朱珂,何先波,滕芊芊. 基于 RISC-V 架构的寄存器分配研究[J]. 智能计算机与应用,2025,15(5):61-67. DOI:10.20169/j.issn.2095-2163.24111701

基于 RISC-V 架构的寄存器分配研究

朱 珂, 何先波, 滕芊芊
(西华师范大学 电子信息工程学院, 四川 南充 637009)

摘 要: 本文设计了一种针对 RISC-V 架构特性的寄存器分配方法。该方法基于寄存器分配与指令调度优化间的信息交互,结合历史分配策略来调整寄存器分配的结果,旨在减少寄存器分配后的指令依赖,从而更利于后续的指令调度优化,减少指令延迟,提高指令的执行效率。在 RISC-V 架构下,由于指令集精简,所有计算操作都依赖于寄存器,因此寄存器分配对性能影响显著,合理的分配策略则能充分利用硬件优势。本文方法的核心在于建立了寄存器分配优化与指令调度优化的反馈编译框架,并且通过全局的分配记录指导寄存器分配,实现更具适应性的寄存器分配优化。本方法不仅优化了寄存器分配,减少了对内存访问的需求,还提高了指令执行的连续性。本文基于 RISC-V 架构对 SPECCPU2006 基准测试集进行了测试,在整数基准程序上取得了平均 1.4% 的性能提升,相比于传统 LLVM 寄存器分配方法有明显改进。本文方法不仅有效提升了寄存器利用效率,还进一步验证了寄存器全局优化策略在提升 RISC-V 架构性能方面的潜力,为未来 RISC-V 编译器优化提供了有益的参考。

关键词: 寄存器分配; 编译器; 指令调度; 线性扫描算法
中图分类号: TP332.11 **文献标志码**: A **文章编号**: 2095-2163(2025)05-0061-07

Research on register allocation based on RISC-V architecture

ZHU Ke, HE Xianbo, TENG Qianqian
(School of Electronic and Information Engineering, China West Normal University, Nanchong 637009, Sichuan, China)

Abstract: This paper designs a register allocation method tailored to the characteristics of the RISC-V architecture. The method is based on the information interaction between register allocation and instruction scheduling optimization, combined with historical allocation strategies to adjust the register allocation results. The aim is to reduce instruction dependencies after register allocation, thereby facilitating subsequent instruction scheduling optimization, reducing instruction latency, and improving execution efficiency. In the RISC-V architecture, due to the simplified instruction set, all computational operations depend on registers, register allocation has a significant impact on performance. A reasonable allocation strategy can fully leverage hardware advantages. The core of the method in this paper is the establishment of a feedback compilation framework between register allocation optimization and instruction scheduling optimization. By guiding register allocation through global allocation records, the method achieves more adaptable register allocation optimization. This approach not only optimizes register allocation and reduces memory access requirements, but also improves instruction execution continuity. Based on the RISC-V architecture, the SPECCPU2006 benchmark suite is tested, achieving an average 1.4% performance improvement on integer benchmark programs, demonstrating significant improvement over the traditional LLVM register allocation method. This method effectively improves register utilization, and further validates the potential of global register optimization strategies in enhancing the performance of the RISC-V architecture, providing valuable reference for future RISC-V compiler optimizations.

Key words: register allocation; compiler; instruction scheduling; linear scan algorithm

0 引 言

在科技进步的宏观视野下,芯片技术占据了战略性、基础性和先导性的地位,关乎到公民信息安全乃至国防战略安全[1]。随着 RISC-V 架构兴起,尽管 RISC-V 架构展现出巨大的发展潜力,但由于发展时间不长,其编译器和软件开发环境未臻完善,所以认可度仍然有待提升[2]。开发环境的完备性、尤

作者简介: 朱 珂(1998—),男,硕士研究生,主要研究方向:寄存器分配; 滕芊芊(2000—),女,硕士研究生,主要研究方向:计算机视觉。
通信作者: 何先波(1971—),男,博士,教授,主要研究方向:嵌入式系统。Email:1946034057@qq.com。

收稿日期: 2024-11-17

其是编译器的成熟程度,对于 RISC-V 架构的用户基础及应用领域的拓展具有决定性影响。为了代码编译过程中硬件资源利用的最优化,寄存器分配成为了编译器优化的关键问题[3]。

当前,图着色寄存器分配算法和线性扫描寄存器分配算法[4]是主流寄存器分配算法。其中,图着色算法将寄存器分配问题转换为图着色问题[5],但是图着色问题是 NP 完全问题[6],即便是在 SSA(Static Single Assignment)形式[7]下,也仍需要多项式时间来求解[8],这无法满足现代即时(JIT)编译器快速高效的要求。而线性扫描寄存器分配算法是一种贪婪、快速的寄存器分配算法,更适用于 JIT 编译器[9]。目前,应用最为广泛的线性扫描寄存器分配算法是 Greedy 算法,然而其在寄存器分配后产生的指令依赖问题解决上还存在一定的欠缺。这种指令依赖现象极大地限制了后续指令调度的灵活性,阻碍了系统性能的最大化发挥[10]。因此,如何有效减轻寄存器分配后产生的指令依赖,一直是本领域研究的核心挑战之一。

针对这一难题,本文提出了一种创新的解决方案,该方案着眼于全局寄存器分配状态的优化。通过合理调整寄存器的分配策略,减少因寄存器分配不当而引发的指令依赖。为了验证所提算法的有效性,本文在 SPECCPU2006 基准测试集上进行了实验验证。实验结果表明,本算法在缓解指令依赖、提升指令执行效率方面展现出了显著的能力,从而验证了本文所提方法的可行性和优越性。

1 相关理论及方法

1.1 寄存器分配

寄存器分配是编译器优化中的核心环节,同时也是一个被广泛研究的重要的编译器优化问题,其中涉及到将一组无界的变量分配给一组有限的物理寄存器[11],减少对内存的访问次数。由于访问内存比访问寄存器要慢得多,因此必须尽量减少访存代码的数量,提高指令执行效率,从而优化性能[12]。

1.2 RISC-V 架构特点

RISC-V 是一种基于精简指令集(RISC)理念设计的开源指令集架构(ISA),旨在提供一个高效、灵活的指令集,满足学术界和产业界的需求。RISC-V 的开源特性使得用户能够自由定制指令集,这为创新与差异化提供了极强的设计空间[1]。RISC-V 架构还具有简洁的指令集设计、定制的指令集扩展以及良好的生态系统,这些优势使得 RISC-V 架构成为芯片自主研发的重要选择之一。

1.3 寄存器分配与指令调度的关系

寄存器分配和指令调度是编译器优化中的 2 个关键环节,两者在程序的性能优化中密切相关,并相互影响[13]。寄存器分配的主要任务是将程序中的虚拟寄存器映射到有限的物理寄存器中,而指令调度则旨在重新排列指令的执行顺序,以提高处理器的利用效率。

编译器后端优化流程如图 1 所示,现在的主流编译器通常采用 2 次指令调度夹着 1 次寄存器分配。在寄存器分配前进行的指令调度,此时程序中为虚拟寄存器,程序可以不受限制地使用虚拟寄存器,可在更大程度上优化代码的并行度;在寄存器分配之后的指令调度,全部寄存器已经替换成真实的物理寄存器,这部分调度的目的就是为了调度寄存器分配后产生的溢出指令,再进一步完成这部分后端优化。

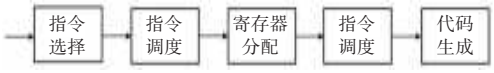

图 1 编译器后端优化流程

Fig. 1 Compiler backend optimization process

2 系统设计

本文的方法采用 LLVM 编译器的框架作为基本的编译框架。在后端优化中,建立了寄存器分配优化与指令调度优化的反馈优化框架如图 2 所示。进而,将这个框架分为了 Greedy 算法部分和与指令调度交互框架部分。其中,Greedy 算法部分为已有工作,与指令调度交互框架部分为本文研究工作,这里将展开研究论述如下。

2.1 已有工作

Greedy 算法是一种用于寄存器分配的启发式算法,广泛应用于编译器优化中。LLVM 编译器采用了 Greedy 算法用于寄存器分配。本文的虚拟寄存器选择阶段也是以 Greedy 算法作为基础,对其进行适用于 RISC-V 架构的改进,流程如图 3 所示。

2.1.1 生命周期分析与建立优先队列

虚拟寄存器的生命周期是指虚拟寄存器的创建、使用和销毁的过程[13]。当多个虚拟寄存器的生命周期不重叠时,分配器就可以将多个变量分配到同一个物理寄存器,提高物理寄存器的利用率[14]。研究时,是通过对虚拟寄存器中数据定义的基本块和数据使用的基本块来确定其生命周期。

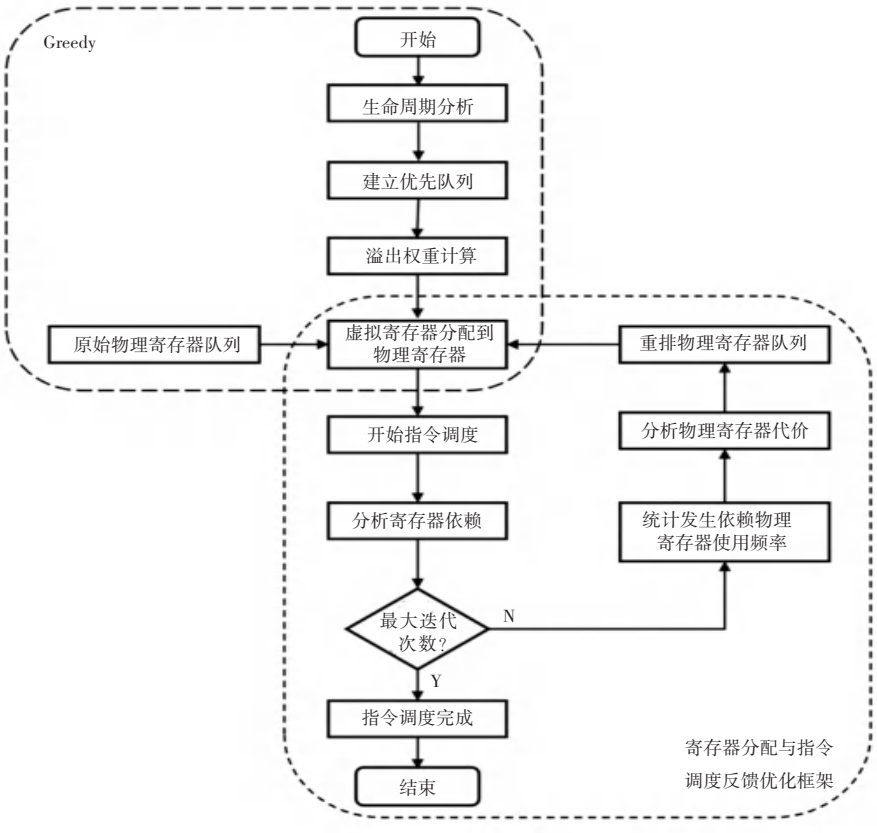

图 2　系统设计图

Fig. 2　Diagram of system design

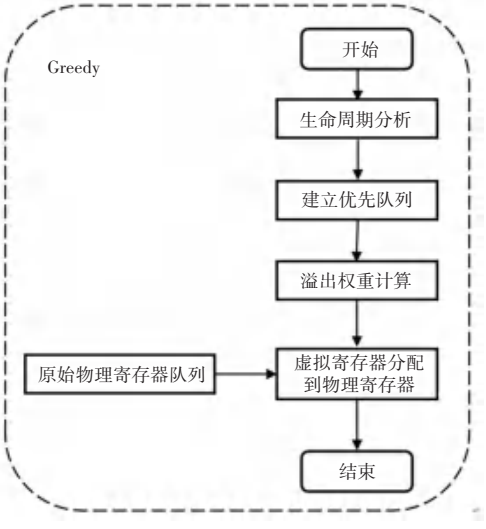

图 3　Greedy 算法流程图

Fig. 3　Greedy algorithm flowchart

假设一个函数的生存周期分析结果如图 4 所示。图 4 中，横轴为生存周期的 slot 节点，纵轴为虚拟寄存器变量和物理寄存器名称，分析中是通过 slot 节点的长度来表示寄存器%X、%Y、%Z 的生命周期，而 $1 和 $2 表示将要分配的物理寄存器。从图 4 中就可以得出，%X 与%Y 与%Z 在 slot 中都有重合的部分，所以这 3 个虚拟寄存器是不能放在同一个物理寄存器中。

同时，生命周期的长度也会影响到优先队列的建立。优先队列确保了在资源紧张时，使用重要性更高的虚拟寄存器能够优先获得物理寄存器。生命周期长的优先级更高，相应的分配顺序会更靠前，全局变量的优先级更高，而局部变量的优先级更低，物理寄存器分配时，选择优先级更高的虚拟寄存器分配将会有更高的执行效率[15]。

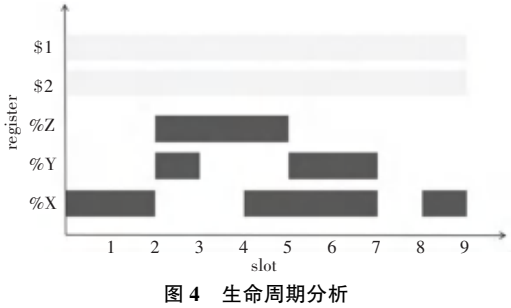

图 4　生命周期分析

Fig. 4　Live interval analysis

2.1.2　溢出权重计算

溢出权重表示每个虚拟寄存器的重要程度，因

为物理寄存器是体系架构中的核心组成,尤其是在复杂程序中,物理寄存器通常是少于变量的,因此必须高效地将重要的虚拟寄存器进行分配,以确保程序尽快运行[16]。当物理寄存器不够存下所有的虚拟寄存器时,就需要根据溢出权重来把重要的值存放在物理寄存器中,否则将会溢出到栈上,从而减少从栈上存取值这个非常耗时的行为[17]。溢出权重是通过虚拟寄存器的使用频率与生命周期长度的比值来计算,影响因素为基本块执行频率、指令所处位置和生命周期长度,具体计算方法如下:

$$Weight = \sum ((isDef + isUse) \times \frac{Block\ Frequency}{Entry\ Frequency} \times Loop\ Factor \times Remat\ Factor) / size\ of\ LiveInterval \quad (1)$$

其中, *isDef* 与 *isUse* 分别表示该虚拟寄存器是否被定义和使用; *Block Frequency* 和 *Entry Frequency* 分别表示 LLVM 编译器对虚拟寄存器对应的基本块执行频率和函数执行频率的预测值; *Loop Factor* 取决于基本块位置是否处于循环中,是、则为 3.2,否则为 1; *Remat Factor* 取决于虚拟寄存器的值是否可立即生成,是、则为 0.5,否则为 1。

权重 *Weight* 的计算公式,其目的就是用使用密度来衡量虚拟寄存器的重要性,尽可能让使用密度高的虚拟寄存器能够分配到物理寄存器中,而密度的计算就是使用频率与生命周期长度的比值。但是一些特殊的情况也需要增加权重,例如在循环中的变量会被频繁使用,这类变量尽量不要被溢出到内存中,否则溢出代价较大会导致性能表现变差。另外,如果一个寄存器的值可以被快速计算出来,那也无需将其一直保存在寄存器中,所以会将其对应的权重减半。

2.1.3 物理寄存器分配

算法依次从优先队列中取出虚拟寄存器,并尝试为其分配物理寄存器。在这一过程中,算法会检查当前寄存器是否有可用的物理寄存器,若有,则直接分配;若无,则比较已分配与待分配的虚拟寄存器溢出权重,低优先级的虚拟寄存器将会被取出来,考虑将其做生命周期分割、再进行重新分配,或者溢出处理,也就是存储到栈上。物理寄存器分配原理如图 5 所示。

从图 5 看到,假设权重%Z>%Y>%X,按照建立的优先队列(%X,%Y,%Z),依次进行物理寄存器分配,首先把 $1 分配给%X,接下来把 $2 分配给%Y,当对%Z 进行寄存器分配时,系统已经没有空闲的物理寄存器了,并且由于%Z 与%X、%Y 存在生命周期冲突,不能共享一个物理寄存器,此时就对比权重决定谁更重要,由于%X 权重小于%Z,将%X 从 $1 中删除,将 $1 重新分配给%Z。之后再对%X 进行分割生命周期后的重新分配或者进行后续的溢出处理。

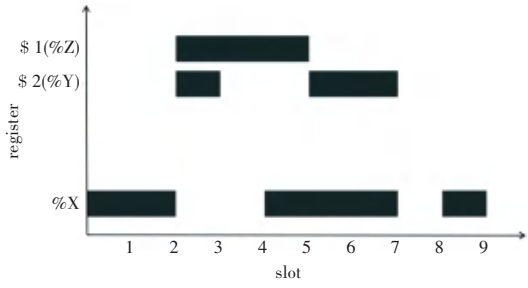

图 5 物理寄存器分配

Fig. 5 Physical register allocation

2.2 核心设计

本小节阐述了论文的核心工作,本阶段工作称为寄存器分配与指令调度交互框架。这部分研究是基于 LLVM 的编译框架,整体框架内容如图 6 所示。

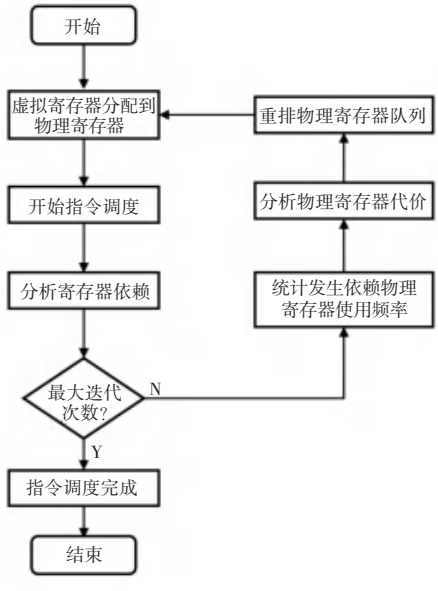

图 6 寄存器分配与指令调度交互框架

Fig. 6 Register allocation and instruction scheduling interaction framework

2.2.1 建立寄存器分配与指令调度反馈优化框架

(1)建立反馈优化框架。建立反馈优化框架的目的是,在寄存器分配与指令调度迭代过程中,接收到在指令调度中发生的因寄存器产生反依赖的位置信息,再携带此信息回到寄存器分配优化中,指导寄存器分配进行重分配。

在寄存器分配优化过程中,首次进入框架时采

用 Greedy 的原始物理寄存器队列参与分配。分配完成后，进入指令调度阶段。在指令调度过程中，通过数据依赖分析识别寄存器分配而产生的指令依赖，并对产生依赖的位置进行记录，再回到寄存器分配优化阶段，采用根据代价重排的物理寄存器队列对产生依赖的虚拟寄存器进行重分配。完成重分配后，再次进入指令调度优化阶段。该过程不断循环，直到不再有因寄存器而产生的指令依赖或达到设定的最大迭代次数，才会结束指令调度优化进程。此后再进入编译器后续的优化流程，最终完成整个程序的编译优化。

(2)建立过程。研究基于 LLVM 编译器的编译框架，模仿 LLVM 的 pipline，新建立了一个名为 RegAllocInstrSchedInteraction 的优化 Pass ，处于寄存器分配与指令调度之间。

当进行寄存器分配后，进入优化 Pass，指令调度中记录反依赖基本块位置的流程如图 7 所示。在 Pass 中，首先正常开始指令调度，指令调度以基本块为单位，在对基本块进行依赖分析后、进行调度前，检测最佳调度位置与当前调度位置是否不同，若是不同，则判断其发生了反依赖，就要记录下此基本块位置，然后进行下一个基本块的调度。直到调度完成，携带记录的基本块位置信息重新进入寄存器分配，对记录位置的基本块进行重分配。

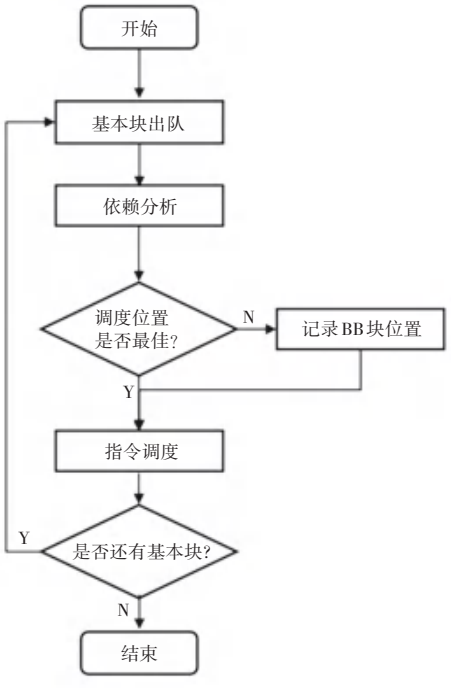

图 7　依赖分析流程图

Fig. 7　Dependency analysis flowchart

若是迭代过程已经找不到反依赖信息或者已经达到迭代最大次数，就结束本 Pass 的执行，进入指令调度优化，完成后续的编译工作。

2.2.2　通过重排物理寄存器队列顺序解决反依赖问题

这一步的目的是解决因寄存器分配产生的指令依赖问题。在寄存器分配执行过程中，按照原始的物理寄存器队列依次出队选择物理寄存器，无法避免出现指令依赖的情况。

(1)问题描述。当汇编要实现 2 条指令，第一条指令的源操作数用过物理寄存器后，该物理寄存器处于空闲状态，此时若寄存器分配算法将其分配给下一条指令的操作数，就会因这个物理寄存器造成指令上的依赖。在此基础上的指令调度，就无法将第二条指令调度到第一条指令之前，指令调度效果受到限制。在 RISC-V 架构上，所有的运算操作都依赖于寄存器，所以这种指令依赖的影响在 RISC-V 架构上尤为明显。

(2)解决方案。针对目前算法所存在的局限性，并充分发挥 RISC-V 架构的优势，在物理寄存器选择阶段，本文重新设计了物理寄存器队列的排列方式，根据使用代价重排物理寄存器队列。代价的具体计算如下：

$$Cost = Category + \frac{Use\ Frequency - \min(Use\ Frequency)}{\max(Use\ Frequency) - \min(Use\ Frequency)} \tag{2}$$

代价 *Cost* 的影响因素包括物理寄存器类型与近期物理寄存器的使用频率。

定义 *Category* 是为了区分物理寄存器、以及旨在提高分配的效率和避免颠簸的现象，不同类型的寄存器尽可能用于各自对应的方面，同时也考虑到不同类型寄存器之间的可通用性[18]。在 RISC-V 架构中，物理寄存器分为参数寄存器、保存寄存器、临时寄存器、返回寄存器等等。

参数寄存器、临时寄存器和保存寄存器都可以用作常规计算，参数寄存器和临时寄存器在函数调用前后不保留值，所以在寄存器分配时优先使用[19]。保存寄存器需要确保在函数调用前后的值保持不变，所以就需要保存现场和恢复现场，这个操作也就是保存到栈上，提前分配会造成更多的溢出，所以分配优先级较低。返回寄存器则是为了存储函数返回地址，分配优先级更低。在分配各自对应的寄存器时，*Category* 的值来自于根据分配优先级对应的常数，对应的值见表 1。

表 1 物理寄存器及对应类型

Table 1 Physical registers and the corresponding category

寄存器编号	物理寄存器	寄存器类型	category
51 ~ 58	*a*0 ~ *a*7	参数寄存器	1
46 ~ 48	*t*0 ~ *t*2	临时寄存器	2
69 ~ 72	*t*3 ~ *t*6	临时寄存器	3
49 ~ 50	*s*0 ~ *s*1	保存寄存器	4
59 ~ 68	*s*2 ~ *s*11	保存寄存器	6
42	*ra*	返回寄存器	7

Use Frequency 则是获取了单位基本块内的分配记录,对物理寄存器的使用次数做出的统计。研究中通过归一化公式将所有的频率缩放到 0 和 1 之间,并且 *Use Frequency* 小的值在队列中更靠前。这个计算的目的是统计物理寄存器使用密度,目的就是为了在保证寄存器类型的优先级下,优先使用未使用或很少使用的寄存器,尽可能减少由寄存器产生的指令依赖关系。

对每个物理寄存器,以 *Category* 和 *Use Frequency* 的值参与运算,得出该物理寄存器在本次分配的代价(*Cost*),并以代价重排物理寄存器队列。物理寄存器选择阶段,从物理寄存器队列中选择代价最小的物理寄存器参与分配。

3 实验评估

3.1 实验平台

SPEC CPU2006 是一套计算机性能评估基准测试套件,专注于评估现代处理器和编译器的性能[20],受到业界广泛肯定,并将其作为实验对象。

LLVM 编译器中,有 4 种寄存器分配方法:Fast、Basic、Greedy 和 PBQP。其中,Fast 分配方法是局部性的,作用于各个基本块;Basic 和 Greedy 的核心算法都是线性扫描算法的变种,但是 Greedy 算法结合了贪心思想;PBQP 分配方法将寄存器分配映射为分区布尔二次规划问题[20]。此后将本文的方法与这 4 种寄存器分配方法进行了比较。

研究中采用编译器 LLVM17.0.1 进行编译,开启-O3 优化,加入额外编译选项"-ffast-math -march=rv64gc -flto -static",启用不同的寄存器分配方法,编译出 5 组 binary,分别在商用芯片玄铁 c910 中加以运行并记录实验结果。

3.2 实验结果

寄存器分配实验结果见表 2。在表 2 中,呈现了 Basic 获得的运行时间(s)以及其它寄存器分配方法相较于 Basic 在每个基准测试中获得的提升,其中正(负)数为表示更慢(快)。同时,为了避免波动带来的影响,对于每一个程序都是运行 3 次取中位数获得准确结果。

实验结果表明,本文的分配方法比 Basic 快了 1 489.28 s,性能提高了 8.5%;相比于处于第二的 Greedy 算法,本文的方法也快了 245.18 s,性能提高了 1.4%。总体而言,在 5 种寄存器分配方法中,本文的方法总用时最少,性能最好,展示了本文的分配方法在 RISC-V 架构上的优越性。

表 2 寄存器分配实验结果

Table 2 Register allocation experimental results

Benchmark	Basic	Fast	PBQP	Greedy	Ours
400.perlbench	1 503.6	793.0	73.7	5.4	-14.74
401.bzip2	1 804.3	1 452.2	-142.7	-156.5	-175.25
403.gcc	1 161.5	514.0	70.9	-11.8	-22.93
429.mcf	1 372.5	900.9	-10.9	5.5	-14.69
445.gobmk	1 317.5	921.5	167.7	-42.6	-76.18
456.hmmer	1 452.8	2 042.7	10.8	-33.7	-47.47
458.sjeng	2 379.9	1 057.7	-510.3	-591.3	-574.78
462.libquantum	1 032.1	772.3	-52.4	-65.6	-68.17
464.h264ref	2 545.0	888.8	-86.3	-299.5	-355.57
471.omnetpp	1 556.0	276.7	159.4	42.8	19.80
473.astar	1 434.8	455.5	133.9	-110.8	-145.70
483.xalancbmk	1 277.3	763.6	107.7	14.0	-13.60
合计	18 837.3	10 838.9	-78.5	-1 244.1	-1 489.28

4　结束语

针对 RISC-V 架构特性，本文基于 LLVM 编译器，设计了寄存器分配与指令调度交互的编译框架，通过全局的分配视野结合重建的寄存器 *Cost* 模型指导寄存器分配，从而减少寄存器分配后产生的依赖，更利于后续的指令调度优化，使程序展现出更高的性能。实验表明，本文提出的寄存器分配方法相比于经典的 Basic、Fast、Greedy、PBQP 有着更好的性能效果。在未来的工作中，将会更注重寄存器分配优化与其他优化的联动，用优化间的交互信息指导寄存器分配，以获得更精确的优化指导信息，从而得到更好的性能。

参考文献

[1] 谢依伦. 基于 LLVM 的 RISC-V 代码密度优化算法[D]. 长沙：湖南大学，2021.

[2] 魏忠军. RISC-V 精简指令架构的研究及基于 RV32I 指令集的处理器内核设计[D]. 成都：电子科技大学，2021.

[3] 唐镇，胡勇华，陆浩松，等. 基于弱约束指派的 DSP 寄存器偶对分配算法研究[J]. 计算机科学，2021，48(S1)：587-595.

[4] ZHANG Xin，GU Qing，CHEN Xiang ，et al. A study of relative redundancy in test-suite reduction while retaining or improving fault-localization effectiveness[C]//Proceedings of the ACM Symposium on Applied Computing. New York：ACM，2010：2229-2236.

[5] CHAITIN G J. Register allocation & spilling via graph coloring[J]. ACM SIGPLAN Notices，1982，17(6)：98-101.

[6] CHAITIN G J，AUSLANDER M A，CHANDRA A K，et al. Register allocation via coloring[J]. Computer Languages，1981，6(1)：47-57.

[7] CYTRONR，FERRANTE J，ROSEN B K，et al. Efficiently computing static single assignment form and the control dependence graph[J]. ACM Transactions on Programming Languages & Systems，1991，13(4)：451-490.

[8] 岳峰，庞建民，赵荣彩. 一种基于分区域优先级的寄存器分配算法[J]. 电子与信息学报，2013，35(12)：3005-3010.

[9] 唐镇，胡勇华，陆浩松，等. 基于弱约束指派的 DSP 寄存器偶对分配算法研究[J]. 计算机科学，2021，48(S1)：587-595.

[10] 张鑫. 专用向量寄存器分配编译技术研究[D]. 湘潭：湖南科技大学，2023.

[11] VENKATA K S，JAIN S，KUNDU A，et al. Rl4real：Reinforcement learning for register allocation[C]//Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction. New York：ACM，2023：133-144.

[12] SHOBAKI G，SCOTT G V，MCHUGH P，et al. Register-pressure-aware instruction scheduling using ant colony optimization[J]. ACM Transactions on Architecture and Code Optimization，2022，19(2)：1-23.

[13] PANIGRAHI P，KARFA C. An investigation into the security of register allocation with spilling and splitting[C]//2023 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). Piscataway，NJ：IEEE，2023：1-6.

[14] DAS D，AHMAD S A，KUMAR V. Deep learning-based approximate graph-coloring algorithm for register allocation[C]//2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar). Piscataway，NJ：IEEE，2020：23-32.

[15] TERCI G S. A Learning-based coloring algorithm for register allocation problem[C]//2023 31st Signal Processing and Communications Applications Conference (SIU). Piscataway，NJ：IEEE，2023：1-4.

[16] CYTRON R，FERRANTE J，ROSEN B K，et al. Efficiently computing static single assignment form and the control dependence graph[J]. ACM Transactions on Programming Languages & Systems，1991，13(4)：451-490.

[17] 赵兴耀，单雅英. 代码生成中的寄存器分配及优化[J]. 计算机工程，1983(1)：48-56.

[18] CHOW F，HENNESSY J. Register allocation by priority-based coloring[J]. ACM SIGPLAN Notices，2004，39(4)：91-103.

[19] PHANSALKAR A，JOSHI A，JOHN L K. Subsetting the SPEC CPU2006 benchmark suite[J]. ACM SIGARCH Computer Architecture News，2007，35(1)：69-76.

[20] KIM M，PARK J K，MOON S M. Irregular register allocation for translation of test-pattern programs[J]. ACM Transactions on Architecture and Code Optimization (TACO)，2020，18(1)：5.