基于跨基本块变换和循环分布的SLP优化技术

索维毅[1] 赵荣彩[1] 姚 远[1] 张小妹[2]
(解放军信息工程大学 郑州 450002)[1] (解放军73311部队自动化站 晋江 362200)[2]

摘 要 现有的SLP优化算法无法处理内层循环中存在的依赖环和归约,并且在基本块边界产生大量的冗余拆包和赋值语句,从而导致向量化效率不高。针对该问题,提出了一种基于跨基本块变换和循环分布的SLP优化算法。该算法以控制流图为基础,根据基本块间各数组变量的Define-Use关系以及跨越基本块之间的数据依赖关系进行跨基本块的向量化变换,有序地采用跨基本块变换和循环分布,尽可能发掘最内层循环基本块内语句的并行性,使SLP自动向量化编译器生成具有更多SIMD指令的向量化代码。实验结果表明,该算法能够隐藏更多跨基本块冗余操作的开销,同时利用跨基本块的数据依赖生成更优的SIMD指令,有效地提高了向量化程序的加速比。

关键词 SLP,跨基本块变换,循环分布,数据依赖,控制流图,Define-Use关系
中图法分类号 TP311 **文献标识码** A

SLP Optimization Algorithm Using Across Basic Block Transformation and Loop Distribution

SUO Wei-yi[1] ZHAO Rong-cai[1] YAO Yuan[1] ZHANG Xiao-mei[2]
(PLA Information Engineering University, Zhengzhou 450002, China)[1]
(PLA 73311 Army Automation Station, Jinjiang 362200, China)[2]

Abstract The existing SLP algorithms cannot handle dependent ring and the reduction of the inner loop, and generate a large number of redundant packet disassembly and assignment statements in a basic block boundary, which leads to the lower quantization efficiency. In order to solve the problem, this paper proposed a SLP optimization algorithm using cross basic block transformation and loop distribution. Based on the control flow graph, according to the basic blocks of the array variable between Define-Use and across basic block data relation between across basic block, the algorithm makes the quantized transform, orderly uses across basic block transform and loop distribution, and then expands inner loop within a basic block sentence parallelism as far as possible, making SLP automatic vectorization compiler to genera te the vectorization code which has more SIMD instruction. The experimental results show that the algorithm can hide more across basic block redundancy operation cost, at the same time generate better SIMD instructions across basic block data dependence, effectively improving the vectorization program speedup.

Keywords SLP, Cross basic block, Loop distribution, Data dependence, Control flow graph, Define-Use relationship

1 引言

随着各种数字信号处理在嵌入式系统中的广泛应用,已知的微处理器需要更高的计算能力。最近通用微处理器和数字信号处理已具备SIMD(单指令多数据)指令,可满足更高的需求。如通用处理器中有Intel的SSE、AMD的3DNOW!、PowerPC的AltiVec等扩展指令集[1],以及TI C64x、StarCore、TigerSharc等数字信号处理器DSP所采用的许多指令[2-4]。

虽然SIMD指令在现代微处理器中十分常见,但是,对当前的编译器而言,自动将高级语言编写的源程序变换为高效的SIMD代码仍然是一项艰巨的任务。目前的编译器不能充分利用SIMD指令,主要原因是SIMD代码生成的目标范围目前仅限于由单一基本块组成的单一基本块或循环。SIMD指令通常不能跨越基本块边界被映射,导致了基本块内部控制结构有足够的并行不能被发掘,以致产生出大量的冗余操作,从而严重影响程序的性能。文献[5]提出了跨基本块边界SIMD代码生成的编译技术,其打破了由于控制流因素抑制SIMD代码生成,主要是利用跨越基本块边界的数据依赖来引入高效的向量化打包。但针对内层循环存在依赖环或归约等情况,以及在进行SLP向量化的过程中,基本块边界产生出大量的冗余拆包和赋值语句,不能很好地对其进行优化,导致向量化效率不高。因此本文给出了跨基本块和循环分布的SLP算法,它能很好解决以上两种问题,对一些实例的核心循环取得高达1.92的加速比。

现在主流编译器中的向量化方法是基于传统向量化技

到稿日期:2012-12-24 返修日期:2013-03-03 本文受核高基重大专项(2009ZX01036-001-001-2)资助。
索维毅(1987—),男,硕士生,主要研究方向为先进编译技术,E-mail:weiyisuo@126.com;赵荣彩(1957—),男,教授,博士生导师,主要研究方向为先进编译技术和高性能计算等;姚 远(1972—),男,教授,硕士生导师,主要研究方向为先进编译技术和高性能计算等;张小妹(1981—),女,主要研究方向为先进编译技术和网络安全等。

术，主要根据依赖关系分析，构建依赖关系图来求解强连通分量，如果强连通分量中只有一条语句，则可以对该语句进行向量化。2000 年 Larsen 和 Amarasinghe 提出了超字并行(Superword Level Parallelism，SLP)向量化算法[6]，它可以更有效地对科学计算领域里的程序进行向量化。文献[7]在 SLP 的基础上提出了对控制流语句的向量化，其通过消除控制流扩大了 SLP 算法应用的基本块。它的基本思想是首先将尽可能多的条件语句转换成猜测执行的形式，然后应用 SLP 算法，最后根据向量化的猜测语句生成对应的 SIMD 代码，没有向量化的语句则由猜测执行转换回到条件执行的形式。文献[8]提出了基于短向量的外层循环向量化方法，即主要针对最内层循环出现依赖环和规约等情况，或是外层循环相对于每层循环有更多的迭代次数，选择外层循环进行展开和压紧的向量化方法。这种方法虽然消除了因为内层循环存在依赖环和归约所带来的开销，但不能优化掉跨基本块边界所带来的冗余操作。本文提出的算法可以很好利用跨基本块边界的数据依赖引入高效的向量化打包，能够生成性能更优的 SIMD 指令，大大提升了程序的效率。

2 跨基本块变换在 SLP 中的实现框架

2.1 编译框架

整个 open64 的编译框架如图 1 所示，前端首先采用 gcc 的前端，将源程序转化成相应的中间表示，接着进行过程间的分析和优化、循环级的分析和优化以及全局优化。这些优化不一定严格地按照顺序进行，有时存在一定的交叉，比如在循环级的优化中需要借助全局优化中的一些优化。向量化工作主要在循环嵌套优化(Loop Nest Optimization，LNO)中进行，后端由 whirl2c/whirl2f 将中间表示转换成 C/C++/Fortran 程序，向量化主要通过在源程序中添加 intrisic 向量函数调用的形式实现。

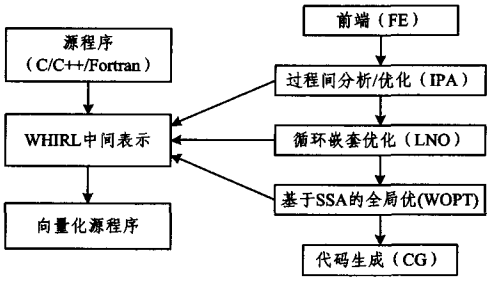

图 1 基于 Open64 的向量化编译框架

2.2 跨基本块变换的实现流程

SLP 自动向量化的部分实现流程如图 2 所示。首先在 IPA 阶段进行数组数据流分析，进入 LNO 阶段后进行预优化，对循环进行筛选，选取那些适于进行向量化识别和变换的循环；之后进行跨基本块的 Define-Use 关系分析，对给定程序的基本块从前至后进行分析，通过中间表示 whirl 树建立跨基本块的定义使用关系图，从而确定程序是否易于进行跨基本块优化；然后进行数据流分析，收集基本块内依赖关系，从而确定可向量化语句和不可向量化语句，并通过本文提出的优化技术基于跨基本块的依赖关系分析实现跨基本块变换和循环分布优化算法；最后按照 SLP 算法进行 pack 生成，通过 whir2c 产生最终的向量化代码。

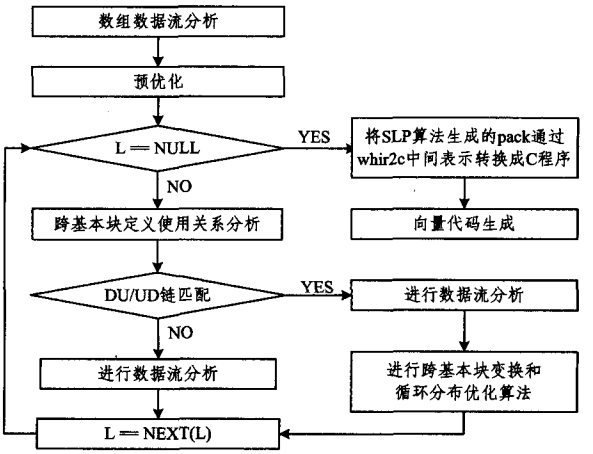

图 2 SLP 自动向量化部分实现流程

2.2.1 SLP 的依赖关系分析和生成控制流图

基本块作为 SLP 向量化的对象[9,10]，控制结构单一并且没有函数调用等复杂情况，因此易于进行数据依赖关系图(Data Dependency Graph，ADG)的构建。依赖关系主要由标量和数组引起，如果两条语句之间存在依赖，就说明它们之间存在定义使用关系，所以一般采用定义使用链来确定标量携带的依赖，如果程序中存在反依赖或输出依赖，则一般通过标量和数组重命名的方法消除依赖。每个 PU 有一个依赖图，两个操作之间存在边当且仅当相同循环内的两个操作引用相同的地址空间，数组 whirl 节点的 load/store 操作对应依赖图中的顶点。当数组的 LDID/STID 操作与 load/store 操作有依赖关系时也会出现在依赖图中。但是两个 LDID/STID 操作之间一定不会有边。

控制流图(Control Flow Graph，CFG)是一种图形表示，用于表示程序运行时可能执行的路径图。在编译器中控制流图是一个由数据结构表示的一个有向图，顶点表示基本块，边表示从一个基本块到另一个基本块的控制流的可能转换[11]。当前我们的 SLP 向量化主要针对的是 for 循环结构，其中不包含函数调用和 goto、if-else 等控制流语句，所以循环只有一个入口。为了使控制流图能够对程序有一个形象的表示，最重要的就是标示基本块(basic block)，基本块是一段只能从它的开始处进入并从它的结束处离开的线性代码序列。whirl 树被用于表示控制流，根据中间代码将源程序划分为多个基本块，控制流只能从基本块的第一条指令进入该块，从基本块的最后一块指令离开，控制流在离开基本块之前不会停机或者跳转。基本块形成了控制流图的节点，而控制流图的边指明了基本块之间的排列顺序，规定了哪个基本块跟在一个基本块之后执行[9]。

2.2.2 基于 CFG 和 ADG 的跨基本块变换规则

本文的算法首先按照代码生成的顺序依次对各个循环层次进行分析，主要收集候选循环的相关信息，包括全局的标量信息、数组私有化信息和某些数组引用信息，这些信息是经过全局分析和过程间分析得到的。其中全局标量信息经过过程间分析所得到的循环边界标量之间的关系进行收集；数组私有化信息根据 DU 链进行收集；数组引用仅仅出现在一个 PU 内——过程间分析；这些信息都为后续的循环优化和其他优化作准备。根据获取的信息生成程序的控制流图(CFG)和数据依赖关系(ADG)，根据 ADG 的信息对 CFG 进行调整，通

过对相邻基本块 WN 树，消除跨基本块重复运算，采用基于 CFG 和 ADG 的跨基本块变换可有效提升程序的性能。

3 基于跨基本块变换和循环分布的 SLP 优化算法

本节首先用形式化方式来定义算法中的一些概念，然后详细地描述基于跨基本块变换和循环分布的 SLP 优化算法。

3.1 形式化描述

定义 1(控制流图，Control Flow Graph，CFG) 是一个二元组，即 CFG=(N,E)，N 是基本块集，E 是一控制流边的集合。根据数据依赖关系对控制流图进行变换，主要通过把原有的中间表示转化为优化后的中间表示，之后根据优化后的中间表示生成性能更优的向量化代码。

定义 2(定义-使用图，Define-Use Graph，DUG) 是一个五元组，即 DUG=($use(a)$,$def(a)$,$defout(a)$,$kill(a)$,$reach(a)$)，其中，$use(a)$表示在基本块 a 中使用的变量集合；$def(a)$表示到达基本块 a 的定义的集合；$defout(a)$表示在基本块 a 中能够到达基本块 a 出口的定义变量集合；$kill(a)$表示在基本块 a 中被新定义消除的原定义集合，其中的变量无法到达基本块 a。

定义 3(IPA 调用图 IPA_CALL_GRAPH * IPA_Call_Graph) 主要通过执行 Perform_Interprocedural_Analysis() 建立 IPA 调用图；DFS_change(df)深度优先搜索的主要方法是优先遍历 IPA_CALL_GRAPH 的每一个节点。

定义 4(变量 ARA_LOOP_INFO) 存放数据节点的定义和使用的信息。Collect_Loop_St_Info(wn,$skip$,ARA_LOOP_INFO)，$skip$ 为语句对应的表达式。根据 wn 节点与 $skip$ 进行匹配，遍历所有 ISTORE 和 STID 节点，把它们加入 ARA_LOOP_INFO 的 may_def 部分；遍历所有 ILOAD 和 LDID 节点，把它们加入 ARA_LOOP_INFO 的 may_use 部分。

定义 5(变量 ALF_INTERCHANGE) 存放循环对应 LOOP_DATA 的循环优化策略。其根据函数 ALF_Interchange_Inter_Loop($scalar\_map$,$loop\_data\_objs$)得到，其中 $scalar\_map$ 为循环上下界的信息，$loop\_data\_objs$ 为循环数据信息。对区域内的所有循环两两比较分析，判断是否需要循环交换，若需要则写入循环对应的 LOOP_DATA 的循环优化策略 ALF_INTERCHANGE 中，具体的交换策略主要是根据对应层次的上下界是否相等来判断。

3.2 算法描述

3.2.1 CommOpt

CommOpt 是本文算法的入口，形参 L 是一个程序的核心循环，N 是过程间分析得到基本块的个数，IPA_info 是过程间数组数据流分析信息。该算法从程序的全局出发，通过数组数据流分析得到各个基本块之间的控制流图和定义使用关系，扩展了相邻基本块之间的映射关系，尽可能地发掘跨越基本块之间语句的并行性并减少不必要的冗余操作。如图 3 所示，CommOpt()的基本思想是：第一步，调用 Across_BB_Bound_UsingProTrans()，采用基于 CFG 和 DUG 的跨基本块变换，尽可能地发掘相邻基本块之间映射。第二步，调用 Across_BB_Bound_Using_Loop-Distribution()，在第一步基础上结合过程间分析的信息，将跨基本块变换和循环分布配合使用，对 L 中的相邻基本块进行合并和拆分，在尽量减少不必要的冗余操作前提下，扩大基本块内语句的并行性，使向量化后得到最优的加速比。第三步，调用 Overall_Situation_Data_Transfers()，生成跨基本块的 CFG。

```
1. algorithm CommOpt(L,N,IPA_info)
2. Across_BB_Bound_UsingProTrans(CFG,DUG)
3. Across_BB_Bound_Using_LoopDistribution(CFG,DUG)
4. Overall_Situation_Data_Tranfers()
5. end CommOpt
```

图 3 CommOpt()的算法实现

3.2.2 Across_BB_Bound_UsingProTrans

作为算法的第一步，Across_BB_Bound_UsingProTrans()采用了基于 ADG 和 CFG 进行跨越基本块变换以消除跨基本块的读写操作，从而避免不同基本块生成冗余的 simd_unpack 和 simd_set 语句。如图 4 所示，Across_BB_Bound_UsingProTrans()的基本思想是：首先根据过程间分析的结果得到程序 L 的基本块个数 N，令 m 为 N 的总个数，如果 m 小于等于 1，则结束；反之则通过一个循环索引 i 为 1 到 m 的 for 循环，每次处理 N 和 $N->$ nest，遍历各个基本块。之后根据 ADG 判断在 N 和 $N->$ nest 是否存在依赖关系，如果不存在 $N->$ nest 到 N 的使用或是 N 到 $N->$ nest 的定义，则结束；反之则调用 IPA_CALL_GRAPH，对其进行深度优先搜索，遍历 IPA_CALL_GRAPH 中的每一个节点。之后根据 ARA_LOOP_INFO(may_def,may_use)获取 N 和 $N->$ nest 中的数据读取操作，根据数组和标量重命名方法消除。再向下检查，重复以上操作，直到 N. nest=m 时，算法第一步结束。

```
1. //功能：基于 CFG 和 ADG 进行跨越基本块代码变换
2. //输入：需要进行基本块并行性挖掘的源程序 L
3. //输出：进行跨基本块变换后的向量化代码
4. procedureAcross_BB_Bound_UsingProTrans(L,N,IPA_info)
5. m=N 的基本块的总数；
6. for each N{1,2,…,m} do begin
7.   N=N-> nest;
8.   N. nest=下一个基本块；
9.   if N. nestNULL&& N 与 N. nest 之间没有依赖关系 &&
10.    没有到达 N. nest 内数据的定义 && 在基本块 N. nest 内数
       据的使用 then
11.    N. nest= m;
12.  end foreach
13.  else begin
14.    if N 与 N. nest 之间有依赖关系
15.      IPA_CALL_GRAPH * IPA_Call_Graph; // "The" call
           graph of IPA
16.      DFS_change(df)深度优先搜索深度优先遍历
17.            IPA_CALL_GRAPH 的每一个节点；
18.      ARA_LOOP_INFO(may_def,may_use);
19.      两个相邻基本块中的不同数据引用了相同的地址空间
20.    else begin
21.      if 有到达 N. nest 内数据的定义 && 在基本块 N. nest 内有
           数据的使用
22.        在 def(N. nest)中，对跨基本块数据的读写删除，
23.                通过标量或是数组重命名；
24.  end foreach
25. end Across_BB_Bound_UsingProTrans
```

图 4 Across_BB_Bound_UsingProTrans()的实现

现在举例说明 Across_BB_Bound_UsingProTrans()的实现过程，以图 5 为例，解释基于 CFG 和 ADG 的跨基本块变换

规则。图 5 中(a)是源程序核心循环段，图 5(b)是消除 ADG 跨基本块依赖关系前的向量化代码；通过第一步算法找出相邻基本块边界产生的冗余操作，显然，其中的 simd_unpack 和 simd_set 都是冗余操作，并将其消除。如果遵循跨基本块变换规则，则可得到如图 5(c)所示的优化后的代码。

Code segment 5-1: 测试用例核心循环段

```
for (i = 0 ; i < n ; i++){
    sum = 0;
    for (j = 0 ; j < n; j++){
        sum += a[i+j] * f[j];}
    c[i] = sum;}
```

(a)

Code segment 5-2: 对 (a) 进行SLP向量化

```
for (vi = 0 ; vi < n ; vi++){
    sum = 0;
    sum_simd_unpack1 = sum[0];
    sum_simd_unpack2 = sum[1];
    sum_simd_unpack3 = sum[2];
    sum_simd_unpack4 = sum[3];
    for (j = 0 ; j < n; j++){
        v1=simd_load(&f[j]);
        v2=simd_set(a[i+j],a[i+j]);
        vsum = simd_set(sum_unpack1,sum_pack2,
                        sum_unpack3,sum_unpack4);
        vsum += v1 * v2;
        simd_store(vsum,sum[0:3])}
    c[i] = sum[0] + sum[1] + sum[2] + sum[3];}
```

(b)

Code segment 5-3: 对 (b) 进行跨基本块变换

```
for (vi = 0 ; vi < n ; vi++){
    sum = 0;
    for (j = 0 ; j < n, j++){
        v1=simd_load(&f[j]);
        v2=simd_set(a[i+j],a[i+j]);
        vsum += v1 * v2;
        simd_store(vsum,sum[0:3])}
    c[i] = sum[0] + sum[1] + sum[2] + sum[3];}
```

(c)

图 5　一个基于 CFG 跨基本块变换实例

3.2.3　Across_BB_Bound_Using_LoopDistribution

作为算法的第二步，Across_BB_Bound_Using_Loop-Distribution()在采用循环分布的情况下，对所有循环两两比较分析，将收集到的信息写入循环对应 LOOP_DATA 的循环优化策略 ALF_INTERCHANGE 中，具体的交换策略主要是根据对应层次的上下界是否相等来判断。目的是消除内层循环的依赖环，同时进行跨基本块变换消除由于循环分布在不同的循环而引起的多次读取操作。如图 6 所示，Across_BB_Bound_Using_Loop-Distribution()的基本思想是：首先根据过程间分析的结果得到程序 L 的基本块个数 N，令 m 为 N 的总个数，如果 m 小于等于 1，则结束；反之则通过一个循环索引 i 为 1 到 m 的 for 循环，每次处理 N 和 N—>nest，遍历各个基本块。若 N 中包含两条以上的语句，则根据 ADG 判断 $loop_t$ 与 $loop_{t+1}$是否存在两个反依赖，构成了一个依赖环，如果不存在，则结束；反之则调用函数 ALF_Interchange_Inter_Loop 对循环进行判断，从而选取最优的交换策略写入到 ALF_INTERCHANGE 中；调用 IPA_CALL_GRAPH，对其进行深度优先搜索，遍历 IPA_CALL_GRAPH 中的每一个节点。之后根据 ARA_LOOP_INFO(*may_def*，*may_use*)获取 *loop_N* 和 *loop_N. nest* 中的数据读取操作；根据 DUG=(*use*(*a*)，*def*(*a*)，*defout*(*a*)，*kill*(*a*)，*reach*(*a*))图，找出 *use*($loop_t$)和 *def*($loop_{t+1}$)之间数据的读取操作，通过标量和数组重命名将其删除；最后根据 Overall_Situation_Data_Transfers()算法，实现跨基本块间的数据传输。再向下检查，重复以上操作，直到 N. nest=m 时，算法第二步结束。

```
1. //功能：由于依赖关系而阻碍向量化的代码生成，
2. 采用基于循环分布和跨基本块变换的 SLP 算法
3. //输入：需要进行基本块并行性挖掘的源程序 L
4. //输出：循环分布和跨基本块变换后高效的向量化代码
5. procedureAcross_BB_Bound_Using_LoopDistribution(L, N, IPA_
   info)
6. m=N 的基本块的总数；
7. for each N{1,2,…,m} do begin
8.    N=N—> nest;
9.    N. nest=下一个基本块；
10.      if loop_N =N. nest 所在循环；
11.        if loop_N 内有两条以上的语句 &&loop_N 有相邻两条语
             句 loop_t 与
12.              loop_{t+1} &&loop_t 与 loop_{t+1}无数据依赖关系
13.          在loop_N 中，根据 IPA_info 的信息进行循环分布，将
             loop_{t+1}从 N. nest
14.            中分布出去，在 loop_N 之后生成含有 loop_{t+1}的循环；
15.        loop_N. nest=包含 loop_{t+1}的循环；
16.      else begin
17.        if loop_N 内两条相邻语句 loop_t 与 loop_{t+1}有数据依赖关
           系
18.          依据 ALF_INTERCHANGE 中的循环优化策略进行
             循环变换
19.          IPA_CALL_GRAPH * IPA_Call_Graph; // "The"
               call graph of IPA
20.          DFS_change(df)深度优先搜索深度优先遍历
21.              IPA_CALL_GRAPH 的每一个节点；
22.          ARA_LOOP_INFO(may_def,may_use);
23.          根据DUG =(use(a),Def(a),defout(a),Kill(a),
               reach(a)),
24.              找出 use(loop_t)和 def(loop_{t+1})之间数据的读取
                 操作，
25.              通过标量和数组重命名将其删除；
26.          Overall_Situation_Data_Transfers;//全局数据传递
27.   end foreach
28. end Across_BB_Bound_Using_LoopDistribution
```

图 6　Across_BB_Bound_Using_LoopDistribution()的实现

举例说明 Across_BB_Bound_Using_LoopDistribution 的实现过程，图 7(a)是源程序核心循环段，由于最内层循环依赖关系的原因，得到的代码没有进行向量化(见图 7(c))。采用第二步算法，根据循环变换策略对内层循环进行循环分布，从而打破内层循环的依赖环(见图 7(b))。但是，循环分布可能破坏数据的局部性，如在一个循环中可一次读取的数据由于分布在了不同的循环，需要多次读取；由于循环分布可将原循环分解为较小循环，因此改善了指令 cache 和存储器的局部性，减少了变量引用，增加了寄存器的重复使用。循环分布之后，分布在不同循环的数据需要多次读取，造成额外的开销(见图 7(d))。需要使用跨基本块变换，消除冗余的操作(见图 7(e))。

Code segment 7-1: 测试用例核心循环段

```
for (i = 0 ; i < n ; i++){
    sum = 0;
    for (j = 0 ; j < 5; j++){
        b[j] = f[j] + d[j];
        sum += a[i+j] * b[j];}
    c[i] = sum;}
```

(a)

Code segment 7-2:对 (a) 采用循环分布

```
for (i = 0 ; i < n ; i++){
    sum = 0;
    for (j = 0 ; j < 5; j++){
        b[j] = f[j] + d[j];}
    for (j = 0 ; j < 5; j++){
        sum += a[i+j] * b[j];}
    c[i] = sum;}
```

(b)

Code segment 7-3: (a)由于依赖关系无法向量化

```
for (i = 0 ; i < n ; i++){
    sum = 0;
    vb[0] = f[0] + d[0];
    sum = sum + a[i] * vb[0];
    vb[1] = f[1] + d[1];
    sum = sum + a[i+1] * vb[1];
    vb[2] = f[2] + d[2];
    sum = sum + a[i+2] * vb[2];
    vb[3] = f[3] + d[3];
    sum = sum + a[i+3] * vb[3];
    vb[4] = f[4] + d[4];
    sum = sum + a[i+4] * vb[4];
    c[i] = sum; }
```

(c)

Code segment 7-4: 对 (b) 进行SLP向量化

```
for (vi = 0 ; vi < n ; vi++){
    sum = 0;
    for (vj = 0 ; vj < 5; vj++){
        v1 = simd_load(&f[vj]);
        v2 = simd_load(&d[vj]);
        v3 = v1 + v2;
        simd_store(v3,b[vj]);
        simd_unpack1 = b[0];
        simd_unpack2 = b[1];
        simd_unpack3 = b[2];
        simd_unpack4 = b[3];}
    for (j = 0 ; j < 5; j++){
        v4 = simd_set(simd_unpack1,simd_unpack2,
                      simd_unpack3,simd_unpack4)
        v5 = simd_set(a[i][j],a[i][j]);
        vsum += v4 * v5;
        simd_store(vsum,sum[0])}
    c[i] = sum[0] + sum[1] + sum[2] + sum[3];}
```

(d)

Code segment 7-5: 对 (d) 采用跨基本块变换

```
for (vi = 0 ; vi < n ; vi++){
    sum = 0;
    for (vj = 0 ; vj < 5; vj++){
        v1 = simd_load(&f[vj]);
        v2 = simd_load(&d[vj]);
        v3 = v1 + v2;
        simd_store(v3,b[vj]);}
    v4 = v3;
    for (j = 0 ; j < 5; j++){
        v5 = v4;
        v6 = simd_set(a[i][j],a[i][j]);
        vsum += v5 * v6;
        simd_store(vsum,sum[0])}
    c[i] = sum[0] + sum[1] + sum[2] + sum[3];}
```

(e)

图 7　使用循环分布

3.2.4　Overall_Situation_Data_Transfers

作为算法的第三步，Overall_Situation_Data_Transfers()

为了减少相邻基本块之间不必要的冗余拆包和赋值操作，就需要跨基本块的数据传输算法，将上一个基本块中的打包向量传递到下一个基本块，从而更有效地提高向量化效率。如图 8 所示，Overall_Situation_Data_Transfers() 的基本思想是：首先根据第二步得到的 IPA 对每个基本块进行过程间信息分析，遍历每个 PU，收集相邻基本块之间存在定义使用关系的语句，并确定其在基本块中的地址。之后对生成的跨基本块的控制流图进行扩展，如果数组 x 在相邻基本块 BB_front，BB_back 中都出现，根据 DUG 图判断是否存在 def_use 关系，若存在，则将 BB_front，BB_back 中的 x 进行数组重命名为 y，并将其声明为全局变量，在 BB_back 之前对数组 y 进行赋值，对 CFG 进行更新；如果不存在，则结束；算法第三步结束。

```
1. algorithm Overall_Situation_Data_Transfers
2. Input: ARA_LOOP_INFO(may_def,may_use);
3. use(loop_t);
4. def(loop_t);
5.   for each PU's do begin
6.        get define data v in BB_front;
7.     get use data s in BB_back;
8.     Output: cross_BB_data_list(BB_front,BB_back);
9.     G:CFG=(N,E),BB_front:BB_front∈N,
10.            BB_back:BB_back∈N;
11.     if(array x ∈BB_front,BB_back) do
12.       for each array x define in BB_front do
13.         for each array x use in BB_back do
14.           对 BB_front,BB_back 中的 array x 进行重命名 array y
15.           在 BB_back 之前将其命名为全局变量并对其赋值
16.       end foreach
17.         end foreach
18.       end if
19.         if(array y is Gobal) do
20.           Y ∈E :CFG=(N,E) //extended_CFG
21.     end foreach
22.   return
23. end Overall_Situation_Data_Transfers
```

图 8　Overall_Situation_Data_Transfers()的实现

4　实验结果和分析

本文中使用的 DSP 自动向量化编译系统是基于开源编译器 Open64[12] 框架实现的 HTC06，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，1 级数据 cache 为 32kB，2 级 cache 为 256kB。处理器的向量化因子为 128bit，即可以同时处理 4 个定点复数类型数据、8 个短整形数据或者 4 个 int 型数据。

实验时，首先用 HTC06 将源程序转化为向量化程序，并将得到的向量化程序在 Xtensa 可配置、可扩展处理器上运行，最后针对验证代码的输入、输出进行比较，确认自动转换工具的输出是正确的，并将其与串行程序和手工向量化程序进行比较，计算出各自的加速比。

表 1 简要描述实验所用到的程序核心片段，其中两个来自多媒体和数字信号处理领域：一个是有限脉冲反应滤波器 finite impulse response filter(fir)，数据类型是 short-int；另一个为 2D 卷积运算 fcpx_convolution(convolve)，规模是 8x20，数据类型为复数；其他的测试用例采用 SPEC CPU2000 中的核心函数和整体应用作为测试集，分别就串行程序、向量化程序、经过跨基本块优化 3 个版本进行了测试，最后给出各版本的加速比。这些程序具有广泛的代表性，包含了向量化所具有的一些特征，诸如多种数据类型、规约、跨基本块访问以及特殊的处理方式。这里的测试结果均是在相应程序上运行的 cycle 数，优化后的 cycle 的减少，说明以上的优化措施对向量化的效果是显而易见的。

表 1　跨基本块优化测试集

程序名	串行程序	向量化	跨基本块	优化后提升
fir	464	368	331	11%
171. swim	2246	2410	1827	32%
172. mgrid	3196	2581	2258	14%
convolve	4196	2640	3659	−27%
buts	926	772	846	−8%
183. equake	3268	2403	3010	−20%

对这些程序的核心片段分别按照串行程序、常规的 SIMD 向量化程序以及跨基本块变换的向量化程序在目标系统上进行测试，之后分别给出两种向量化程序相对于串行程序的加速比，测试结果如图 9 所示。Sequence 代表串行程序的加速比，SIMD 代表常规向量化加速比，Across Basic Block 代表按照本文提出的跨基本块变换策略向量化之后得到的加速比。

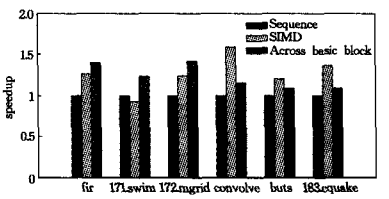

图 9　跨基本块优化后得到的加速比

从向量化结果可以看出，本文所提出的基于跨基本块变换的向量化方法可以明显地提升向量化效率。其中 fir，171. swim，172. mgrid 由于向量化代码生成阶段基本块边界产生了大量冗余操作，通过跨基本块边界的 SLP 代码生成算法，使得程序中大部分冗余操作消除，因此性能得到提升。但是 convolve、173. applu 核心函数(buts)和 183. equake 这 3 个测试用例，由于内层循环存在依赖关系阻碍向量化，通过跨基本块向量化方法不但不能消除内层循环的依赖，反而使数据依赖关系更加复杂，极大地影响了程序的执行效率，导致向量化收益降低甚至存在负收益。因此在跨基本块变换的基础上添加了循环分布优化算法，消除了内层循环依赖关系。测试结果如表 2 所列，可见在使用基于跨基本块变换和循环分布的向量化方法后测试用例的性能提升分别为 21%、23% 和 22%，优化后的收益为正。

表 2　跨基本块和循环分布算法测试集

程序名	向量化	跨基本块变换	跨基本块和循环分布	优化后提升
convolve	2640	3659	2176	21%
buts	772	846	630	23%
183. equake	2403	3010	1967	22%

分别就常规向量化方法、经过跨基本块变换向量化方法

(下转第 60 页)

[4] 付雄,彭冰. 基于 Shadowing 模型的无线入侵主机物理定位研究[J]. 微电子学与计算机,2010,27(12):4-9

[5] Li Bing-hao, Dempster A, Rizos C, et al. Hybrid Method for Localization Using Wlan[C]// Spatial Sciences Conference. Melbourne: Spatial Sciences Institute, 2005: 12-16

[6] Dricot J-M, De Doncker P. High-accuracy physical layer model for wireless network simulations in NS-2[C]//Wireless Ad-Hoc Networks, 2004 International Workshop. Oulu: CWC Oulu, 2004: 249-253

[7] Xu Jiu-qiang, Liu Wei, Lang Feng-gao, et al. Distance measurement model based on RSSI in WSN[J]. Wireless Sensor Network, 2010, 2(8): 606-616

[8] 彭建盛,李兴秦,志强. 三维立体空间定位算法的研究与实现[J]. 传感器与微系统,2012,31(7):33-35

[9] Lim Yu-xi, Schmoyer T, Levine J, et al. Wireless Intrusion Detection and Response[C]// Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society. New York: IEEE, 2003: 68-75

[10] Kowalik K, Bykowski M, Keegan B, et al. Practical issues of power control in IEEE 802.11 wireless devices[C]// International Conference on Telecommunications, 2008, ICT 2008. Berlin: ICWE, 2008: 1-5

[11] 杨哲. 无线网络安全攻防实战进阶[M]. 北京:电子工业出版社,2011:12-21

[12] Erceg V, Greenstein L J, Tjandra S Y. An empirically based path loss model for wireless channels in suburban environments[J]. IEEE Journal on Selected Areas in Communications, 1999, 17(7): 1205-1211

(上接第 28 页)

以及跨基本块和循环分布向量化方法 3 个版本进行了测试,最后给出各版本的加速比,如图 10 所示。SIMD 代表常规向量化加速比,Across Basic Block 代表本文提出的跨基本块变换策略向量化之后得到的加速比,Across Basic Block and Loop Distribution 代表按照本文提出的跨基本块和循环分布向量化算法所得到的加速比。从图中可以看出,本文所介绍的基于跨基本块变换和循环分布的 SLP 优化算法相对于常规的 SIMD 向量化有明显的性能提升,3 个测试用例 convolve、173. applu 核心函数(buts)和 183. equake 的加速比分别为 1.92、1.46 和 1.66,算法所带来的收益比较明显。

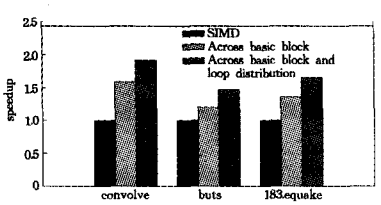

图 10 跨基本块和循环分布优化加速比

结束语 由于 SLP 算法能更好地对数字信号处理领域的应用进行向量化,而该算法不能充分发掘跨越基本块边界语句的并行性,因此本文在对现有的循环变换和跨基本块数据依赖分析的基础上,提出了基于跨基本块变换和循环分布的 SLP 优化技术,以尽可能地发掘不同基本块之间语句的并行性,消减不必要的冗余运算,使跨基本块变换代价达到最优。最后基于 SPEC2000 测试集和数字信号处理领域的程序核心片段对该优化方法进行了测试,结果表明,该方法结合 SLP,可以提升向量化程序的执行效率。

参 考 文 献

[1] Franchetti F, Kral S, Lorenz J, et al. Efficiem utilization of SIMD extensions[J]. Proceedings ofthe IEEE, 2005, 93(2): 409-425

[2] TMS320C6000 CPU and Instruction Set Reference Guide(Rev. F)[M]. TexasInstruments Inc. 2000

[3] SC140 DSP Core Reference Manual[R/OL]. http://cache.freescale.com/files/dsp/doc/ref_manual/MNSC140CORE.pdf, 2012-05-20

[4] Fridman J, Greenfield Z. The Tiger SHARC DSP Architecture [J]. IEEE Micro, 2000, 20(1): 66-76

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

[6] Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets[C]// Proc of the ACM SIGPLAN Conference on Programming Language Design and Implementation. June 2000: 145-156

[7] Shin J, Hall M, Chame J. Superword-level Parallelism in the Presence of Control Flow[C]// Proc. of the International Symposium on Code Generation and optimization. March 2005: 165-175

[8] Nuzman D, Zaks A. Outer-loop vectorization: revisited for short simd architectures[C]// Proceedings of the 17th international conference on parallel architectures and compilation techniques, PACT '08. New York, NY, USA, ACM, 2008: 2-11

[9] Aho A V, Lam M S, Sethi R, et al. 编译原理[M]. 赵建华,郑滔,戴新宇,译. 北京:机械工业出版社,2009

[10] 陈火旺,刘春林,谭庆平,等. 程序设计语言编译原理(第 3 版)[M]. 北京:国防工业出版社,2001

[11] Allen F E, Cocke J. A program data flow analysis procedure[J]. Communications of ACM, 1976, 19(3)

[12] Open64[EB/OL]. http://open64.sourceforge.net, 2012-09-10