基于条件分类的控制流向量化

孙回回 赵荣彩 高 伟 李雁冰
(解放军信息工程大学 郑州 450001) (数学工程与先进计算国家重点实验室 郑州 450002)

摘 要 现代编译器越来越依赖SIMD指令来提高向量化性能,但控制流的复杂性严重阻碍了SIMD向量化的发掘。现有的控制流向量化方法对于单层控制流的向量化很有效,但对嵌套等复杂控制流无法取得令人满意的效果。因此,提出了一种基于条件分类的控制流向量化方法。该方法对条件为循环不变量的控制流,以层次遍历的顺序实施IF外提;对条件为循环变量的控制流,结合语句匹配和条件合并递归地进行IF转换,生成相应的SIMD指令,从而实现嵌套控制流的向量化。实验结果表明,该方法能够有效消除循环中的嵌套控制流,提高向量化发掘的能力,有效提升测试程序的性能。

关键词 控制流,SIMD向量化,条件分类,IF外提,IF转换
中图法分类号 TP314 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.11.049

Control Flow Vectorization Based on Conditions Classification

SUN Hui-hui ZHAO Rong-cai GAO Wei LI Yan-bing
(PLA Information Engineering University, Zhengzhou 450002, China)
(State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450002, China)

Abstract Modern compilers increasingly rely on SIMD instructions to improve the performance of vectorization. However, the complexity of control flow seriously inhibits the exploitation of SIMD vectorization. Current vectorization methods of control flow are proved effective for single-level control flow, but attain poor performance for nested control flow. Hence, a method of control flow vectorization based on conditions classification was presented. Loop Unswitching is applied in the order of level traverse if the condition of the control flow is loop invariant. When the condition of the control flow is loop variant, IF conversion is implemented recursively combined with statement matching and condition join. Then SIMD instructions are generated correspondingly and vectorization of nested control flow is realized. The experimental results show that the proposed method can efficiently remove the nested control flow from the loops and promote the ability of vectorization. Effective acceleration is achieved for the test applications.

Keywords Control flow, SIMD vectorization, Conditions classification, Loop unswitching, IF conversion

1 引言

如今,越来越多的处理器集成了SIMD(Single Instruction Multiple Data)扩展部件。SIMD扩展部件最初仅用于多媒体领域,后来人们将SIMD扩展部件应用在数字信号处理器和高性能计算机中。SIMD通过并行执行一条指令的多个数据,可以使应用程序实现并行处理。基于SIMD扩展部件的向量化已成为程序并行的重要手段。然而,SIMD程序的手工实现非常复杂,而且容易出错,有时还会破坏代码的灵活性,因此自动向量化已经成为开发面向SIMD部件的向量化程序的主要途径。目前主流的自动生成SIMD指令的方法有两种,一种是面向循环的传统向量化方法[1],另一种是面向基本块的SLP(Superword Level Parallelism)向量化方法[2]。不管是传统向量化还是SLP,存在控制流的向量化一直是并行化研究中的重点和难点。

控制流在高性能计算应用课题中十分常见,它的存在严重阻碍了向量化的发掘。若能在向量化前将控制流消除或转换为简单的数据依赖,则将大幅增加向量化的机会,编译器的优化性能也将因此得以提升。Cocke和Allen提出的IF外提(Loop Unswitching)技术,通过将条件为循环不变量的控制流外提到循环外来减少或消除循环内的控制流[3]。GCC编译器引进了IF外提技术,但仅针对最内层循环中的简单控制流结构,而且该技术只适用于条件为循环不变量的控制流分支。针对控制流,另一类有效的编译优化方法是IF转换。IF转换由Allen在文献[4]中提出,其主要思想是将所有的控制依赖转换为数据依赖,从而把所有依赖的形式统一化。IF转换已被广泛应用于现代编译器中,诸如GCC和Open64等都编写了自己的IF转换模块,但是目前的IF转换方法不能有效地对嵌套控制流进行向量化。

为解决上述问题,实现任意复杂嵌套控制流的向量化,本

到稿日期:2014-11-08 返修日期:2015-01-25 本文受"核高基"国家科技重大专项(2009ZX01036)资助。
孙回回(1991-),女,硕士生,主要研究方向为先进编译技术,E-mail:sun_hui_hui@163.com;赵荣彩(1957-),男,博士,教授,主要研究方向为高性能计算与先进编译技术;高 伟(1988-),男,博士生,主要研究方向为先进编译技术;李雁冰(1989-),男,博士生,主要研究方向为先进编译技术。

文提出了一种基于条件分类的控制流向量化方法。根据控制条件与循环迭代之间的关系，将控制流条件变量分成两类，即循环不变量及循环变量。针对条件变量是循环不变量的控制流，在 Cocke 等人 IF 外提技术的基础上，利用层次遍历的方法增强 IF 外提的能力，实现嵌套控制流的 IF 外提，从而消除循环内的控制流，实现向量化；针对条件变量是循环变量的情况，首先对代码进行预分析，结合语句匹配和条件合并对有向量化收益的代码进行 IF 转换，在转换为 SELECT 语句后进行向量化并生成相应的 SIMD 指令，最后对未能成功向量化的 SELECT 语句进行 IF 重构，以避免 IF 转换带来的程序性能降低的问题。测试结果表明，用本文提出的方法能够有效消除循环中的控制流，提高向量化发掘的能力，从而大幅提升程序性能。本文的主要贡献如下：

(1)优化 IF 外提技术，通过层次遍历的方法，实现了条件为循环不变量的嵌套控制流的 IF 外提，并通过优化计算 IF 外提位置，将 IF 外提到嵌套循环的最外层；

(2)通过 IF 转换和语句匹配以及条件合并的有效结合，实现了条件为循环变量的嵌套控制流的 IF 转换，有效地将控制依赖转换为数据依赖，实现向量化；

(3)通过 IF 重构算法，避免 IF 转换给未向量化代码带来的性能降低的问题。

本文第 2 节介绍了相关研究；第 3 节提出了控制流条件分类方法；第 4 节介绍了控制流向量化的框架及其实现；第 5 节给出了实验结果及分析；最后是结束语。

2 相关研究

SIMD 功能单元在现代微处理器中是普遍存在的。有效地利用这些 SIMD 功能单元对于取得最高性能是必需的。由于手工向量化存在许多限制而难以推广，自动向量化方法成为当前 SIMD 向量化的主要手段。当前主要有两种使用广泛的自动向量化方法：一种是借鉴向量机上的传统向量化方法[1,5-8]；另一种是 SLP 向量化方法[2,9-11]。自动生成含有控制流的 SIMD 指令是非常具有挑战性的。此前，研究人员针对控制流向量化已经进行了大量的研究。

Cocke 和 Allen 提出了 IF 外提技术，通过将条件为循环不变量的控制流外提到循环外来减少或消除循环内的控制流[3]。该技术的优势在于，一方面能够实现更高效的调度、更方便的寄存器分配和更快的执行速度；另一方面能够减少执行分支数，增加发掘循环向量化的机会[12]。但 IF 外提后代码大幅膨胀，不利于编译器做其他优化。Lokuciejewski 等人利用不变路径信息，提出一种基于 WCET(Worst-case Execution Time)优化驱动的 IF 外提方法[13]，能有效改善代码膨胀问题；但是由于 WCEP(Worst-case Execution Path)信息不能得到及时有效的更新，该方法很难实现应用。Barton 等人利用索引集分裂技术将含有条件为循环变量控制流的循环分成几个等价的不含控制流的循环[14]，但能实施 ISS(Index-Set Splitting)的控制流条件必须与索引变量同名。

Towle 最早提出通过维护一个位向量的守恒集合来模拟简单的控制流，从而在向量机上实现了对控制流的处理[15]。Allen 在文献[4]中提出了 IF 转换技术，其主要思想是将所有的控制依赖转换为数据依赖，从而把所有依赖统一成一种形式。Park 和 Schlansker 改进了 IF 转换技术，使生成的谓词的数目和谓词定义指令的数目达到最佳，但生成的代码依然需要在所有的控制通道都执行，以得到最终结果[16]。

Bik 等人利用位掩码技术来合并从不同控制流分支产生的数值，但这个方法仅限于单层的控制流语句[17]。Shin 和 Hall 等人提出了一种基于 SLP(Superword-Level Parallelism)实现单层 IF 向量化的方法[18]；随后又通过生成嵌套的 BOSCCs 指令和 vec_any_* 指令，将嵌套控制流成功引进向量化代码中，但该技术依赖于预测执行[19,20]。Zhu Jia-feng 等人利用程序结构等价变换实现了含有出口分支控制流循环的向量化[21]。Tanaka 等人提出了一种以维持基本块间数据依赖关系为基础的 SIMD 代码生成技术，在不对控制流结构作改变的情况下实现了基本块的向量化，但并没有消除条件分支[22]。Karrenberg 等人在 SSA(Single Static Assignment)形式下利用 mask 和 blend 操作来处理任意的控制流图，实现了含有控制流函数的向量化，但该方法只适用于以 SSA 作为中间表示的编译器[23]。多面体模型是一个灵活的富有表现力的非完美嵌套循环的表示方法，前提条件是循环中含有的控制流是静态可预测的[24]。Sujon 等人基于路径优化策略对含有控制流分支的语句生成 SIMD 指令，但该方法只支持给定循环的单一路径的向量化[25]。

3 控制流条件分类

高级编程语言提供一个用于引导应用程序检查现有条件并决定相应操作的判断语句，称为控制流，主要包括条件选择控制语句(如 if-then-else 语句)、多重条件选择控制语句(如 switch-case 语句)和循环控制语句(如 while 循环和 for 循环)等。除了 for 循环外，其它的控制流语句一直是向量化研究的难点，原因在于向量化的基础是发掘并利用数据流中的并行性，而控制流为数据流分析带来了复杂性和不确定性。本文用到的向量化方法在预分析阶段将多重条件选择控制语句转换为条件选择控制语句。因此下文提到的控制流指的是条件选择控制语句，即 if-then-else 语句。

在实际程序中，控制流的形式多种多样。本文主要研究不含跳转、过程调用和过程返回语句的控制流，这种情况在高性能计算程序中大量存在。为了方便讨论，下面给出一般控制流结构(Common Control Flow，CCF)的定义。

定义 1 如果一个 IF 结构满足：(1)条件是比较操作符或由逻辑操作符连接的比较操作符，(2)无跳转、过程调用和过程返回语句，那么该 IF 结构就是一般控制流结构。

控制流的向量化一直是现代编译领域的研究热点，不仅因为其难度大，也因为其收益高[19]。处理控制流主要有两种基本的策略：第一种是将控制依赖转换成数据依赖来消除控制流，主要用于自动向量化；另一种是将控制依赖作为数据依赖的拓展来处理，并在依赖图中加入控制依赖边，适用于自动并行化。本文主要采用第一种策略来消除控制流。

程序中的循环占据了程序运行的大部分时间，是向量化的主要对象。如果循环内存在控制流，循环优化和分析变得举步维艰。以往的向量化发掘方法对控制流的处理比较单一，本文根据控制流条件与循环迭代之间的关系，将控制条件变量分成循环不变量和循环变量两类。循环不变量不会随着循环迭代的改变而改变，而循环变量会随着循环迭代的改变而发生改变。基于此分类，对于控制条件变量是循环不变量的控制流，采用优化的 IF 外提消除循环内的控制流。如下所示的循环内的控制流，其控制条件为循环不变量，可以通过

IF 外提消除循环内控制流。

```
do i=1,100
  if(am. lt. 10. AND. an. gt. 10) then
    x(i)=y(i)
  endif
  m(i)=n(i)
enddo
```

对于控制条件变量是循环变量的控制流，采用 IF 转换将控制流转换为 SELECT 指令。这么做是因为大多数现代编译器中都支持 SELECT 指令[26]。如下所示的代码段来源于 SPEC2006 测试集中的 435. gromacs 程序，其中的控制流条件为循环变量，可以通过 IF 转换实现向量化。

```
for(i=0;(i<F_EPOT);i++)
    if (i !=F_DISRES)
        epot[F_EPOT]+=epot[i];
```

4 控制流向量化的实现

4.1 控制流向量化框架

本研究旨在对含有嵌套 IF 结构的循环进行向量化。图 1 示出了基于条件分类的控制流向量化方法的实现框架。在进入该框架前，首先需要对循环进行冗余代码删除、常量传播等预优化。对于给定的一段串行代码，顺序遍历每个含控制流的循环，判断 IF 结构的条件是否是循环不变量条件，若是，则对该循环以层次遍历的顺序执行 IF 外提，直到循环内没有可以外提的控制流结构。若循环内还含有控制流结构，则对该控制流结构按照反序遍历的方式将最内层控制流结构转换为 SELECT 语句，并通过条件合并处理嵌套控制流的条件，直到循环内没有可以做 IF 转换的控制流结构。经过上述两步处理，大部分的控制流结构都能通过向量化预分析，从而实现向量化，这部分代码最终生成 SIMD 指令；但仍有一部分循环在将控制流消除后，因依赖关系限制而无法向量化。为了减少 SELECT 指令引入的额外计算，需要对转换为 SELECT 的控制流结构进行 IF 重构。

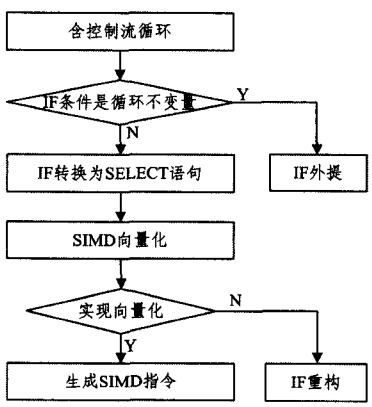

图 1 控制流向量化框架

4.2 IF 外提

IF 外提可以将条件为循环不变量的控制流外提到循环外并分别在 then 分支和 else 分支中生成新的不含控制流的循环。该技术的优势在于能够减少执行分支数，增加发掘循环向量化的机会；另一方面，消除循环内的控制流更便于编译器做其他编译优化，提高代码的性能。

当前主流的编译器 GCC 和 open64 等都尝试进行 IF 外提，能够实现单层控制流外提到最内层循环外。但实际程序中控制流的形式比较复杂，因此当前编译器对实际程序中的控制流很难实现外提。

本节对 IF 外提算法进行了改进，设计了一种以层次遍历顺序实施 IF 外提的算法，能对包含嵌套控制流的循环进行处理，并且能将嵌套控制流的条件外提到嵌套循环的最外层。针对实际应用程序，遍历最内层循环，在最内层循环中以层次遍历的顺序查找 IF 结构，对查找到的 IF 语句做如下处理：判断 IF 语句是否不含 else 分支且循环块中只有 IF 一条语句，若是，则置 else_part 为 true，否则置为 false；在控制条件对循环是可观察的条件下，计算外提位置，若不能外提则返回，否则记下外提到的新位置 new_pos；若 else_part 为真，则复制一份 new_pos 作为 copy，将 copy 中 else 块中的语句放到循环体内，并删掉 copy 中的 IF 结构；然后删掉 new_pos 中的 else 块，将 new_pos 中 then 块中的语句抽离出来放到循环体中；而后将 IF 条件外提到合适的位置，并插入 copy 作为 else 块，new_pos 作为 then 块；更新定义使用(DU)关系，并对分裂出来的新循环依次递归调用该算法。对于嵌套的控制流，先处理外层的 IF 结构，处理完后，原先内层的 IF 结构就以外层控制流的形式暴露在各个分支中的新循环中，然后递归外提新循环中的控制流。算法的具体实现过程如图 2 所示。计算外提位置时，在满足循环内无跳转、过程调用和过程返回语句的前提下，控制流可以外提到最外层循环外，使程序的加速性能达到最大。

```
1. //功能:将最内层循环中条件为循环不变量的 IF 结构外提到循环外
2. //输入:最内层循环 wn
3. //输出:若一般嵌套循环中有可以提到循环外的 IF 结构,则外提到循环外,消除循环内控制流
4. procedure Loop_Unswitching_ForSIMD(wn)
5.     获取当前 wn 的操作符 operator;
6.     if operator==OPR_IF then
7.         else_part=TRUE;
8.         if 当前 IF 结构不含 else 分支 && 循环中只有 IF 一条语句 then
9.             else_part=FALSE;
10.        end
11.        n=Non_Const_Loops(WN_if_test(wn)); //计算控制流条件对哪层循环来说是循环不变量
12.            //当对包围它的所有循环都是循环不变量时返回 0,否则返回 1,若无法观察,则返回-1
13.            if n !=-1 then
14.                new_pos=Highest_Condition_Point_ForSIMD(wn,n); //计算外提的位置
15.                if new_pos !=wn then //可以将 IF 结构提到循环外
16.                    if else_part != NULL then
17.                        克隆 new_pos 作为 else_block,在保留 else 分支的条件下,删除 else_block 中的 IF 结构;
18.                        更新 DU 链以及数据依赖图;
19.                    end
20.                    在保留 then 分支的条件下,删除 new_pos 中的 IF 结构;
21.                    将 wn 中的 IF 结构外提到 new_pos 外;
22.                    将 new_pos 和 else_block 分别作为新的 IF 结构的 then 分支和 else 分支;
```

```
23.                    更新 DU 链以及数据依赖图;
24.                    Update IF_INFO;
25.                end
26.            end
27.            对当前的 wn 遍历最内层循环,并递归调用该函数
               处理外提后的最内层循环中的 IF 结构;
28.    else if operator==OPR_BLOCK then
29.        for each stmt in this wn begin
30.            Loop_Unswitching_ForSIMD(stmt);
31.        end
32.    else
33.        for each kid of the wn begin
34.            Loop_Unswitching_ForSIMD (kid);
35.        end
36.    end
37. end Loop_Unswitching_ForSIMD
```

图 2　IF 外提算法

```
do i=1100
if(am. lt. 10) then
    if(an. gt. 10) then
        a(i)=c(i)
    else
        a(i)=d(i)
    endif
end if
m(i)=n(i)
end do
```

(a)原程序

```
if(am. lt. 10) then
    if(an. gt. 10) then
        do i=1100
            a(i)=c(i);
            m(i)=n(i);
        enddo
    else
        do i=1100
            a(i)=d(i);
            m(i)=n(i);
        end do
else
    do i=1100
        m(i)=n(i)
    end do
end if
```

(b)IF 外提后代码

```
IF(AM . LT. (10)) THEN
    IF(AN . GT. (10)) THEN
        DO I=11004
          CALL simd_load(SIMD_P1_PREG_0,C(I))
          CALL simd_store(SIMD_P1_PREG_0,A(I))
          CALL simd_load(SIMD_P0_PREG_0,N(I))
          CALL simd_store(SIMD_P0_PREG_0,M(I))
        END DO
    ELSE
        DO I=11004
          CALL simd_load(SIMD_P3_PREG_0,D(I))
          CALL simd_store(SIMD_P3_PREG_0,A(I))
          CALL simd_load(SIMD_P2_PREG_0,N(I))
          CALL simd_store(SIMD_P2_PREG_0,M(I))
        END DO
    ENDIF
ELSE
        DO I=11004
          CALL simd_load(SIMD_P4_PREG_0,N(I))
          CALL simd_store(SIMD_P4_PREG_0,M(I))
    END DO
ENDIF
```

(c)向量化代码

图 3　IF 外提示例

图 3(a)中含嵌套控制流的代码段,实施外提后生成的代码如图 3(b)所示,最后经过向量化生成的 SIMD 指令如图 3(c)所示。IF 外提消除了循环内的 IF 分支,使原先不能向量化的循环实现了向量化,提升了程序的性能。

4.3　IF 转换为 SELECT 语句

IF 外提虽能取得很高的程序加速,但是只对控制条件为循环不变量的控制流有效。对于含有条件是循环变量控制流的循环,由于条件与循环迭代之间存在数据依赖关系,IF 外提前后语义不一致,有可能导致程序运行出错。本文采用 IF 转换的方法来实现此类循环内控制流的消除。IF 转换是由 Allen 等人提出的,通过把 IF 语句转换成带控制条件语句的形式来删除所有分支语句,其中控制条件反映语句被执行时实际的条件集合,可以看成是语句的输入[4]。IF 转换通过将语句转化成有条件控制的形式,将控制依赖转化成数据依赖,从而把所有依赖统一成一种形式。理论上它是一种处理控制依赖的完美方法。

IF 转换已被广泛应用于现代编译器中,诸如 ICC、GCC 等都编写了自己的 IF 转换模块。尽管如此,任意复杂的控制流的 IF 转换仍是现代编译器的一大难题。当前的 IF 转换技术不能很好地向量化 IF 嵌套,因此本文提出了一种新的 IF 转换算法,通过语句匹配和条件合并等方法实现了嵌套控制流的 IF 转换,进而实现了向量化。

与 IF 外提不同,我们先对最内层的 IF 结构进行 IF 转换,而后用后序遍历的方法递归处理外层的循环。顺序遍历到给定程序的最内层循环,查找循环中的最内层 IF 结构,然后用图 4 所示算法依次对最内层 IF 结构进行 IF 转换,将任意复杂嵌套控制流转换为 SELECT 语句。在转换的过程中,对 then 块中的语句和 else 块中的语句分别进行处理,若遇到 SELECT 语句,将 SELECT 的条件输入与控制流条件进行合并,得到新的 SELECT 语句,否则将 then 块和 else 块中的语句按左值相等的原则进行语句匹配,如果匹配成功,则将两条语句合成一条 SELECT 语句;对 then 块和 else 块中匹配不成功的语句分别生成一条 SELECT 语句。为保证 IF 转换前后语义一致,在语句匹配时需要考虑 then 块和 else 块中语句的相对顺序。以图 5 所示控制流程序为例来介绍本节语句匹配算法。由 then-block 开始按照从前向后的顺序遍历,对于每一条语句,在 else-block 中寻找与之匹配的语句:

case1:在当前 else-block 的 stmt 链中,第一条就能与之匹配,生成 b(i)=select(cond,c(i),d(i));

case2:找不到与之匹配的 stmt,生成 h(i)=select(cond,k(i),h(i));

case3：找到与之相匹配的 stmt 是当前 else-block 语句链的第 $N(N!=1)$ 条，先暂时不处理，而先处理 else-block 中的 stmt，若在 then-block 中找不到与之相匹配的语句，则生成 f(i)=select(cond,f(i),g(i))；若能找到，就说明存在语句交叉。需要进行依赖测试，若能将语句 5 放到语句 4 前，则调换语句顺序，生成 u(i)=select(cond,v(i),w(i)) 和 l(i)=select(cond,m(i),n(i))，否则 return。

```
1. //功能：通过语句匹配和条件合并遍历实现嵌套控制流的 IF 转换
2. //输入：最内层循环的最内层 IF 结构 wn
3. //输出：将 IF 结构转换为 SELECT 语句
4. procedure Translate_If_To_Select (wn)
5.   if 当前 wn 的操作符是 OPR_IF && 该 IF 结构是当前循环中最内层 IF then
6.     if 当前的 IF 结构不是一般控制流结构 then
7.       return;
8.     end
9.     新建一个 block 块 sel_wn;//sel_wn 用来存放新生成的 SELECT 指令，以替换当前的 wn
10.    for each 当前 IF 结构的 then-block 或 else-block 中的 stmt begin
11.        couple=FALSE; //couple 为 TRUE 时表示 if 中的 else-block 中含有与 then-block 中相匹配的语句
12.        if stmt 是 SELECT 指令 then
13.          合并当前 IF 结构条件与 SELECT 指令的条件操作数，生成新的 SELECT 指令; //嵌套控制流
14.        end
15.        if then_stmt != NULL then
16.          for each else_stmt begin
17.            if then_smt 和 else_stmt 相匹配 then
18.              couple=TRUE;
19.              记下相匹配的语句为 else_iter;
20.            end
21.          end
22.          if couple==TRUE then
23.          if else_iter==else_stmt then //then_stmt 和 else_stmt在 block 中的位置对等
24.            生成 dst=select(cond,src1,src2)语句并插入到 sel_wn;
25.          else
26.            couple=FALSE;
27.            for each then_stmt begin
28.              if then_stmt 和 else_stmt 相匹配 then
29.                couple=TRUE;
30.                记下相匹配的语句为 then_iter;
31.              end
32.                if couple==TRUE then
33.                    if Forward_Motion(then_stmt,WN_next(then_iter) then
34.                      //位置不对等时，考察改变语句顺序是否会引起依赖关系或语义的改变
35.                      生成 dst=select(cond,src1,src2)语句并插入到 sel_wn;
36.                  else
37.                    return;
38.                  end
39.                  else
40.                    生成 dst=select(cond,dst,src2)语句并插入到 sel_wn;
41.                  end
42.              end
43.            end
44.        else
45.          生成 dst=select(cond,src1,dst)语句并插入到 sel_wn;
46.        end
47.      else if else_stmt != NULL then
48.        生成 dst=select(cond,dst,src2)语句并插入到 sel_wn;
49.      end
50.    end
51.    用 sel_wn 替换原来的 IF 结构，更新当前 wn;
52.    更新 DU 链;
53. end Translate_If_To_Select
```

图 4　IF 转换算法

循环进行上述过程，直到 then-block 中的所有语句都处理完。对 else-block 中剩余语句 6，生成 x(i)=select(cond,x(i),y(i))。

```
if(cond)then
    b(i)=c(i); ……1
    h(i)=k(i); ……2
    l(i)=m(i); ……4
    u(i)=v(i); ……5
else
    b(i)=d(i); ……1
    f(i)=g(i); ……3
    u(i)=w(i); ……4
    l(i)=n(i); ……5
    x(i)=y(i); ……6
```

图 5　语句匹配控制流代码示例

对于图 6(a)中的程序，其经过 IF 转换后，得到图 6(b)所示的代码段，最后生成的 SIMD 指令如图 6(c)所示。由于篇幅限制，这里只给出 S1 向量化后生成的 SIMD 指令。条件合并的原则是对 then 分支中的语句直接将条件与 SELECT 语句的条件操作数相与，对 else 分支的语句，将条件取反后与 SELECT 语句的条件操作数相与。如果程序中有多层条件嵌套，则按此规则逐级向外层规约。

```
for(i=0;i<1024;i++)
  {
    if(a[i]>0&&a[i]>b[i])
    {
S1    v[i]=w[i];
        if(b[i]>10)
S2      p[i]=m[i];
    else
    {
S3      p[i]=n[i];
        if(c[i]<100)
S4        x[i]=y[i];
      }
    }
}
```

(a)原程序

```
for(i=0;i<1024;i++)
{
S1  v[i]=select(a[i]>0&&a[i]>b[i],w[i],v[i]);
S2  p[i]=select(a[i]>0&&a[i]>b[i]&&b[i]>10,m[i],p[i]);
S3  p[i]=select(a[i]>0&&a[i]>b[i]&&b[i]<=10,n[i],p[i]);
S4  x[i]=select(a[i]>0&&a[i]>b[i]&&b[i]<=10&&c[i]<
100,y[i],x[i]);
}
```

(b)IF 转换后代码

```
for(i=0; (i<1024); i+=4)
{
   simd_load(V_a,a[i,i+3]);
   simd_load(V_b,b[i,i+3]);
   simd_load(V_v,v[i,i+3]);
   simd_load(V_w,w[i,i+3]);
   V_0= (floatv4)(0);
   V_cmp1=simd_vcmple(V_a,V_0);
   V_cmp2=simd_vcmple(V_a,V_b);
   V_and1 =simd_vand(V_cmp1,V_cmp2);
   V_v=simd_vseleq(V_and1,V_v,V_w);
   simd_store(V_v,v[i,i+3]);
}
```

(c)向量化代码

图 6　IF 转换示例

4.4　IF 重构

虽然 IF 转换是非常有用的转换，但由于转换前无法识别出语句之间是否存在阻碍向量化的依赖环，造成部分不能向量化的循环实现了 IF 转换。为了避免 IF 转换使这些没有向量化的循环性能变差，采用条件重构的逆转换过程。顺序遍历最内层循环，查找未能向量化的 SELECT 语句，对条件相同的 SELECT 语句生成一个 IF 结构；对条件不同的 SELECT 生成不同的 IF 结构，具体算法如图 7 所示。

```
1. //功能:将最内层循环中未向量化的 SELECT 重构成 IF 结构
2. //输入:最内层循环 wn
3. //输出:若 wn 中有未向量化的 SELECT,则重构成 IF 结构
4. procedure Walk_Unvectorized_Select2If(wn)
5.     获取当前循环的循环体 body;
6.     for each stmt in the body begin
7.         if stmt 是 SELECT 语句 then
8.             获取当前 SELECT 语句的条件作为控制流的条件
               cond;
9.             新建一个块 then_block;
10.            新建一个块 else_block;
11.            for body 中 stmt 及其后继结点 next_stmt begin
12.                if next_stmt 是 SELECT 语句 && 条件与 stmt
                   条件相同 then
13.                    将 SELECT 语句中的 then_stmt 和 else_st-
                       mt 还原放到相应的 then _ block 或 else _
                       block 中;
14.                end
15.            end
16.                利用 cond 以及 then_block,else_block 重构 IF
                   结构;
17.                更新循环及控制流信息;
18.        end
19.    end
20. end Walk_Unvectorized_Select2If
```

图 7　IF 重构算法

5　实验与分析

5.1　实验环境

本文的控制流向量化方法基于开源编译器 Open64 实现，在其 LNO 模块中实现 SIMD 自动向量化功能。

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。测试平台为国产高性能处理器 SW1600，其中包含了向量长度为 256 位的 SIMD 部件，并安装有支持向量化扩展接口的基础编译器。操作系统内核为 Linux 2.6.18，版本为 Redhat Enterprise AS 5.0。

5.2　测试集

为验证本文提出的基于条件分类的控制流向量化方法，对 SPEC2006 标准测试集进行测试。SPEC2006 标准测试集是 SPEC 组织推出的最新版的 CPU 子系统评估软件，共有 29 个程序，其中 8 个程序的核心函数中含有控制流，如表 1 所列。表 1 各列分别给出了核心函数的功能描述、所在程序、占程序的执行时间比重、控制流的类型及在后序图中的标记等信息。由表 1 可以看出，这 8 个核心函数中有 7 个控制流的类型是嵌套控制流，向量化嵌套控制流对程序性能的提升有重要作用。

表 1　测试用例

核心函数	描述	程序	核心执行时间/%	语言	控制流类型	图中标记
primal_bea_mpp	组合优化	429.mcf	49.95	C	嵌套	mpp
P7Viterbi	基因序列搜索	456.hmmer	99.53	C	嵌套	P7Viterbi
SetupFastFullPelSearch	视频压缩	464.h264ref	40.93	C	嵌套	Search
e_c3d	结构力学	454.calculix	69.12	Fortran 90 & C	嵌套	e_c3d
do_play_move	围棋	445.gobmk	10.57	C	嵌套	move
vector_gautbl_eval_logs3	语音识别	482.sphinx3	38.67	C	单层	logs3
std_eval	国际象棋	458.sjeng	15.11	C	嵌套	eval
quantum_toffoli	量子计算	462.libquantum	63.41	C	嵌套	quantum

5.3　测试结果与分析

为说明本文方法的向量化性能，首先通过自动向量化生成相应的向量化代码，再用基础编译器编译成可执行的二进制码，最后在 SW1600 上验证算法的正确性。用串行程序的运行时间除以用本文方法向量化后程序的运行时间得到加速比。用 Open64 编译器的控制流向量化方法获得的加速比测试结果标记为 Open64 Original。加入本文控制流向量化方法后，获得的加速比测试结果标记为 Open64＋CF。对每个测试用例，分别用 test 和 reference 两种数据规模测试。测试结果如图 8 和图 9 所示。

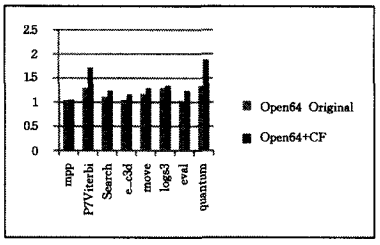

图 8　test 规模加速比测试

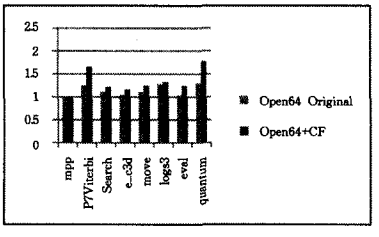

图 9　reference 规模加速比测试

从图 8 和图 9 中可以看出，当测试输入的数据规模变大时，加速比会降低。这是因为向量化主要是减少指令执行时间，对访存操作的时间影响比较小。当数据规模增大时，访存操作所占的时间比例增大，向量化带来的加速比性能就降低了。这与 Shin 在文献[19]中的实验论断是一致的。P7Viterbi 和 quantum 的加速比在应用本文的方法后得到了显著的提升，这是因为嵌套控制流在这两个核心函数中构成了循环体的主要执行部分，利用改进的 IF 外提和 IF 转换方法对嵌套控制流所在的循环进行了向量化，充分发掘了程序的并行性，程序性能得到了有效的提升。mpp 和 logs3 的加速比相比于 Open64 之前的向量化方法几乎没有提升。前者是因为嵌套控制流块中含有复杂的结构体数据，当前编译器无法对其进行向量化，且控制流条件因是循环变量而无法实施外提。后者是因为控制流类型是单层控制流，原先 Open64 的控制流向量化方法就能实现向量化。Search、e_c3d、move 以及 eval 也取得了一定的性能提升，这是因为循环中的控制流得到了向量化，但由于控制流在整个核心代码段中占的比重低于 P7Viterbi 和 quantum，因此性能提升没有 P7Viterbi 和 quantum 明显。

从实验结果可以看出：

(1)本方法对条件为循环不变量的控制流能够实现改进的 IF 外提，消除循环中的控制流，对于向量化和未向量化的代码都能带来很大的性能提升；

(2)本方法对条件为循环变量的控制流能够实现改进的 IF 转换，将控制流转换为数据流，实现循环的向量化，向量化后能给程序带来很大收益；

(3)未实现向量化的 SELECT 语句都能成功进行 IF 重构；

(4)在 test 规模下，应用本文的方法与 Open64 原先的向量化方法相比，加速比平均提高 24.2%，reference 规模性能平均提升 20%。

结束语　控制流的向量化一直是现代编译器面临的一大难题，但因为其可能带来高收益，所以也一直是研究的热点。以往的控制流向量化方法对于单层控制流的向量化很有效，但对嵌套控制流或是条件复杂的控制流的向量化收益甚小。本文提出的基于控制条件分类的向量化方法能针对不同的控制流结构进行不同的变换，将循环内的控制流消除或将控制流转换为数据依赖，从而实现向量化，在性能上有较大的提升。本文所研究的控制流是不包含跳转、过程调用或过程返回语句的 IF 结构，而在高性能计算程序中有大量含有跳转语句的 IF 结构，因此下一步的工作重点是对含有跳转语句的控制流进行向量化。

参 考 文 献

[1] Sreraman N, Govindarajan R. A Vectorizing Compiler for Multimedia Extensions[J]. International Journal of Parallel Programming, 2000, 28(4): 363-400

[2] Larsen S, Amarasinghe S. Exploiting Superword Level Parallelism with Multimedia Instruction Sets[C]// Conference on Programming Language Design and Implementation. 2000: 145-156

[3] 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

[4] 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

[5] Wald I, Leißa R, Hack S. Extending a C-like Language for Portable SIMD Programming[C]// Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP). 2012

[6] Stock K, Poudhet L. Using Machine Learning to Improve Automatic Vectorization[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2012, 8(4)

[7] Vasilache N, Meister B, Baskaran M, et al. Joint Scheduling and Layout Optimization to Enable Multi-level Vectorization[C]// Proceedings of the International Workshop on Polyhedral Compilation Techniques (IMPACT). 2012

[8] Kong M, Pouchet L-N, Sadayappan P. Abstract Vector SIMD Code Generation Using the Polyhedral Model: Technical Report Technical Report 4/13-TR08[R]. Ohio State University, 2013

[9] Barik R, Zhao Ji-sheng, Sarkar V. Efficient Selection of Vector Instructions using Dynamic Programming[C]// Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 2010

[10] Liu Jun, Zhang Yuan-rui, Kandemir M. A Compiler Framework for Extracting Superword Level Parallelism[C]// Proceedings of the 2012 Conference on Programming Language Design and Implementation (PLDI). 2012

[11] Park Yong-jun, Park H, Cho H K, et al. SIMD Defragmenter: Efficient ILP Realization on Data-parallel Architectures[C]// Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2012

[12] Cooper K D, Torczon L. Engineering a Compiler[M]. San Francisco: Morgan Kaufmann, 2004

[13] Lokuciejewski P, Gedikli F, Marwedel P. Accelerating WCET-driven Optimizations by the Invariant Path Paradigm: A Case Study of Loop Unswitching[C]// Proceedings of the 12th International Workshop on Software and Compilers for Embedded Systems. Nice, France, 2009: 23-24

[14] Barton C, Tal A, Blainey B, et al. Generalized Index-set Splitting [C]// Proceedings of the 14th international conference on Compiler Construction (CC'05). Berlin, Heidelberg, 2005: 106-120

[15] Allen R, Kennedy K. Optimizing Compilers for Modern Archi-

tectures: A Dependence-based Approach[M]. Morgan Kaufmann Publishers,2001

[16] Park J, Schlansker M. On Predicated Execution[R/OL]. 1991. http://www.hpl.hp.com/techreports/91/HPL-91-58.pdf

[17] Bik A J C, Girkar M, Grey P M, et al. Automatic Intra-register Vectorization for the Intel Architecture[J]. International Journal of Parallel Programming, 2002, 30(2): 65-98

[18] 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). Washington, DC, USA, 2005: 165-175

[19] Shin J. Introducing Control Flow into Vectorized Code [C]// Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07). Washington, DC, USA, 2007: 280-291

[20] Shin J, Hall M W, Chame J. Evaluating Compiler Technology for Control-flow Optimizations for Multimedia Extension Architectures[C]// In 6th Workshop on Media and Streaming Processors. 2009

[21] Zhu Jia-feng, Zhao Rong-cai. A Vectorization Method of Export Branch for SIMD Extension[C]// Proceedings of the 10th conference IEEE/ACIS International Conference on Computer and Information Science (ICIS). 2011

[22] Tanaka H, Ota Y, Matsumoto N, et al. A New Compilation Technique for SIMD Code Generation across Basic Block Boundaries[C]// Proceedings of the 15th Asia and South Pacific Design Automation Conference (ASP-DAC). 2010: 101-106

[23] Karrenberg R, Hack S. Whole Function Vectorization[C]// 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2011: 141-150

[24] Kong M, Veras R, Stock K. When Polyhedral Transformations Meet SIMD Code Generation[C]// Proceedings of the 2013 Conference on Programming Language Design and Implementation (PLDI). 2013

[25] Sujon M H, Whaley R C, Yi Qing. Vectorization Past Dependent Branches Through Speculation[C]// Parallel Architecture and Compilation Techniques(PACT'13). 2013

[26] Smith J E, Faanes G, Sugumar R. Vector Instruction Set Support for Conditional Operations[C]// International Symposium on Computer Architecture. ACM, 2000

(上接第 216 页)

数据流检测的规则模板，结合 Snort 网络处理机制和异常行为表中有效信息可迅速完成检测规则的编写。在未来的研究工作中，可将 Modbus TCP 的 Snort 异常检测规则的设计方法进一步扩展到 DNP3、EtherNet/IP、Profibus 等其他工控协议中，从而构建一套完整的基于 Snort 的典型工控协议的异常检测系统。

参 考 文 献

[1] 张运凯，王长广，王方伟，等. “震荡波”蠕虫分析与防范[J]. 计算机工程，2005，31(18)：65-67
Zhang Yun-kai, Wang Chang-guang, Wang Fang-wei, et al. Other “Sasser” worm analysis and prevention [J]. Computer Engineering, 2005, 31(18): 65-67

[2] Beaumont P. Stuxnet worm heralds new era of global cyberwar [N]. London: Guardian.co.uk, 2010-9-30(16)

[3] Ardisk K. Stuxnet 病毒引发的嵌入式系统安全性考虑[J]. 电子技术设计，2013(3)：49-50
Ardisk K. Stuxnet virus triggered embedded system security considerations [J]. Electronic Technology Design, 2013(3): 49-50

[4] 高国辉. 西门子被曝工业系统漏洞或影响多数工业化国家[N]. 南方日报，2011-6-8(A18)
Gao Guo-hui. Siemens traced to industrial system vulnerabilities or affected most industrialized countries [N]. Nanfang Daily, 2011-6-8(A18)

[5] Bencsáth B, Pék G, Buttyán L, et al. Duqu: Analysis, detection, and lessons learned[C]// ACM European Workshop on System Security (EuroSec). 2012

[6] 纪芳. Flame 病毒深度分析及防范技术[J]. 信息网络安全，2012(12)：67-69
Ji Fang. Flame virus-depth analysis and prevention techniques [J]. Information Network Security, 2012(12): 67-69

[7] 李鸿培. 工业控制系统及其安全性研究报告[R]. 绿盟科技，2013
Li Hong-pei. Industrial control systems and safety research report [R]. NSFOCUS, 2013

[8] 卢慧康. 工业控制系统脆弱性测试与风险评估研究[D]. 上海：华东理工大学，2014
Lu Hui-kang. Industrial control systems vulnerability testing and risk assessment studies [D]. Shanghai: East China University of Technology, 2014

[9] Morris T H, Jones B A, Vaughn R B, et al. Deterministic intrusion detection rules for MODBUS protocols[C]// 2013 46th Hawaii International Conference on System Sciences (HICSS). IEEE, 2013: 1773-1781

[10] Fovino I N, Carcano A, De Lacheze Murel T, et al. Modbus/DNP3 state-based intrusion detection system[C]// 2010 24th IEEE International Conference on Advanced Information Networking and Applications (AINA). IEEE, 2010: 729-736

[11] Quickdraw scada IDS [EB/OL]. [2014-09-25]. http://www.digitalbond.com/tools/quickdraw/

[12] Modbus Application Protocol Specification V1.1b [DB/OL]. [2014-09-25]. http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf

[13] MODBUS over Serial Line Specification and Implementation Guide V1.02[DB/OL]. http://www.modbus.org/docs/Modbus_over_serial_line_V1_02.pdf

[14] MODBUS Messaging on TCP / IP Implementation Guide V11 [DB/OL]. [2014-09-25]. http://www.electroind.com/pdf/Modbus_messaging_on_TCPIP_implementation_guide_V11.pdf

[15] Roesch Martin, Green Chris. Snort users manual 2.9.6 [EB/OL]. [2014-09-25]. http://manual.snort.org/

[16] 卞峥嵘. Backtracks 从入门到精通[M]. 国防工业出版社，2012
Bian Zheng-rong. Backtracks From Novice to Professional [M]. National Defense Industry Press, 2012

[17] Blanchette J, Summerfield M. C++ GUI programming with Qt 4[M]. Prentice Hall Professional, 2006