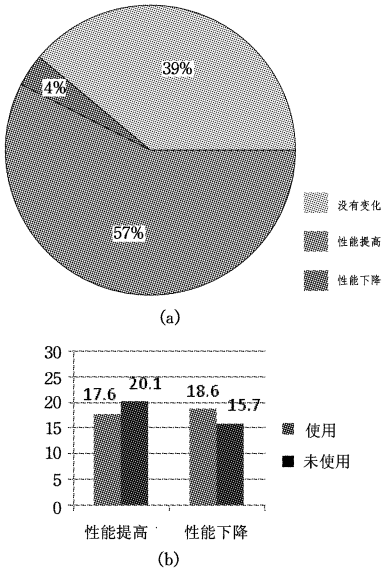


图 7　SIMD 优化后性能对比图

分析性能提高循环的 SIMD 代码发现，本文中的循环优化技术可以有效避免不可向量化语句对程序性能的消极影响，例如图 8(a)的循环 I，其中 S1 可以向量化，而 S2 具有自身写依赖，无法向量化，且(S1，S2)存在正向的真依赖关系(对数组 $b[i]$先写后读)。图 8(b)是未使用循环优化技术情况下 SLP 算法产生的 SIMD 代码，其中对应 S2 的向量化代码实际上是串行执行，每次使用 simd_ srlow 指令从 SIMD 寄存器中取出本次运算的标量，然后进行运算，将结果保存在临时整型变量内，最后将这 8 次的运算结果以向量赋值的形式更新数组 b 的内容。图 8(c)是经过循环分布优化后，生成的 SIMD 代码，指令个数明显减少。

```
for(i=8;i<N;i++){
    S1  b[i] = a[i]+4 ;
    S2  c[i-1] = b[i-1]+c[i] ;
}
```
(a) 循环I

```
_v_1_preg_w2c = simd_set_intv8(4,4,4,4,4,4,4,4)
for(i=1; i<N/8; i++){
L1   simd_load(_v_0_preg_w2c_0,a+(i*8)*4);
L2   _v_2_preg_w2c_0 = _v_0_preg_w2c_0 + _v_1_preg_w2c ;
L3   simd_store( _v_2_preg_w2c_0,(b+(i*8)*4));
L4   simd_loadu(_v_3_preg_w2c_0,(b+i*8-1)*4);
L5   simd_loadu(_v_4_preg_w2c_0,(c+i*8-1)*4);
L5   _SLP_p44_preg = (_INT32)(_v_4_preg_w2c_0)
L6   _SLP_p43_preg = (_INT32)(_v_3_preg_w2c_0) ;
L7   _SLP_p42_preg =  _SLP_p43_preg* _SLP_p44_preg;
L8   _SLP_p38_preg =  simd_srlow(_v_4_preg_w2c_0,32U);
L9   _SLP_p37_preg =  simd_srlow(_v_3_preg_w2c_0,32U);
L10  _SLP_p36_preg =   _SLP_p37_preg * _SLP_p38_preg ;
 ::          ::                  ::
 ::          ::                  ::
L29  _v_7_preg_w2c_0 = simd_set_intv8(_SLP_p42_preg,
                                    _SLP_p36_preg,
                                    ... ...          );
L30   simd_store(_v_7_preg_w2c_0,c+(i*8)*4 );
}
```
(b) 未经优化后生成的SIMD代码

```
_v_1_preg_w2c = simd_set_intv8(4,4,4,4,4,4,4,4)
for(j=1; j<N/8; j++){
L1   simd_load(_v_0_preg_w2c_0,a+(i*8)*4);
L2   _v_2_preg_w2c_0 = _v_0_preg_w2c_0 + _v_1_preg_w2c ;
L3   simd_store( _v_2_preg_w2c_0,(b+(i*8)*4));
}

For(i_preg=8; i_preg <N; i_preg++){
L4   c[i_preg-1] = b[i_preg-1] + c[i_preg];
}
```
(c) 经过优化后生成的SIMD代码

图 8　SLP 算法向量化循环

在分析性能下降的情况中发现，循环优化会带来额外的循环开销，也会抵消或超过 SIMD 指令带来的收益，造成性能出现负加速情况。该类情况虽然出现较少，但对性能影响较大，下一步需要针对该类情况在代价模型、收益分析等方面进行深入研究。

7　相关研究

面向 SIMD 向量化部件的循环优化技术随着 SIMD 功能部件的广泛使用得到了越来越多的关注[5,6,13,14]，大部分研究主要集中在编译器后端优化，即 SIMD 代码生成部分[1,16,18]。S. Larsen 和 S. Amarasinghe[1] 提出了一种循环展开-压缩方法，该方法将 SIMD 代码生成转化为无依赖关系指令的排列组合问题，但因为缺乏高层循环依赖关系信息，导致生成大量无用指令。D. Nuzman[16] 针对非连续交叉内存引用操作总结出 3 种交叉访问模式并采用虚拟向量的方法完成 SIMD 代码的生成，他的方法在非连续引用变量较多时无法得到最佳性能。

基于循环交换的嵌套循环向量化技术[7,8]在面向传统向量机优化中可以取得明显的效果，例如 Cray。通过循环交

换，可以将计算量大的循环发在最内层，从而提供更好的向量性。对 SIMD 功能部件来说，该方法忽略了不可向量化语句（例如不支持的 SIMD 运行类型）的负面影响，该文中介绍的循环优化方法不仅可以作为 SIMD 编译优化的预处理部分，将依赖关系信息以注视的方式传递给后端 SIMD 生成部分，降低非对齐变量的个数，还可以作为循环交互的补充，将不可向量化语句的负面影响进行综合考虑，提高 SIMD 优化效果。

K. Trifunovic 和 D. Nuzman[20] 利用多面体模型计算 SIMD 优化开销，从而指导 SIMD 优化方案，一个循环被抽象为多面体模型的条件比较苛刻，例如未知依赖关系、函数调用等等，从而限制了该优化方法的应用范围。本文中提到的 SIMD 优化方法可以有效消除函数调用、控制流等影响多面体模型的条件。

结束语 本文首先对 SIMD 功能部件执行方式的特性进行深入分析，提出了“受限并行”的概念，并在此基础上形成了 SIMD 功能部件数据依赖关系约束条件，从而为判定语句的 SIMD 并行提供了理论依据。通过分析 Tarjan 最大强连通分量识别算法在语句 SIMD 判定性上的不足，设计了一种基于依赖关系逆向图扩展 Tarjan 算法，其提高了编译器在识别循环 SIMD 在并行性成分的能力。针对 SLP 算法在代码生成时没有考虑到不可向量化语句对 SIMD 代码性能的负面影响，借鉴传统向量化方法，将依赖环判定、语句重排和循环分布融合在一起，实现了面向 SIMD 功能部件的循环优化技术。通过测试结果证明，在解决复杂依赖关系情况下 SIMD 在并行判定以及消除不可向量化语句对 SIMD 代码性能的负面影响方面具有明显优势，同时也发现该优化技术还有以下待改进之处：

(1)现实程序情况复杂，非结构循环、结构数据、多维数组、指针等情况比较普遍，但现在只能处理结构化循环内数组情况，无法处理以上复杂情况。针对以上不足，需要在依赖关系分析时考虑依赖层次、别名分析等问题，同时为了弥补编译器在静态分析时的不足，还需要引入交互调优的手段，用户向编译器提供额外程序信息，协助编译器完成复杂情况下的循环优化，目前该方法还在进一步研究中。

(2)在已实现的优化过程中没有考虑代价/收益比，造成过度优化情况的发生，导致出现了 SIMD 代码负加速现象。循环优化会不可避免地带来额外开销以及破坏其它优化机会的条件，例如寄存器优化、指令优化等。在现有研究成果基础上，下一步将从循环工作量、额外开销、数据重组、混合并行等方面综合考虑，建立 SIMD 功能部件的代价模型，以便更准确、更全面地评估循环优化的优化效果。

参 考 文 献

[1] Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets[C]// Conference on Programming Language Design and Implementation. Vancouver, BCCanada, June 2000: 145-156

[2] Intel. IA-32 Intel Architecture Optimization Reference Manual [OL]. http://www. intel. com/products/processor/manuals/index. htm, 2005

[3] Franchetti F, Kral S, Lorenz J, et al. Efficientutilization of SIMD extensions[J]. Proc. IEEE, 2005, 93(2): 409-425

[4] Al-Shayka O, Chen S, Mattavelli M. Introduction to the special issue on multimedia implementation[J]. IEEE Transactions on Circuits and System for Video Technology, 2002, 12(8): 629-632

[5] 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. March 2005: 153-164

[6] Wu Peng, Eichenberger A E, Wang A, et al. An integrated simdization framework using virtual vectors[C]// Proceedings of the 19th Annual International Conference on Supercomputing. June 2005: 169-178

[7] Kennedy K, Allen J R. Optimizing compilers for modern architectures: a dependence-based approach[M]. San Francisco CA, USA: Morgan Kaufmann Publishers Inc, 2002

[8] Bacon D F, Graham S L, Sharp O J. Compiler Transformations for High-Performance Computing[J]. ACM Computing Surveys, 1994, 26(4)

[9] Tarjan R E. Depth-first search and linear graph algorithms[J]. SIAM Journal on Computing, 1972, 1(2): 146-160

[10] Fournier J J A, Moore S W. A vector approach to cryptography implementation[M]// Digital Rights Management. Technologyies, issues, challenges and systems. Springer Berlin Heidlberg, 2005: 277-297

[11] Franchetti F. A portable short vector version of FFTW [C]// Proc. Fourth IMACS Symposium on Mathematical Modeling (MATHMOD 2003). 2003, 2: 1539-1548

[12] Slingerland N T, Smith A J. Measuring instruction sets for general purpose microprocessors: A survey[R]. CSD-00-1122. University of California at Berkeley Technical Report, 2000

[13] Shin J, Hall M, Chame J. Superword-level parallelism in the presence of control flow[C]// International Symposium on Code Generation and Optimization. 2005: 165-175

[14] Feld D, Soddemann T, Jünger M, et al. Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Automatic Code-Transformation[C]// Proceeding of the 3rd International Workshop on Polyhedral Compilation Techniques. 2013: 45-54

[15] Eichenberger A, Wu P, O'Brien K. Vectorization for simd architectures with alignment constraints[C]// PLDI. 2004

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

[17] Kong M, Veras R, Stock K, et al. Polyhedral Transformations Meet SIMD Code Generation[C]// PLID. I2013

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

[19] Chen C, Chame J, Hall M. Chill: A framework for composing high-level loop transformations[R]. Technical Report 08-897. USC Computer Science Technical Report. 2008

[20] Trifunovic K, Nuzman D, Cohen A, et al. Polyhedral-model guided loop-nest auto-vectorization[C]// PACT. Sept. 2009

[21] Vasilache N, Meister B, Baskaran M, et al. Joint scheduling and layout optimization to enable multi-level vectorization [C]// Proc. of IMPACT'12. Jan. 2012