计算机科学
COMPUTER SCIENCE

基于区域划分的跨基本块SLP向量化技术

韩林, 丁永强, 崔平非, 刘浩浩, 李浩然, 陈梦尧

引用本文

韩林, 丁永强, 崔平非, 刘浩浩, 李浩然, 陈梦尧. 基于区域划分的跨基本块SLP向量化技术[J]. 计算机科学, 2025, 52(9): 186-194.

HAN Lin, DING Yongqiang, CUI Pingfei, LIU Haohao, LI Haoran, CHEN Mengyao. SLP Vectorization Across Basic Blocks Based on Region Partitioning [J]. Computer Science, 2025, 52(9): 186-194.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

基于申威编译器的并行调度策略优化技术研究

Research on Parallel Scheduling Strategy Optimization Technology Based on Sunway Compiler

计算机科学, 2025, 52(9): 137-143. https://doi.org/10.11896/jsjkx.241200072

深度学习编译器研究综述

Review of Research on Deep Learning Compiler

计算机科学, 2025, 52(8): 29-44. https://doi.org/10.11896/jsjkx.250100062

基于大语言模型的移动应用隐私政策合规性检测方法

Privacy Policy Compliance Detection Method for Mobile Application Based on Large LanguageModel

计算机科学, 2025, 52(8): 1-16. https://doi.org/10.11896/jsjkx.250300156

基于粒子群算法的自动向量化收益评估模型研究

Research on Automatic Vectorization Benefit Evaluation Model Based on Particle SwarmAlgorithm

计算机科学, 2025, 52(7): 248-254. https://doi.org/10.11896/jsjkx.241000181

向量DSP的数组计算高效代码生成技术研究

Research on Efficient Code Generation Techniques for Array Computation for Vector DSPs

计算机科学, 2025, 52(6A): 240300156-7. https://doi.org/10.11896/jsjkx.240300156

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.241100130

基于区域划分的跨基本块 SLP 向量化技术

韩　林[1,2]　丁永强[1]　崔平非[1]　刘浩浩[2]　李浩然[2]　陈梦尧[2]
1 中原工学院网络空间安全学院　郑州 451191
2 国家超级计算郑州中心　郑州 450001
(strollerlin@163.com)

摘　要　自动向量化作为发掘数据级并行性、提升程序性能的重要方式,被广泛应用于主流编译器中。超字级向量化(Superword-Level Parallelism,SLP)专注于发掘相邻同构语句级别的数据并行性检测并聚合标量指令生成向量指令。然而,传统的SLP框架在发掘跨基本块的语句向量化时能力不足,特别是当连续的可向量化指令被基本块边界分割时,SLP分析无法有效发掘潜在的向量化语句。针对这一问题,提出了一种基于区域划分的跨基本块 SLP 向量化方法。该方法通过扩大分析范围至支配关系内的多个基本块,打破了基本块边界的限制,从而能捕捉更多潜在向量化机会,有效提升了 SLP 向量化效率。所提出的方法基于 GCC10.3.0 编译器实现,并挑选 SPEC CPU2006 测试集中包含相关程序段的测试程序进行了实验。实验结果显示,在 SPEC CPU2006 测试集挑选的测试程序中,与传统 SLP 方法相比,所提出的方法可使 SPEC CPU2006 测试程序加速比最高提升 12%,相关测试程序的平均加速比提升 8%,在 polybench 测试中获得了平均 3%的加速比,其有效性得到验证。该工作可为提升 GCC 编译中 SLP 向量化效率提供技术参考。

关键词:编译优化;自动向量化;SLP;跨基本块;区域划分
中图分类号　TP314

SLP Vectorization Across Basic Blocks Based on Region Partitioning

HAN Lin[1,2], DING Yongqiang[1], CUI Pingfei[1], LIU Haohao[2], LI Haoran[2] and CHEN Mengyao[2]
1 College of Cyber Security, Zhongyuan University of Technology, Zhengzhou 451191, China
2 National Supercomputing Center in Zhengzhou, Zhengzhou 450001, China

Abstract　Automatic vectorization is a key technique in mainstream compilers for uncovering data-level parallelism and enhancing program performance. Traditional SLP vectorization struggles with cross-basic-block statement vectorization, particularly when consecutive vectorizable instructions are split by basic block boundaries, limiting its ability to detect potential vectorization opportunities. To address this, this paper proposes a region-based cross-basic-block SLP vectorization method that extends the analysis scope to multiple basic blocks within dominance relations, effectively breaking basic block boundaries and uncovering more vectorization opportunities. Implemented in the GCC 10.3.0 compiler, the proposed method is evaluated using relevant program segments from the SPEC CPU2006 benchmark. Experimental results demonstrate that the proposed method achieves up to a 12% speedup in SPEC CPU2006, an average speedup of 8% for related test programs, and a 3% average speedup in the Polybench benchmark compared to traditional SLP methods, validating its effectiveness. This work provides a technical reference for improving SLP vectorization efficiency in GCC compilers.

Keywords　Compilation optimization, Automatic vectorization, SLP, Across basic blocks, Region partitioning

1　引言

随着高性能计算需求的快速增长,各大处理器厂商广泛支持单指令多数据技术,并在指令集架构中添加 SIMD(Single Instruction Multiple Data)扩展指令[1],如 Intel 的 AVX512 指令集、ARM 的 SVE 指令集以及 RISC-V 高效大规模向量指令集等。SIMD 向量指令有效增强了处理器的计算效能,尤其在科学计算和数字信号处理、人工智能和卷积核计算等关键领域[2-3],其对高效 SIMD 数据级并行化技术的需求更为迫切[4-5]。此外,CPU 硬件厂商也在探索如何更好地整合 SIMD 与其他并行计算技术(如进程级多核、线程级 GPU 加速),以实现更高效的多层次系统级并行处理[6]。

到稿日期:2024-11-25　返修日期:2025-04-15
基金项目:2024 河南省科技攻关项目(242102211094);2022 河南省重大科技专项 17(221100210600)
This work was supported by the 2024 Henan Provincial Science and Technology Tackling Project(242102211094) and 2022 Major Science and Technology Programs in Henan Province 17(221100210600).
通信作者:崔平非(cpf1975@126.com)

然而，高效利用 SIMD 向量部件并非易事，需要软件开发人员熟悉兼顾目标平台的向量指令特性，并且能够手工编写高效的向量化代码。这给程序员带来了一系列挑战：首先，依赖内嵌汇编或内部向量接口函数可能导致编写的向量代码在不同架构间的可移植性较差[7-8]；其次，随着计算机技术的飞速发展，大量开发人员留下无数宝贵的应用软件，包含的优秀代码难以计量，仅依靠手工改写向量代码不仅难度高，工作量巨大，而且容易出错，是一件极度耗时且费力的事情。因此，当前工业界更倾向于使用编译器自动将标量代码转换为向量代码，例如 GCC 和 LLVM 等[9]。当前主流编译器提供两种类型的自动向量化方法：循环级向量化和超字级并行（Superword Level Parallelism，SLP）向量化。循环级向量化聚焦于挖掘程序中循环结构向量化机会，通过将多次标量操作融合为一次向量操作，实现了数据的并行处理[10]。SLP 向量化主要关注程序代码段中多条相邻语句间的向量化机会，通过将多条结构相同或相似的语句合并为一条向量语句，实现了并行加速执行[11]。本文主要关注 SLP 向量化。

SLP 向量化技术由 Larsen 等[12]首次提出，旨在发掘程序中相邻同构语句间的数据级并行性。该技术的主要思路是寻找代码段中地址连续的同构指令，然后通过定义使用链和使用定义链寻找更多的同构指令并将其打包，最后合并打包后的多条标量指令形成对应的 SIMD 向量指令[13]。但该 SLP 方法是以基本块为单位进行同构语句的发掘，并在构建 SLP 树过程中遇到基本块边界的 PHI 语句时停止树的构建。这种策略不会发掘跨越基本块的连续的可向量化指令，导致 SLP 构建过程提前终止，进而错失了部分可向量化机会。

目前，学者们从多方面探索了对 SLP 技术的改进。Porpodas 等[14]提出的 PSLP 方法，在寻找 SLP 同构指令过程中通过填充冗余指令来扩大可向量化的范围。Feng 等[15-16]提出多种同构化变换的 SLP 向量化方法，通过对结构相似的指令进行变换来形成更多的同构指令。Zhang 等[17]则在原有的 SLP 方法上新增了异构语句同构化模块，通过寻找基本块中遗漏的向量化机会并构建 SLP 补充图来增加可向量化的指令。这些方法都是通过增加同构语句的数量来扩大 SLP 的向量化范围的思路进行改进的。Li 等[18]提出了基于剪切的 SLP 方法，关注 SLP 向量化的收益，选择向量化收益最高的指令集合进行向量化，而不是追求向量化指令的数量。除此之外，跨基本块的同构语句向量化也是改进 SLP 方法的一个方向。Xu 等[19]提出了一种跨基本块向量化指令选择方法，综合考虑基本块内与基本块间的语句关系，采用动态规划的方式确定收益较大的向量化策略，从而达到跨越多个基本块来扩大向量化检测范围的目的。Chen 团队[20]提出了一种名为 Predicated Static Single Assignment（SSA）的新中间表示（IR），用于简化在不同基本块和循环间移动代码的过程，从而处理跨越这些控制流区域的 SLP，但它也增加了编译器实现的复杂性，并可能带来代码膨胀、运行时开销以及对硬件支持的依赖等问题。此外，使用谓词可能增加分析与优化的难度。Ye 等[21]提出了一种新的循环展开模型，该模型考虑了后续的超字级并行向量化和寄存器压力。在计算展开因子时，该模型使用 SLP 收益模型来识别那些在循环展开后可通过 SLP 进行向量化的循环，并计算出支持后续 SLP 向量化的展开因子，实现了循环体内的 SLP 向量化。Li 等[22]提出了一种新颖的掩码指令转换技术，将 IF 结构转换为选择指令。这种转换使得控制流矢量化不再依赖内联函数，有效地将控制依赖转变为数据依赖。此外，他们还提出了对现有 Phi 节点生成算法的改进，以增加控制流向量化的机会。

在自动向量化的研究方向，Chen 等[23]基于对 GCC 自动向量化的研究提出了通过先导实例划分待处理数据并计算向量化收益，最终为收益为正的子图生成向量代码，优化了 SLP 的收益计算方法。Tayeb 等[24]提出了自动向量化工具 Autovesk，它能将标量代码转换为向量化代码，提升不规则数据访问模式下的程序性能，但其目前存在适用范围有限、依赖特定输入格式、未处理对齐问题等局限。

本文考虑将支配关系约束下的多个基本块划分为一个区域，以每个区域为基本单位进行向量化处理，提出了一种新的面向跨基本块 SLP 向量化方法。该方法通过跨越基本块边界，扩大对可向量化指令的检测范围，结合收益分析策略对嵌套循环区域和含有控制流的区域进行向量处理。相较于其他方法，以区域划分为基础的跨基本块 SLP 向量化方法（以下称为区域 SLP 向量化方法）的优势在于利用了基本块间的支配关系来划分区域，将其作为向量化处理的基本单元，从而有效扩展了 SLP 向量化的发掘范围，并且能够以较小的实现代价获得性能提升。

2　背景与动机

2.1　SLP 向量化

超字级并行性是一种关注于多媒体应用程序中短向量 SIMD 操作并行性的方法。与传统向量处理中所利用的循环级并行性不同，它的核心在于识别和利用基本块内的并行性，而不仅仅是循环结构中的并行性。SLP 关注的是如何将相似的操作分组，形成超级字（Superword），这些超级字包含打包在一起的数据，可以在 SIMD（单指令多数据）指令集上高效执行。通过这种方式，SLP 能够跨循环迭代和在基本块内部同时发掘并行性，从而提高执行效率。

SLP 算法首先寻找各个基本块中的同构性指令，并将其打包为种子指令；然后根据种子指令，按照使用定义链和定义使用链寻找更多可向量化的同构指令构建 SLP 图；最后根据 SLP 图进行代码调度，将标量代码转化为向量代码。SLP 向量化的过程如图 1 所示。

识别种子指令是 SLP 算法的开始和基础。一般情况下，符合下列条件的指令就可以作为种子指令执行打包操作。

1）同构性指令。同构性是指具有相同操作特性的指令，即操作的数量和数据类型相同，并且指令执行的操作相同。

2）无数据依赖。这些指令之间在执行顺序上相互独立，以免在执行向量化的过程中出现数据依赖的问题。

3）访问相邻的内存地址。对连续内存地址的访问是典型的可向量化的操作，也是最优先考虑进行 SLP 向量化的代码。对连续内存的访问指令进行向量化可以极大地提升数据的加载和存储速度。SLP 图构建完成后，根据代价模型进行分析，只要向量化后的成本小于标量执行的成本，就认为向量

化有益，可以进行向量化。在代码生成阶段，SLP 算法从叶子节点开始自上而下对每个节点执行代码调度形成向量指令。

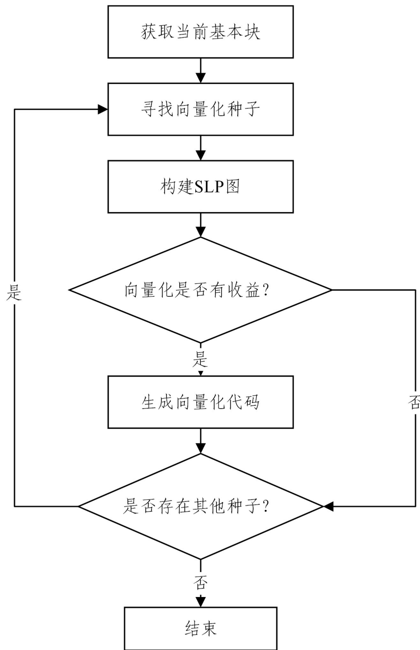

图 1 SLP 向量化过程

Fig. 1 Process of SLP vectorization

2.2 动机示例

经典的 SLP 向量化方法致力于发掘单一基本块中的超字级并行性。向量化是以单个基本块为单位的。在向量化处理过程中，向量化器逐一遍历控制流图中的所有基本块，进行向量化处理。由于检测范围集中在单一基本块内部，因此向量化器所收集到的数据级并行性极为有限。缺乏足够的并行性会导致收益空间狭小，分析模型的局限性也会使得收益评估进一步恶化。

如图 2 所示的示例代码 1，以基本块为单位进行 SLP 向量化分析时，将该代码分为 3 个基本块，其中第一个基本块中对 ai 变量的操作属于加载指令。由于 GCC 编译器在实践中识别连续的存储操作作为种子指令，第一个基本块内的操作是加载操作，而不是对内存的连续存储操作，因此不被识别为可向量化的种子指令，所以第一个基本块没有被向量化，依旧以标量指令执行。第二个基本块存在对数组 b 内存地址的连续访问操作，由于是可向量化的种子语句，以该语句组为根节点构建 SLP 树。同理，第三个基本块对 out 数组进行操作并构建 SLP 树。于是，以基本块为单位成功构建了两棵 SLP 树。

```
a0＝in[0]＋23；a1＝in[1]＋142；
a2＝in[2]＋2；a3＝in[3]＋31；
if(x＞y){
b[0]＝a0；b[1]＝a1；
b[2]＝a2；b[3]＝a3；
}else{
out[0]＝a0＊(x＋1)；out[1]＝a1＊(y＋1)；
out[2]＝a2＊(x＋1)；out[3]＝a3＊(y＋1)；
}
```

图 2 示例代码 1

Fig. 2 Example code 1

如图 3 所示，在进行代价分析时，第二个基本块进行向量化有收益；而第三个基本块在进行向量化时，通过代价模型分析对该段代码进行向量化的总成本（预处理操作＋向量计算操作＋尾处理操作）要大于直接执行标量操作的总成本，向量化收益的判断是负收益，所以没有被向量化。因此整体上只有第二个基本块被成功向量化，而另外两个基本块向量化失败。而这种向量化的结果正是以基本块为单位进行指令并行性检测带来的弊端。

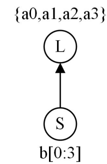

图 3 SLP 树

Fig. 3 SLP tree

2.3 优化思想

基于上述问题，本文提出基于区域划分的跨基本块 SLP 向量化方法。该方法的核心思想是，考虑将多个基本块划分为一个区域，以每个区域为基本单位进行 SLP 向量化。以图 2 示例代码 1 为例，将 3 个基本块划分为一个区域进行向量化，生成如图 4 所示的 SLP 图。该方法不仅能向量化原本没有被向量化的第一个基本块；而且因为新节点的加入，在进行收益分析时整体具有正收益，从而使得第三个基本块也能够向量化。区域 SLP 向量化在向量化分析时以多个基本块构成的区域为基本单位，以此来发掘出更多可向量化代码，提高了程序运行效率。

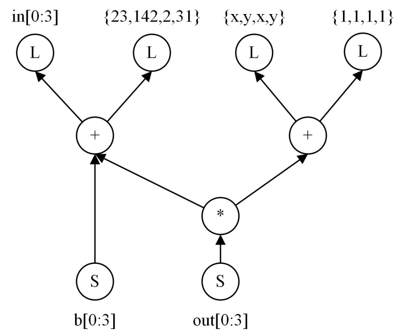

图 4 区域 SLP 图

Fig. 4 Regional SLP graph

3 基于区域划分的跨基本块 SLP 向量化

3.1 向量化框架

基于区域划分的跨基本块 SLP 向量化，首先将待向量化的函数区域根据支配关系和控制流结构划分为若干子域，它们实际上是待处理的向量化区域。在一个由多个基本块构成的向量化区域中，可能存在超字级并行性的多个实例。为了同时对多个实例构成的集合进行收益分析和调度，需要划分出可以独立进行收益分析的 SLP 子图，并引入新的 SLP 子图调度的算法。

向量化处理方法如图 5 所示。相较于传统 SLP 方法，区域 SLP 处理流程以划分完成的区域作为输入。在完成区域

内所有种子指令为根节点的 SLP 树构建后再进行子图划分，然后分析每个子图的收益，并在判断具有正收益时进行代码调度，生成向量代码。

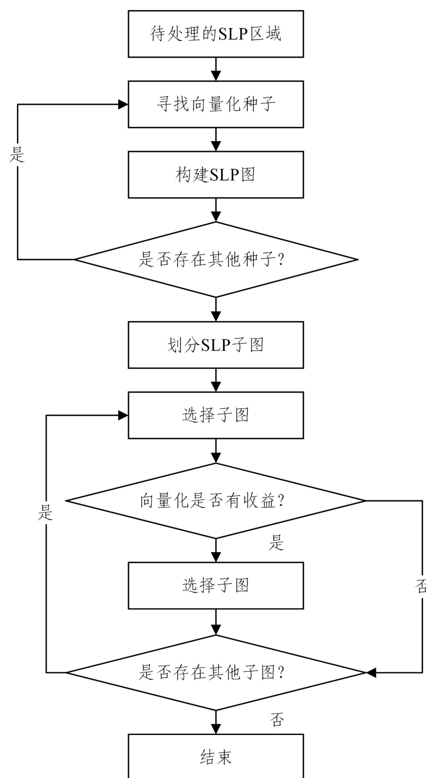

图 5　区域 SLP 流程图

Fig. 5　Flow chart of regional SLP

为了在编译器中实现该方法，需要解决区域划分、跨越 PHI 节点向量化和基于 SLP 区域的收益分析 3 个问题。在编译器中，SLP 属于中间代码优化，因此都是以静态单赋值（Static Single Assignment，SSA）形式的中间表示为基础进行的。

3.2　基于 SSA 控制流的区域划分

该部分的基本问题是生成待处理的具有 SLP 向量化机会的代码区域。为了解决这个问题，本文提出了一种启发式的算法，如算法 1 所示。SLP 图的构建是以种子代码为起点的，种子代码包括对连续内存地址的存储操作。将种子代码打包作为 SLP 图的起始节点，根据定义-使用链追踪操作数的定义语句，并将可向量化的语句打包作为子节点。由此可知，子节点的语句处于受支配关系约束的基本块中。因此，有以下划分规则。

1）支配边界处。从第一个基本块开始直至遇到不由第一个基本块支配的基本块结束，划分为一个区域。

2）循环边界处。在循环结构中，当基本块从所属的循环进入另一个不同且非嵌套的循环时，也会进行区域划分。这样做是为了避免跨循环边界引入难以处理的循环不变量和数据依赖。

3）控制流改变点。如果基本块的最后一个语句是一个改变控制流的语句（例如跳转、返回等），并且该语句有左值（即定义了一个变量），为了避免引入复杂的控制流和数据依赖性，以及避免插入额外的指令来确保正确处理控制流和数据依赖性，选择在此处进行区域划分。

算法 1　区域划分算法

输入：函数控制流图上的基本块拓扑序列{bb1，bb2，…，bbn}

输出：划分好的 SLP 区域集合 R

```
1. R←Ø / * 初始化区域集合 R * /
2. for i←1 to n do / * 遍历基本块设置划分区域标志 * /
3.     split←false，bbs←Ø / * bbs 用于存放同一区域中的代码块 * /
4.     if bbs≠Ø && !(bb[i].dominated_by(bb[0]))
5.       split←true / * 根据规则 1 划分区域 * /
6.     else if bbs≠Ø &&(bb[i].loop ！ =bb[0].loop)
7.       split←true / * 根据规则 2 划分区域 * /
8.     if split=true 且 bbs≠Ø
9.       R←R∪{bbs}/ * 将当前区域添加到区域集合 R 中 * /
10.      bbs←Ø / * 重置 bbs 为划分新的区域做准备 * /
11.      bbs←bbs∪{bb[i]} / * 将当前基本块作为区域内首个基本块 * /
12.    else
13.      bbs←bbs∪{bb}
14.      last=last_stmt(bb[i])/ * 获取当前基本块最后一条语句 * /
15.      if(get_lhs(last) && is_ctrl_altering_stmt(last))
16.        R←R∪{bbs}/ * 根据规则 3 划分区域 * /
17.        bbs←Ø
18. return R
```

算法 1 中，第 1 行，初始化一个空集 R 来存储最终的 SLP 区域集合 R。第 2 行，开始遍历输入的基本块序列{bb1，bb2，…，bbn}。第 3－4 行，对于每个基本块 bbi，初始化两个变量：split 为 false，表示当前不需要划分新的区域；bbs 为空集，用来临时存放属于同一区域的基本块。第 5－7 行，检查是否需要根据规则 1 或规则 2 划分新的区域。规则 1：如果 bbs 非空且 bb[i]不被 bbs 中的首个基本块支配，则设置 split 为 true。规则 2：如果 bbs 非空且 bb[i]与 bbs 中的首个基本块不在同一个循环嵌套区域内，也设置 split 为 true。第 8－10 行，如果 split 为 true 并且 bbs 非空，那么将 bbs 加入区域集合 R 中，并清空 bbs 以开始收集下一个区域的基本块。第 11－13 行，如果没有发生划分（即 split 仍为 false），那么将当前基本块 bb[i]添加到 bbs 中。第 14－17 行，获取 bb[i]中最后一个语句 last，如果该语句有左值（意味着它是一个赋值操作）且是控制改变语句（如分支、跳转等），那么同样根据规则 3 划分新的区域，将 bbs 加入 R 中并重置 bbs。完成遍历后，返回区域集合 R。

对算法 1 的时间复杂性进行分析。第 1 行初始化集合 R，这是一个常数时间操作。第 2 行开始一个循环，其迭代次数为 n 次，其中 n 是基本块的数量。因此，这个循环的复杂度是 O(n)。在循环体内，每次迭代包括多个条件检查和集合操作。每个条件检查（第 4，6，15 行）通常涉及比较或查询操作，如果这些操作可以在常数时间内完成，则每个条件检查的时间复杂度为 O(1)。集合操作，如 R←R∪{bbs}（第 9，16 行）和 bbs←bbs∪{bb}（第 11，13 行），通常在平均情况下可以视为 O(1)，尤其是当使用高效的数据结构时（如哈希集）。第 4 行和第 6 行中的条件检查涉及支配关系和循环嵌套的检查。

这些检查可能依赖于预处理的信息，如支配树和循环嵌套信息。如果这些信息已经以线性时间复杂度预处理过，那么在运行时这些检查可以视为 O(1)。但如果这些检查在运行时动态进行，它们可能会增加额外的时间成本，具体取决于实现方式。在最坏的情况下，这些检查可能需要遍历整个控制流图，这将导致 $O(n)$ 的复杂度。第 14 行获取基本块中最后一个语句的属性，假设语句属性已经被预先计算并存储于数据结构中，那么该操作通常是一个常数时间操作。因此在最优情况下，算法的时间复杂度为 $O(n)$，其中 n 是基本块的数量。但在最坏情况下，如果基本块的支配关系或者嵌套循环结构没能在算法开始前构建支配树或者对循环的嵌套结构进行解析，时间复杂度可能上升至 $O(n^2)$。

3.3 跨越 PHI 节点向量化

传统的 SLP 向量化以基本块为单位进行并行性检测并建立 SLP 树。当遇到 PHI 语句时，会直接终止 SLP 树的构建，这种方法无法满足 SLP 向量化跨越基本块并行性检测的需求。

向量化过程中会遇到由控制分支交汇产生的 PHI 和由循环结构产生的 PHI。由控制分支交汇产生的 PHI 可以自然地扩展 SLP 发掘过程。对于由循环结构产生的 PHI，需要处理 SLP 构建中产生的回边并打包 PHI 语句为 SLP 节点。进行代码调度时，SLP 图中可能会出现由循环结构导致的强连通分量（Strongly Conntected Component，SCC），这一部分需要进行特别处理，以保证能按正确的顺序生成向量代码。对 PHI 节点的处理使得可以在外层循环向量化中使用区域 SLP 方法直接向量化带有嵌套 SLP 环的代码区域，也可以对含有控制流交汇的非环区域进行向量化处理。

在向量化过程中，首先对种子代码进行识别，之后根据定义-使用链发现更多可向量化代码。遇到 PHI 时，遍历其所有参数，并找到参数的定义语句，若符合 SLP 向量化要求，则将其打包并作为 SLP 图的子节点。之后需要根据 SLP 图进行代码调度，生成向量化代码。

以图 6 所示示例代码 2 为例，传统向量化在进行向量化分析时仅能发现代码末尾处对数组 a 的连续存储，最终只能对该部分进行向量化并形成如图 7 所示的 SLP 图。可以看到，该代码在传统的 SLP 方法下只能成功向量化很小一部分，错失了潜在的向量化收益。

```
void foo(void){
  for(int i=0;i<1020;i+=4){
    int suma=a[i];int sumb=a[i+1];
    int suma =a[i+2];int suma=a[i+3];
    for(int j=0;j<77;++j){
      suma=(suma^i)+1;sumb=(sumb^i)+2;
      sumc=(sumc^i)+3;sumd=(sumd^i)+4;
    }
    a[i]=suma;a[i+1]=sumb;
    a[i+2]=sumc;a[i+3]=sumd;
  }}
```

图 6 示例代码 2

Fig. 6 Example code 2

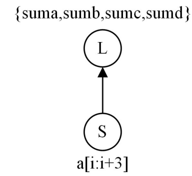

图 7 嵌套循环中的 SLP 树

Fig. 7 SLP trees in nested loops

基于区域划分的 SLP 向量化能够实现对整个循环结构的向量化处理。在对示例代码 2 进行向量化时，先将代码中的循环体划分为一个独立的区域。在该区域中，首先发现种子指令，即对 a 数组的连续存储操作。以该组指令为根节点，沿着使用-定义链依次查找操作数的定义语句，并将其作为子节点自下而上构建 SLP 树。这个过程会遇到两种 PHI，由外层循环变量 i 产生的 PHI 和内层循环中对变量 suma，sumb，sumc，sumd 的操作产生的 PHI。其中，由循环变量 i 产生的 PHI 的操作数都在外层循环开始时定义，不涉及数据依赖，所以可以将两个操作数作为子节点直接加入 SLP 树。而由内层循环产生的 PHI 的操作数中，一个来自外层循环的定义可以直接作为子节点加入 SLP 树；另一个则是来自内层循环的迭代，其定义语句已经在先祖节点中，因此会生成指向先祖节点的回边，至此 SLP 树构建完成。

如图 8 所示，虽然成功为整个嵌套循环区域构建了 SLP 图，但因为循环迭代间存在数据依赖，所以该图在内层循环范围内出现了 SSC 区域。在调度阶段需要进行额外的处理，以保证能按正确的顺序生成向量代码。

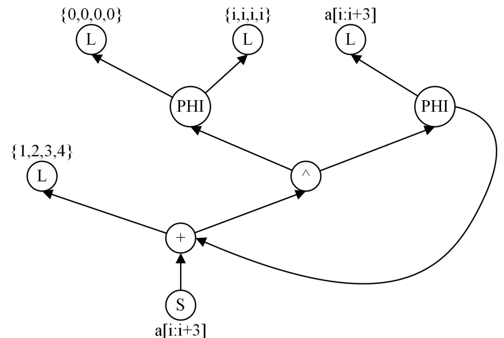

图 8 嵌套循环中的区域 SLP 图

Fig. 8 Region SLP graph in nested loops

完整的调度过程如算法 2 所示。算法中的调度对象分为 3 种类型：定义在区域外部的节点、单独的内部节点和强连通分量。调度算法为每个内部节点维护一个深度优先序索引 dfs 和 SCC 内部的回溯索引 lowlink。对于单独的内部节点或者 SCC 入口节点，lowlink 总是回溯到节点自身，这表明有一个合适的对象进行调度。若两个值不相等，则说明该节点处于 SCC 内部，并非调度 SCC 的入口。每一个定义在 SLP 区域内部的节点在被调度例程选中后，都会被压入调度栈中，并在生成代码后从调度栈中移除，以保证代码生成的正确性。外部定义作为 SLP 实例的叶子节点，不再入栈，以避免进一步的递归。

由于父节点的操作数依赖于孩子节点，SLP 算法采用深度优先方式对实例中的节点进行递归调度，首先为孩子节点

生成向量代码。为 SCC 区域生成代码之前，要求所有与之相关的孩子节点都必须完成向量代码生成。为此，调度算法有如下的处理规则：1）如果是叶节点，直接为其生成向量代码；2）如果孩子节点没有被访问过，就递归地调度孩子节点；3）如果孩子节点已经出现在栈中，以回边指向的节点作为 SCC 入口，标记该节点在 SCC 内部。

根据深度优先遍历的顺序，SCC 的调度入口在其内部具有最小的深度，入口在 SCC 外部的孩子节点完成代码生成后，所有和 SCC 相关的孩子节点就都完成了代码生成，此时可以对 SCC 进行处理。在此之前，不应该对 SCC 中的任何节点执行代码生成操作。在调度到 SCC 时，栈顶元素是 SCC 中深度优先序索引最大的节点。调整栈顶指针到入口节点，根据当前栈大小和 SCC 入口索引可以计算 SCC 工作队列的长度。为 SCC 生成代码时，仍按照深度优先遍历顺序处理回边，需要将源点的使用对应到汇点定义。生成代码后，将 SCC 队列从栈中移除，继续处理栈中剩余的节点。

算法 2　SCC

```
输入：SLP 实例 inst
输出：向量化后的 SLP 实例
/* 定义全局变量用于深度优先搜索(DFS)的计数 */
1. maxdfs←0，node←inst. root
/* 从根节点开始递归调度所有子节点，更新最大 DFS 值 */
2. schedule_scc(node，maxdfs)
3. dfs←maxdfs，lowlink←maxdfs / * 初始化 DFS 和 lowlink 值 * /
4. maxdfs ++ / * 增加 DFS 计数器作为下一个节点的 DFS 编号 * /
5. stack←Ø / * 初始化栈用于追踪节点访问路径 * /
6. stack. push(node) / * 开始调度过程，将当前节点压入栈中 * /
  / * 遍历当前节点的所有子节点 * /
7. for each child do
8.     if(!child. is_scheduled( ))
9.         schedule_scc(child，maxdfs) / * 递归调度子节点 * /
10.        lowlink←MIN(child. dfs，lowlink) / * 更新 lowlink 值，用于
           检测 SCC * /
11.        if lowlink ≠ dfs
12.            return
   / * 若当前节点等于栈顶节点，说明其是一个单独的内部节点 * /
13. if node＝stack. top{
14.     schedule_node()/ * 调度单独的内部节点 * /
15. }else{/ * 否则，当前节点属于一个 SCC * /
   / * 调度 SCC 内的所有节点 * /
   schedule_scc_node()
17. }
18. return
```

算法 2 中，第 1 行，初始化全局变量 maxdfs 为 0，它将被用来跟踪深度优先搜索（DFS）过程中的最大编号；同时，设置 node 为输入 SLP 实例 inst 的根节点。第 2 行，调用 schedule_scc 函数，开始从根节点递归地调度所有子节点，并更新最大 DFS 编号 maxdfs。第 3－4 行，初始化当前节点的 DFS 编号和 lowlink 值为 maxdfs，然后增加 maxdfs 计数器，以便下一个节点可以得到新的 DFS 编号。第 5－6 行，初始化一个空栈 stack，该栈将用于追踪节点访问路径。将当前节点压入栈中，表示正在访问这个节点。第 7－10 行，对于当前节点的每一个子节点，如果该子节点还没有被调度过，则递归地调度子节点，并更新当前节点的 lowlink 值为其自身 lowlink 与子节点 DFS 编号之间的最小值。第 11－12 行，如果 lowlink 不等于当前节点的 DFS 编号，说明没有发现 SCC，直接返回。第 13－14 行，如果当前节点等于栈顶节点，说明这是一个单独的内部节点，那么调用 schedule_node( )来调度这个节点。第 15－17 行，如果当前节点不是栈顶节点，说明当前节点属于一个 SCC。此时，调用 schedule_scc_node( )来调度 SCC 内的所有节点。整个算法过程利用了深度优先搜索和 Tarjan 算法来识别图中的 SCC，并根据是否属于 SCC 来进行不同的调度操作。

该算法是对一个有向图进行深度优先搜索（DFS）并识别其中的强连通分量，其时间复杂度主要由 DFS 过程决定。对于每个顶点，只会在其作为根节点时被调用一次，因此这部分的时间复杂度为 $O(V)$。对于图中的每条边，只会遍历一次，这部分的时间复杂度为 $O(E)$。SCC 检测过程是在 DFS 遍历的过程中完成的，无需额外的时间复杂度，因为它是在 DFS 框架内进行的。每次将一个顶点入栈或出栈的时间复杂度为 $O(1)$，整体而言，因为每个顶点最多入栈和出栈一次，所以这部分的时间复杂度也为 $O(V)$。因此，整个算法的时间复杂度为 $O(V+E)$。这意味着算法的运行时间与图的规模呈线性关系，具体取决于顶点数量 V 和边数量 E。在稠密图中（即 E 接近 V^2），时间复杂度接近 $O(V^2)$；而在稀疏图中（即 E 接近 V），时间复杂度接近 $O(V)$。这使得该算法在处理大多数实际场景下的图形问题时都非常高效。

3.4　面向区域向量化的收益分析

传统 SLP 方法在完成并行性检测后，会计算 SLP 实例的向量化收益，并根据分析结果来决定是否进行代码调度生成向量指令。然而，基于区域划分的 SLP 向量化因为在多个基本块间进行向量化检测，很可能获得多个种子指令且在构建 SLP 图时遇到已经收集过的节点会直接进行复用，因此最终构建的 SLP 图往往会包含多个 SLP 实例。若仍然按照传统 SLP 方法直接对该 SLP 图进行向量化，可能会出现图中某一实例在进行收益分析时不通过，导致整个 SLP 图中所有实例都无法被向量化的情况。更为合理的做法应该是，将 SLP 图中的实例划分为能独立计算向量化代价的子图，并以此为单位进行收益分析和调度。

划分子图的核心思想是检测实例之间的计算相关性，选择包含相同标量语句的 SLP 实例融合为一个子图。通过简单的分区策略，可以有效确定哪些部分的向量化能使性能提升，从而做出更加精准的向量化决策。其次，子图融合了多个 SLP 实例，有助于扩大收益空间，提升向量化的效率。图 3 所示的 SLP 图中就包含以 b 数组存储语句为根节点的 SLP 实例和以 out 数组存储语句为根节点的 SLP 实例。因为两个实例有共同的节点引用，所以将两个实例划分进一个子图中进行收益分析。

除了划分 SLP 子图之外，另一个重要的问题是在循环嵌

套区域实施收益分析。对于直线型代码,收集子图中的全部节点,将其作为一个完整的执行单元进行收益分析,可以较为准确地反映子图区域在向量化后的性能改变。而对于嵌套循环区域则不然。在外层循环区域向量化中,SLP 实例的节点可能分布在循环嵌套的不同层级,因此会具有不同的执行特征。例如,内层循环中的节点可能有更多的执行机会,但向量执行的收益为负;而外层循环中的节点执行次数较少,但每次执行的收益较大。如果将子图中的节点作为一个整体进行收益分析,可能导致向量化内层循环带来的不利影响被外层循环的收益所掩盖,造成次优的向量化决策。为此,对于嵌套循环区域上的收益评估采用一种简单的分层方法:按照深度递增的顺序,收集同一循环嵌套层级上的全部 SLP 节点,累加其标量和向量代价进行比较,以此判定是否对该层级进行向量化。如果存在任何层级上的收益分析无法通过,既定的向量化方案将无法完成。

4　实验与分析

为了验证基于区域划分的跨基本块 SLP 向量化方法的有效性,本文基于 GCC 10.3 编译器实现了这一技术,并通过 SPEC CPU2006 基准测试集进行测试评估。实验对比了原始 GCC SLP 向量化与新提出的跨基本块 SLP 向量化在开启向量化选项时的性能加速比。

实验平台采用 Intel Xeon 处理器(Skylake 架构,支持 IBRS(Indirect Branch Restricted Speculation) 特性),操作系统为 CentOS 7,编译器版本为 GCC 10.3.0,并启用了 AVX 指令集扩展来最大化利用现代 CPU 提供的并行计算能力。所有性能测试均在单核模式下执行,以确保结果的可比较性。表 1 列出了本次实验分析用到的代表性程序。

表 1　测试用例

Table 1　Testsuit

测试用例	介绍
435.gromacs	分子动力学模拟,评估生命科学应用的性能
450.soplex	线性规划求解器,评估优化问题的解决能力
453.povray3D	图像渲染,测试图形处理和光线追踪性能
454.calculix	结构有限元分析,衡量工程设计应用的性能
458.sjeng	国际象棋引擎,测试 CPU 在游戏逻辑和搜索算法上的表现
462.libquantum	量子计算模拟库,评估科学计算和物理模拟任务的性能

测试用例包括:SPEC CPU2006 测试集中的 450.soplex, 435.gromacs, 453.povray3D, 454.calculix, 462.libquantum 和 458.sjeng。这些程序中都包含大量 SLP 向量化机会且存在跨基本块的数据引用,非常适合用来测试区域 SLP 在性能方面的提升。实验将这些选定程序分别在标量模式、传统 SLP 向量化模式和区域 SLP 向量化模式下执行,随后通过比较两种 SLP 方法相比于标量模式下的加速比,来量化区域 SLP 向量化所带来的性能提升效果。此举旨在系统地验证区域 SLP 向量化在提高计算效率方面的潜在优势。

以 SPEC CPU2006 测试集中的 453 测试题中的代码为例,如图 9 示例代码 3 所示。该段代码很好地根据区域划分向量化了各个基本块中的代码,达到了跨基本块向量化的效果。在原 SLP 方法中,因为 if 代码段在收益阶段判断向量化收益为负,所以该段代码并没有进行向量化。但在划分区域后,if 代码段外存在对 Te 数组元素的引用,因此对 PHI 节点进行向量化,从而实现了整个 if 代码段的向量化且判断向量化有收益。因此,该代码段成功实现了向量化。

```
if(sample_method==3)
{
    Te[0] *=Interval->ds;
    Te[1] *=Interval->ds;
    Te[2] *=Interval->ds;
}
else{
    Te[0] *=Interval->ds * exp(-Ex[0] * d0);
    Te[1] *=Interval->ds * exp(-Ex[1] * d0);
    Te[2] *=Interval->ds * exp(-Ex[2] * d0);
}
SampCol[0]=Te[0];
SampCol[1]=Te[1];
SampCol[2]=Te[2];
```

图 9　示例代码 3

Fig.9　Example code 3

实验进行了 3 组测试:1)标量测试,该测试下关闭了向量化;2)SLP 测试,该测试使用了 GCC 中原有的 SLP 向量化方法;3)基于区域划分的跨基本块 SLP 测试。测试结果以标量测试为基准,将标量测试下程序运行时间除以 SLP 方法下程序运行时间,得到加速比。实验结果如图 10 所示。浅色柱体代表原 SLP 方法测试对比标量测试的加速比,深色柱体为跨基本块 SLP 方法测试对比标量测试的加速比。测试用例 435.gromac 和 450.soplex 中程序因为可以进行 SLP 向量化的代码段较少,所以 SLP 向量化的性能提升仅为 7% 和 8%;区域 SLP 向量化相对于原 SLP 方法的性能提升也较少,分别为 4% 和 5%。而测试用例 453.povray, 454.calculix, 458.sjeng, 462.libquantum 中可向量机会较多,因此进行 SLP 向量化和区域 SLP 向量化后都能获得较好的性能提升。其中,453.povray 和 462.libquantum 测试用例的程序主体分别涉及对结构体的大量操作和对数组的访问操作,且都包含 if-else 结构,符合区域 SLP 性能提升代码段。458.sjeng 程序主体中含有较多嵌套循环结构,因此进行区域 SLP 向量化也可以获得较大的性能提升。测试用例 454.calculix 中的程序主体为 switch-case 结构的代码段,且每个 case 分支都涉及对数组的操作,虽然该区域包含的基本块较多,但各基本块间的基本不涉及数据引用。因此,SLP 向量化后可获得较好的性能提升,但区域 SLP 向量化带来的优化效果较小。由此得出结论:对于可以 SLP 向量化的代码段,区域 SLP 相比传统 SLP 向量化方法,可能会带来更好的性能提升;但对于原本就含有较少 SLP 机会的代码段区域,SLP 向量化能带来的性能提升也很有限。实验中,核心代码片段中区域 SLP 向量化方法对比标量测试平均性能提升为 17%,对比原 SLP 方法平均性能提升为 8%。

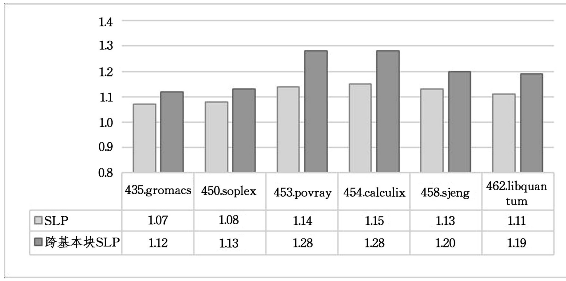

图 10 SPEC CPU2006 测试加速比对比

Fig. 10 SPEC CPU2006 test acceleration ratio comparison

为进一步验证该方法的有效性,使用 PolyBench 测试集进行了相同方法的实验。实验所选取的测试程序如表 2 所列,测试结果如图 11 所示。

表 2 PolyBench 测试程序

Table 2 PolyBenchtest program

测试用例	介绍
gemm	评估矩阵乘法的性能
gesummv	评估矩阵向量乘法和向量加法的性能
correlation	评估相关性计算的性能
deriche	图像处理的测试项目,主要用于评估边缘检测算法的性能
symm	评估所有节点对最短路径计算的性能
fdtd-2d	评估电磁场仿真算法的性能
heat-3d	评估三维热传导方程求解算法的性能

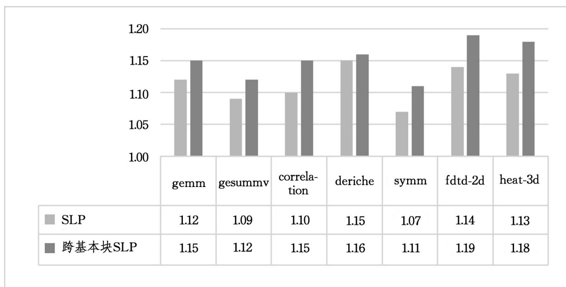

图 11 PolyBench 测试加速比对比

Fig. 11 PolyBench test acceleration ratio comparison

由实验结果可以得出,跨基本块 SLP 向量化方法在 PolyBench 测试集中可以取得平均 3% 的性能提升。对比分析 SPEC 2006 测试程序和 PolyBench 测试程序,SPEC 2006 中的测试程序多用于全面评估计算机系统的性能,特别是在通用计算环境中的表现,其测试程序大多也是模拟真实世界的应用程序;而 PolyBench 用于评估编译器和硬件架构在多面体循环优化方面的性能,特别是在并行计算和资源利用率方面的表现,其测试程序大多是对具体算法进行测试。因此对于 SPEC 2006,该方法可以发掘出真实应用程序中传统 SLP 可能遗漏的非算法核中的可向量化机会;而对于 PolyBench,原有的向量化算法已经很好地对其进行优化,因此该方法很难对其进一步优化。

结束语 本文针对传统的 SLP 框架在发掘跨基本块的语句向量化潜力时存在能力不足的问题,提出了一种基于区域划分的跨基本块 SLP 向量化方法。该方法通过将处于支配关系下的多个基本块整合成一个统一的 SLP 区域,并以这个区域作为基本单元来进行 SLP 向量化处理,能够挖掘程序中更多的超字级并行性。本文方法基于 GCC 10.3.0 中的 SLP 向量化模块实现,并在 SPEC CPU2006 测试集和 polybench 测试程序中进行了验证。实验结果显示,在代码中存在大量适合 SLP 向量化的指令时,与传统局限于单个基本块内部的 SLP 方法相比,本文方法能够实现跨基本块的代码片段的向量化,可为 SLP 向量化范围扩展提供研究思路。

当前的研究中,对于嵌套循环中的区域向量化,通过简单的分层进行收益评估来判断是否进行向量化。但该方法在实践中存在内层收益评估不通过而导致整个区域无法向量化的情况。在未来的工作中,将探索更合理的区域划分方法,对于循环区域,将探索更合理的收益评估方法来加强跨基本块的 SLP 向量化能力。

参 考 文 献

[1] GAO W,ZHAO R C,HAN L,et al. Research on SIMD auto-vectorization compiling optimization[J]. Ruan Jian Xue Bao/Journal of Software,2015,26(6):1265-1284.

[2] FENG J G,HE Y P,TAO Q M. Auto-vectorization:Recent development and prospect[J]. Journal on Communications,2022,43(3):180-119.

[3] LIU H H,HAN L,CUI P F. Insufficient SLP in GCC[J]. Computer Systems & Applications,2022,31(9):265-271.

[4] VENKATESAN A,BANERJEE K,BHATTACHARJEE A,et al. Deep learning inference on ARM:A survey of compute libraries and quantization techniques[J]. ACM Transactions on Embedded Computing Systems,2020,19(1).

[5] HAN S,MAO H,DALLY W J. Neural network acceleration with efficient floating-point SIMD on FPGAs[C]∥2016 IEEE International Solid-State Circuits Conference. IEEE,2016:122-123.

[6] NVIDIA Corporation. Tensor Cores enable high-performance FP16 inference on NVIDIA Volta GPUs[EB/OL]. https://www. nvidia. com/content/dam/en-zz/Solutions/Data-Center/tensor-core-whitepaper. pdf.

[7] AMIRI H,SHAHBAHRAMI A. SIMD programming using Intel vector extensions[J]. Journal of Parallel and Distributed Computing,2020,135:83-100.

[8] STOJANOV A,TOSKOV I,ROMPF T,et al. SIMD intrinsics on managed language runtimes[C]∥Proceedings of the 2018 International Symposium on Code Generation and Optimization. 2018:2-15.

[9] LI J N,HAN L,CHAI G D. Automatic Vectorization Transplant and Optimization of LLVM for Domestic Processors[J]. Computer Engineering,2022,48(1):142-148.

[10] 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. 2008.

[11] HE T. An overview of compilation and optimization of automatic vector quantization based on data leve[J]. Intelligent Computer and Application,2016,6(6):68-71.

[12] LARSEN S,AMARASINGHE S. Exploiting superword level parallelism with multimedia instruction sets[J]. Programming Language Design and Implementation,2000,35(5):145-156.

[13] ZHAO J,ZHAO R C. Identifying superword level parallelism with directed graph reachability[J]. Scientia Sinica(Informationis),2017,47:310-325.

[14] PORPODAS V,MAGNI A,JONES T M. PSLP:Padded SLP automatic vectorization[C]// Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. 2015:190-201.

[15] FENG J,HE Y,TAO Q,et al. An SLP Vectorization Method Based on Equivalent Extended Transformation[J]. Wireless Communications and Mobile Computing, 2022, 2022 (1): 1832522.

[16] FENG J G,HE Y P,TAO Q M,et al. SLP Vectorization Method Based on Multiple Isomorphic Transformations[J]. Journal of Computer Research and Development,2023,60(12):2907-2927.

[17] ZHANG S P,WANG D,DING L L,et al. New framework based on SLP[J]. Application Research of Computers,2017,34(1):21-26.

[18] LI Y Y,XI H X,GAO W,et al. SLP vectorization method based on throttling[J]. Application Research of Computer,2018,35(9):2578-2582.

[19] XU J L,ZHAO R C,HAN L,et al. SIMD Code Selection Method for Inter-Basic-Block[J]. Journal of Information Engineering University,2016,17(2):244-249.

[20] CHEN Y S,MENDIS C,AMARASINGHE S. All You Need Is Superword-Level Parallelism:Systematic Control-Flow Vectorization with SLP[C]// Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation(PLDI '22). New York:ACM,2022:301-315.

[21] YE Z,JIAO J. Loop Unrolling Based on SLP and Register Pressure Awareness[C]// 2024 20th International Conference on Natural Computation,Fuzzy Systems and Knowledge Discovery (ICNC-FSKD). 2024:1-6.

[22] LI J,GAO W,LI Y,et al. An Improved Method for Control Dependency in LLVM[C]// 2024 5th International Conference on Intelligent Computing and Human-Computer Interaction(ICHCI). 2024:291-294.

[23] CHEN M Y,NEI K,LI J N,et al. An SLP automatic vectorization method,apparatus and electronic device:CN202311666914.7 [P]. 2024-03-05.

[24] TAYEB H,PAILLAT L,BRAMAS B. Autovesk:Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations[J]. ACM Transactions on Architecture and Code Optimization,2023,21(1):1-25.

HAN Lin,born in 1978,professor,doctoral supervisor,is a member of CCF (No. 16416M). His main research interests include high-performance computing,advanced compilation,program optimization and home-grown autonomous control.

CUI Pingfei,born in 1975,associate professor,master supervisor. His main research interests include domestic sovereign control,software reverse engineering and code security analysis.

(责任编辑:柯颖)