计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.200600061

基于弱约束指派的 DSP 寄存器偶对分配算法研究

唐 镇 胡勇华 陆浩松 王书盈
湖南科技大学 湖南 湘潭 411201
(tangzhenchina@163.com)

摘 要 在现代高性能数字信号处理器(DSP)中,许多指令把寄存器偶对作为操作数。为了优化寄存器偶对的使用,文中针对寄存器偶对的使用约束条件,提出了一种基于弱约束指派的 DSP 寄存器偶对分配算法。该算法在寄存器指派过程中优先指派空闲寄存器偶对给符号寄存器对。如果无法指派寄存器偶对给符号寄存器对,则指派两个不能组成寄存器偶对的寄存器。为了确保目标代码中寄存器偶对操作数最终获得的寄存器偶对符合寄存器偶对的使用约束条件,提供了一种指令操作数修正方法。采用 6 种经典的算法作为测试用例进行实验,结果表明所提算法的实验效果较好。

关键词:DSP;编译优化;全局寄存器分配;图着色方法;寄存器偶对
中图法分类号 TP311

Research on DSP Register Pairs Allocation Algorithm with Weak Assigning Constraints

TANG Zhen,HU Yong-hua,LU Hao-song and WANG Shu-ying
School of Computer Science and Engineering,Hunan University of Science and Technology,Xiangtan,Hunan 411201,China

Abstract In modern high performance digital signal processors (DSP),many instructions regard register pairs as operands. To optimize register pair usage,this paper presents a register pairs allocation algorithm for DSP based on weak constraint assignment for the rules of using register pairs. In the process of register assignment,the priority of this algorithm is to assign idle register pairs to symbol register pairs. If it is not possible to assign register pairs to symbol register pairs,two registers that cannot be made up of a register pair are assigned. In order to ensure that the register pairs in the target code are consistent with the rules of register pairs,this paper provides an instruction operand correction method. This paper uses six classical algorithms as test cases. The experimental results show that the proposed algorithm is effective.

Keywords DSP,Compiler optimization,Global register allocation,Graph coloring method,Register pairs

1 引言

随着处理器特别是数字信号处理器(DSP)的发展,硬件资源越来越丰富,硬件资源利用问题也随之而来[1]。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需要优化的关键问题[2]。寄存器分配的基本任务是将预分配代码中的变量映射到目标机器的寄存器或内存中[3]。线性扫描寄存器分配算法[4]和图着色寄存器分配算法是两种主流寄存器分配算法,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,适用于 JIT 编译器[5-6]。图着色寄存器分配算法是基于对变量生命期的详细刻画而提出的寄存器分配算法,能够生成质量较高的目标代码。

20 世纪 80 年代,Chaitin 等[7-8]设计了早期的图着色寄存器分配算法,并且将该算法应用到 PL.8 编译器中。基于图着色的寄存器分配算法通过分析预分配代码中变量的冲突关系来构建冲突图,并对冲突图进行着色,进而实现寄存器的分配。虽然 Chaitin 等提出的寄存器分配算法能够提供较好的寄存器分配方案,但是仍然存在两个不足之处。一个是 Chaitin 寄存器分配算法激进的变量合并策略,导致冲突图中顶点的度增大,从而导致寄存器压力增大;另一个是 Chaitin 寄存器分配算法存在变量过渡溢出问题。为了解决 Chaitin 寄存器分配算法中存在的问题,Briggs 等[9]提出乐观图着色寄存器分配算法。针对第一个不足之处,乐观图着色寄存器分配算法引入了保守的变量合并方法。针对第二个不足之处,乐观图着色寄存器分配算法在修剪冲突图阶段不决定符号寄存器的溢出,而是把溢出符号寄存器的决定推迟到寄存器指派阶段。

DSP 为应用需求提供体系结构支持,例如寄存器偶对、双内存部件等,从而造成不规则的体系结构特征[10]。由于 DSP 不规则的体系结构特征,乐观图着色寄存器分配算法生成的目标代码无法充分发挥出目标体系结构的性能。文献[11-13]针对不规则的体系结构中寄存器偶对作为指令的操作数这一硬件特性,对乐观图着色寄存器分配算法进行了算法改进。在改进的寄存器分配算法中,把成对的两个符号寄存器作为一个整体,并且在寄存器指派时为其指派一个符合体系结构要求的物理寄存器偶对。Nickerson 等[11]提出了含

基金项目:湖南省自然科学基金(2017JJ3087);国家自然科学基金资(61308001,61872138)
This work was supported by the Hunan Provincial Natural Science Foundation of China(2017JJ3087) and National Natural Science Foundation of China(61308001,61872138).
通信作者:胡勇华(huyh@hnust.cn)

有寄存器组的寄存器分配算法，其中寄存器组由多个编号连续且对齐寄存器构成，该算法可用于任意长度的寄存器组的寄存器分配。乐观图着色寄存器分配算法无法通过冲突图中结点的度准确地判定寄存器偶对结点的可着色性。Briggs等[12]通过向冲突图中添加边的方法解决了一问题。Smith等[13]提出了权重冲突图着色的方法，给冲突图中的每个结点赋予与之寄存器类对应的权重，把寄存器偶对作为一个新的寄存器类别，通过结点权重判断结点的可着色性。当冲突图中只有寄存器和寄存器偶对结点时，Briggs提出的寄存器分配算法和Smith提出的寄存器分配算法生成的寄存器分配方案相同。

上述对寄存器偶对分配相关的算法进行了阐述，尽管对寄存器偶对分配进行了大量的研究和探索，但是现存的寄存器偶对分配算法还是不能合理地使用寄存器偶对。本文针对寄存器偶对的使用约束条件，在图着色全局寄存器分配算法的基础上提出了一种改进的寄存器偶对分配优化算法，优化了寄存器偶对的使用。该算法解除了在进行寄存器指派时必须给符号寄存器对指派寄存器偶对的这一要求，并配以指令寄存器偶对操作数修正算法作为补充，以保证操作数中的寄存器偶对最终符合使用约束条件。该优化算法提升了寄存器指派过程中寄存器使用的灵活性，减少了由于物理寄存器偶对使用约束条件导致的符号寄存器对的溢出，有效地提高了寄存器指派过程的成功率。

本文第2节对Briggs提出的寄存器偶对分配算法进行了分析；第3节对本文提出的寄存器偶对分配算法进行了介绍；第4节给出了本文算法的示例分析；第5节进行了实验分析；最后总结全文。

2 问题分析

许多先进体系结构[14-17]支持寄存器偶对的使用。寄存器偶对常常用于存储或处理更大字长的数据，或用一个寄存器偶对来存储超过单个寄存器表示范围的存储器地址。通常情况下，大多数体系结构对寄存器偶对的使用设置了两条约束条件，即寄存器偶对中两个寄存器的编号相邻；寄存器偶对的第一个寄存器的编号必须是奇数，例如物理寄存器 R_1 和 R_0 组成的寄存器偶对$[R_1,R_0]$。本文称寄存器偶对的左半部分为寄存器偶对的左位，称寄存器偶对的右半部分为寄存器偶对的右位。在寄存器偶对$[R_1,R_0]$中，R_1 为$[R_1,R_0]$的左位，R_0 为$[R_1,R_0]$的右位。

图着色全局寄存器分配算法是解决DSP寄存器分配问题常用的方法。乐观图着色寄存器分配算法利用冲突图 $G=(V,E)$表示变量间的冲突关系。在冲突图 G 中，V 表示变量集合，E 表示预分配代码中各变量的活跃范围相交(即变量间的冲突关系)。通过对冲突图进行着色，来实现寄存器的分配。每一种颜色对应一个物理寄存器，如果两个变量是冲突图上的相邻顶点，则给两个变量着不同的颜色。针对寄存偶对的使用约束条件，Briggs改进了乐观图着色寄存器分配算法。改进算法把寄存器偶对操作数当成一个整体。在冲突图中使用一个结点表示寄存偶对操作数，并且使用两条边表示寄存偶对操作数与其冲突结点的冲突关系。以表1所列的代码片段为例，对Briggs寄存器分配算法进行分析，并且假设目标体系结构拥有5个寄存器。其中，寄存器编号为 R_0 至 R_4，寄存器 R_1 和 R_0 可作为寄存器偶对使用，同样寄存器 R_3 和 R_2 也可以作为寄存器偶对使用。根据代码片段1中的变量的冲突关系，构建了图1所示的冲突图。然后，按照变量出现的顺序修剪冲突图中邻接结点数小于寄存器总数 N 的结点，并将它们压入栈中(图2左端为栈底)。如果所有剩余结点的度数都大于等于 N(N 为目标体系结构中寄存器的数量)，则从中选择一个溢出代价最小的结点压入栈中。最后，依次弹出栈顶结点，并给它们分别指派一个未被其相邻结点使用的寄存器。

表1 代码片段1

Table 1 Code fragment 1

I_0:load $[a_1,a_0]$,address;
I_1:$b=[a_1,a_0]+[a_1,a_0]$;
I_2:$c=[a_1,a_0]*[a_1,a_0]$;
I_3:$d=[a_1,a_0]$;
I_4:$e=b+c$;
I_5:$f=c+d$;

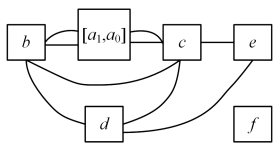

图1 代码片段1中变量的冲突图

Fig. 1 Variable conflict graph in code fragment 1

为结点$[a_0,a_1]$第一次指派物理寄存器时，冲突图中其结点的寄存器指派结果如图2所示。目标体系结构中存在5个寄存器，寄存器 R_0 和 R_2 已经指派给$[a_0,a_1]$的邻接结点，因此$[a_1,a_0]$的可用候选物理寄存器为 R_1 和 R_3。但是，由于寄存器偶对的使用约束条件，寄存器 R_1 和 R_3 无法组成寄存器偶对，因此$[a_1,a_0]$需要进行溢出处理。由此可看出，Briggs提出的寄存器分配算法存在过溢出的情况。在进行寄存器偶对指派时，如果候选物理寄存器集合中不存在满足使用约束条件的物理寄存器偶对，即使符号寄存器对存在至少两个候选物理寄存器，我们也需要对它们进行溢出处理。变量的溢出处理需要在代码中插入内存操作指令，进而导致生成目标代码的运行效率降低。

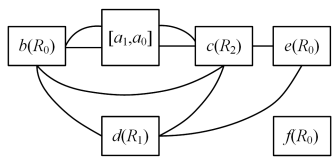

图2 其他结点寄存器的指派情况

Fig. 2 Assigned rigister of other node

除了存在上述不足之外，Briggs提出的寄存器分配算法无法直接处理一个寄存器偶对操作数的左位是另一个寄存器偶对操作数右位的情况。在表2所列的代码片段中，变量 g 是寄存器偶对操作数$[g,e]$的左位，同时也是寄存器偶对操作数$[f,g]$的右位。因此，Briggs提出的算法无法直接为 $[g,e]$和$[f,g]$分配物理寄存器偶对，需要事先引入中间变量才能进行寄存器分配，从而增大了被处理代码的复杂性，也使寄存器分配算法变得更加复杂。

表2 代码片段2

Table 2 Code fragment 2

I_7:$[h_0,h_1]=[g,e]*[g,e]$;
I_8:$[i_0,i_1]=[f,g]*[h_0,h_1]$;

3　基于弱约束指派的 DSP 寄存器偶对分配算法

3.1　算法的总体框架

本文以图着色全局寄存器分配方法为基础，提出了一种基于弱约束指派的 DSP 寄存器偶对分配算法。该算法将寄存器分配过程分成两个阶段，即弱约束寄存器分配阶段和寄存器偶对操作数修正阶段。为了提升寄存器指派过程中寄存器使用的灵活性，该算法优化了无约束寄存器分配阶段中寄存器偶对的指派过程。寄存器偶对操作数修正阶段对指令操作数中没有满足寄存器偶对使用的错误寄存器偶对进行修正处理，保证了最终生成的目标代码中各寄存器偶对的使用符合约束条件。

在无约束寄存器分配阶段，将寄存器偶对操作数的左位和右位分开进行寄存器分配。在符号寄存器分配过程中，分别给寄存器偶对操作数的左位、右位分配符号寄存器。分配给寄存器偶对操作数的两个符号寄存器在本文称为符号寄存器对。在寄存器指派过程中，将符号寄存器对中的两个符号寄存器与分配给普通操作数的符号寄存器一起进行寄存器指派，得到各自的物理寄存器。在寄存器指派过程中，优先为符号寄存器对指派两个能够构成寄存器偶对的寄存器。如果无法指派寄存器偶对，则指派两个不能组成寄存器偶对的寄存器。在溢出过程中，可能存在两个溢出的符号寄存器是一对符号寄存器对的情况。由于寄存器偶对操作数的左位和右位是分开进行寄存器分配的，因此传统的寄存器分配算法将分别对寄存器偶对操作数的左位和右位进行溢出处理。为减少溢出恢复指令的插入，本文将这两个符号寄存器作为一个整体进行溢出处理。

为了提升寄存器指派时寄存器使用的灵活性，无约束寄存器分配阶段中的寄存器指派过程没有严格遵守寄存器偶对的使用约束条件，可能导致寄存器分配结果中存在寄存器偶对的使用不符合约束条件的情况。为了确保指令操作数中寄存器偶对使用的正确性，我们在无约束寄存器分配后添加了一个指令操作数修正过程，该算法包含指令寄存器偶对源操作数修正和寄存器偶对目标操作数修正两部分。该算法依次获取各条指令进行寄存器偶对操作数修正处理，如果指令中存在至少一个以上的操作数涉及寄存器偶对，对指令中各操作数依次进行寄存器偶对合法性检查，并根据检查结果使用替换寄存器将寄存器偶对中不符合使用约束条件的使用错误的寄存器替换，所述替换寄存器根据本文的启发式方法进行选取。条件满足时优先使用同一指令中前面的操作数寄存器偶对的替换寄存器替换后面相同的所述错误使用寄存器。确定替换寄存器偶对之后，更新数据流信息和寄存器的空闲状态。然后，根据替换寄存器偶对和错误寄存器中值的活跃情况插入指令，确保修正处理过程中不会改变原来代码的语义。

3.2　寄存器偶对指派优化算法

本文结合候选物理寄存器的编号和寄存偶对的使用约束条件，对乐观图着色寄存器分配算法中的寄存器指派过程进行了优化。优化算法尽可能地指派符合寄存器偶对给符号寄存器对。当符号寄存器对存在多个候选物理寄存器但不存在寄存器偶对时，指派的物理寄存器无需符合物理寄存器偶对使用的约束条件。在某些情况下，符号寄存器对不能按照寄存器偶对指派方法进行寄存器指派，符号寄存器对中的两个单个符号寄存器需要按照普通单寄存器进行寄存器指派。例如，表 2 中寄存器偶对操作数[g,e]中的左位 g 是寄存器偶对操作数[f,g]中的右位，因此寄存器偶对操作数[g,e]和[f,g]不能使用本文寄存器偶对指派优化算法进行寄存器指派，需要把寄存器偶对操作数[g,e]和[f,g]按照普通单个符号寄存器 e,f 和 g 进行寄存器指派。

指派优化算法的伪代码如表 3 所列，AssignRegs 函数的参数 stack 是修剪冲突图后产生的结点栈，栈中的元素是冲突图中结点在 adjList 中的编号，adjList 是冲突图的邻接表表示。FindoutSymbolRegisterPair 函数获取代码中所有的符号寄存器对。symbolRegisterPair 是由两个单个符号寄存器组成的符号寄存器对。JudgeNodeType 函数用于判断 stack 中元素的类型。如果当前处理的栈元素为非符号寄存器对，则 symbolRegisterPairs 的 lift 和 right 为空。如果当前栈元素是某个符号寄存器对的右位，则 symbolRegisterPair 的左位为空，当前栈元素赋值给 symbolRegisterPair 中的 right 元素。如果当前处理元素是某个符号寄存器对的左位，则把该符号寄存器对的 right 和 left 元素对应地赋值给 symbolRegisterPair。GetCandRegPair 函数用于获取未指派给 symbolRegisterPair 相邻结点寄存器的编号。

表 3　寄存器指派优化算法伪代码

Table 3　Register assignment optimization algorithm pseudo-code

```
AssignRegs(stack)
int inode;
bool success=true;
symbolRegisterPairs=FindoutSymbolRegisterPair ();
while(stack !=Ø){
  inode=stack.pop();
  symbolRegisterPair=JudgeNodeType(inode,symbolRegisterPairs);
  if(symbolRegisterPair.odd !=Ø){
      candRegs=GetCandRegPair(symbolRegisterPair);
  }else{
    candRegs=GetCandRegs(inode);//获取结点 inode 的相邻的结点还未使用的颜色
  }
  if (symbolRegisterPair.left !=Ø){
        if(candRegs.size()>1){
        regGroup=GetRegGroup(candRegs);
        adjList[symbolRegisterPair.left].color=RegGroup.left;
        adjList[symbolRegisterPair.right].color=RegGroup.right;
      }else{
        adjList[symbolRegisterPair.left].spill=true;
        adjList[symbolRegisterPair.right].spill=true;
        success=false;
      }
  }else if(symbolRegisterPair.right !=Ø){
      continue;//跳过本次循环体中余下尚未执行的语句
  }else{
      adjList[inode].color=candRegs.head();
  }
  if(adjList[inode].color==Ø){
      adjList[inode].spill=true;
      success=false;
    }
}
return success
```

在为符号寄存器对选择指派物理寄存器的过程中，优先选择编号最大的寄存器偶对。如果集合 candRegs 中不存在寄存器偶对，则选择两个无法组成寄存器偶对的寄存器，即为符号寄存器对的左位选择编号为奇数的物理寄存器；为符号

寄存器对的右位选择编号为偶数的寄存器。如果集合 candRegs 中不存在奇数编号的寄存器，则为符号寄存器对的左位和右位都选择编号为偶数的寄存器。如果集合 candRegs 中不存在偶数编号的寄存器，则为符号寄存器对的右位和左位都选择编号为奇数的寄存器。GetRegGroup 函数实现了为符号寄存器对选择用于存储其值的物理寄存器的功能，其伪代码如表 4 所列。GetRegGroup 函数调用 GetRegsPair 函数从候选物理寄存器集合 candRegs 中为符号寄存器对选择寄存器偶对。如果集合 candRegs 中不存在合法寄存器偶对，则 GetRegsPair 返回空，否则返回寄存器偶对中的两个物理寄存器的编号。

表 4 为符号寄存器对选择物理寄存器

Table 4 Select physical register for symbol register pairs

```
GetRegGroup (candRegs);
bool haveGlbRegOddSN=false,haveGlbRegEvenSN=false;
regGroup=GetRegsPair(candRegs);
if(regGroup.left==Ø && regGroup.right==Ø){
  bool haveGlbRegOddSN=false,haveGlbRegEvenSN=false;
  for( curIt=candRegs.begin();curIt! =candRegs.end();++curIt){
      if(*curIt %2==1){
      regGroup.left=*curIt;
      haveGlbRegOddSN=true;
    }else{
      regGroup.right=*curIt;
      haveGlbRegEvenSN=true;
    }
  }
  if(!haveGlbRegOddSN ){
      candRegs.erase(regGroup.right);
      regGroup.left=*(candRegs.begin());
  }
  if(!haveGlbRegEvenSN ){
      candRegs.erase(regGroup.left);
      regGroup.right=*(candRegs.begin());
  }
}
return regGroup;
```

3.3 指令操作数修正算法

本文针对上述问题设计了寄存器偶对操作数处理算法。寄存器偶对操作数处理算法以寄存器分配算法的输出指令序列为算法的输入，通过寄存器偶对操作数处理算法修正指令中的寄存器偶对操作数，保证指令操作数中寄存器偶对使用的正确性。寄存器偶对修正算法以单条指令为基本处理对象，包含指令源操作数寄存器偶对修正、目标操作数寄存器偶对修正两部分。该算法结合数据流信息，分别对指令中的源操作数和目标操作数寄存器分配的结果按照不同的模式进行寄存器偶对修正。指令操作数修正算法使得指令操作数中寄存器偶对的使用符合寄存器偶对使用的约束条件和源代码的语义。

首先，分析寄存器偶对操作数修正处理模式。然后，利用启发式方法选择的合法寄存器偶对，以替换指令中的错误寄存器偶对。最后，根据错误寄存器偶对和替换寄存器偶对中值的活跃情况，在当前处理指令的前后插入指令。通过在当前处理指令前插入数据传送指令，把当前处理指令中错误寄存器偶对中的数据暂存到通过启发式方法获得的合法寄存器偶对中。在当前处理指令后添加数据传送指令，将暂存的数据传送到原来的寄存器中。寄存器偶对操作数修正算法使用替换寄存器偶对替换错误寄存器偶对，保证单条指令中寄存器偶对操作数的正确性。通过在当前处理指令前后插入指令，来确保修正处理过程不发生语义变化。

确定当前处理指令中各个寄存器偶对操作数的修正模式。如果指令中存在至少一个以上的操作数涉及寄存器偶对，则结合寄存器偶对使用约束条件来确定操作数中的寄存器偶对的处理模式。确定寄存器偶对的模式后，分别对指令源操作和目标操作数中的寄存器偶对进行修正处理。本文将寄存器分配阶段得到的寄存器分配结果中各条指令的寄存器偶对操作数分为 5 种模式(以指令 S“FADDD src0,src1,dst”为例介绍修正处理模式划分，指令 S 为 64 位浮点加法指令，并且指令的操作数为寄存器偶对)。

(1)Correct 模式，寄存器偶对操作数中的两个寄存器的编号满足寄存器偶对的使用约束条件。该模式的寄存器偶对操作数不需要进行修正处理。

(2)BothError 模式，寄存器偶对操作数中的两个寄存器的编号都不符合要求，即本来应该为奇/偶号的寄存器，但实际得到的是偶/奇号寄存器。如果指令 S 的源操作数 src0 为 $[R_2,R_9]$，则 src0 的处理模式为 BothError 模式。

(3)RightError 模式，寄存器偶对操作数中右位寄存器的编号不符合要求。如果指令 S 的操作数 src0 为 $[R_3,R_1]$，则 src0 的处理模式为 RightError。

(4)LeftError 模式，寄存器偶对操作数中左位寄存器的编号不符合要求。如果指令 S 的操作数 src0 为 $[R_2,R_4]$，则 src0 的处理模式为 LeftError。

(5)Illegal 模式，寄存器偶对操作数中两个寄存器都满足位置模式对奇偶编号的要求，但两者的编号不相连。如果指令 S 的操作数 src0 为 $[R_3,R_8]$，则 src0 的处理模式为 Illegal。

3.3.1 指令源操作数寄存器偶对修正

在指令源操作数寄存器偶对修正过程中，根据活跃变量分析结果，分析出指令入口处的各个物理寄存器空闲的状态信息，将体系结构中的寄存器偶对按照空闲状态分别放到空闲寄存器偶对、半空闲寄存器偶对、非空闲寄存器偶对 3 种集合中。指令入口指当前处理指令与前一条指令之间的程序点。空闲寄存器偶对中包含两个空闲物理寄存器；半空闲寄存器偶对中包含一个空闲寄存器，该空闲寄存器可能是半空闲寄存器偶对的左位也可能是右位；非空闲寄存器偶对的两个物理寄存器都是活跃状态。修正处理方法使用启发式方法在空闲寄存器偶对、半空闲寄存器偶对、非空闲寄存器偶对中挑选替换寄存器偶对。

指令源操作数寄存器偶对修正算法的伪代码如表 6—表 9 所列，伪代码中符号的定义如表 5 所列。指令源操作数寄存器偶对修正算法伪代码中 remove(RPset) 函数的功能为排除 RPset 中含源变量或 RepSet 集合内容的寄存器偶对。

处理 BothError 模式寄存器偶对源操作数，处理过程伪代码如表 6 所列。假设当前处理操作数为 $[R_x,R_y]$，而且当前处理源操作数中两个寄存器的使用不符合寄存器偶对的使用约束条件。为了让操作数中的寄存器的使用符合寄存器偶对的使用约束，因此需要对该操作数中的错误寄存器偶对选择替换寄存器偶对。如果在处理操作数 $[R_x,R_y]$ 之前，修正处理算法对指令 S_0 中的其他寄存器偶对操作数进行了修正处理，则修正处理过程中可能会产生 R_x 的替换寄存器集合

和 R_y 的替换寄存器集合。因此，优先考虑分别从 R_x 和 R_y 的替换集合中各选择一个寄存器，组成替换寄存器偶对 $[R_{2a+1},R_{2a}]$。如果无法组成寄存器偶对 $[R_{2a+1},R_{2a}]$，则选择一个半空闲寄存器偶对作为替换寄存器偶对。该半空闲寄存器偶对中的活跃寄存器中的值为 R_x 或 R_y 中的值。如果不存在这样的半空闲寄存器偶对，则选择一个空闲寄存器偶对作为替换寄存器偶对。如果此时不存在空闲寄存器偶对，则选择一对半空闲寄存器偶对作为替换寄存器，此时的半空闲寄存器偶对中的活跃寄存器的值与 R_x 和 R_y 中的值不同。如果这样的半空闲寄存器偶对不存在，则选择非空闲寄存器偶对作为替换寄存器偶对。

表 5　符号定义

Table 5　Symbol definition

符号	含义
$R=\{R_i \mid i\in N^+\}$	体系结构中物理寄存器集合，R_i 表示编号为 i 的物理寄存器
$[R_m,R_n]$	寄存器偶对，合法物理寄存器偶对为 $[R_{2n+1},R_{2n}]$，$n\in N^+$
SLI	指令入口处活跃变量集合
$A\rightarrow B$	在当前处理指令中，使用符号 A 替换符号 B
$M\Rightarrow N$	把 M 中的数据传送到 N 中的数据传送指令
SourceOperandSet	当前处理指令源操作数集合，该集合的元素为 R_i 或者 $[R_m,R_n]$
$\{R_j\cdots\}:R_k$	式中集合中的寄存器在不同的操作数中替换了寄存器 R_k，称为 R_k 的替换集
$RepSet=\{\{R_j\cdots\}:R_k,\cdots\}$	处理当指令产生的所有替换集集合。$RepSet_{Rk}=\{R_j\cdots\}:R_k$，符号“$RepSet_{Rk}$”中的下标为 R_k
beforeInsertInst	在当前处理指令前面插入该集合中的指令，该集合的元素为 $M\Rightarrow N$
BeProtSet	该集合的元素为单个物理寄存器 R_i，需要在当前处理指令之前插入指令防止 R_i 中的数据被改动，而且在当前处理指令后插入指令把原来数据恢复到 R_i

表 6　BothError 模式错误寄存器偶对修正处理过程

Table 6　Correct process for BothError mode error registerpairs

```
Bool CorrectBothErrorRegisterPairs([ Rx,Ry])
1. if((R2a+1∈RepSetRx)&&(∃R2a∈RepSetRy)){
2.     R2a+1→Rx;
3.     R2a→Ry;
4. }else if((∃R2b+1∈RepSetRx) && (∃[R2b+1,R2b]∈HalfIdleRpSet)
   && R2b+1∈SLI){
5.     R2b+1→Rx;
6.     operatorAfterSelectSubsitutePairs(R2b,Ry);
7. }else if((∃R2c∈RepSetRy) && (∃[R2c+1,R2c]∈HalfIdleRpSet)
   && R2∈SLI){
8.     R2c→Ry;
9.     operatorAfterSelectSubsitutePairs(R2c+1,Rx);
10. }else if(IdleRPSet !=Ø && (∃[R2d+1,R2d]∈IdleRPSet)){
11.    operatorAfterSelectSubsitutePairs(R2d+1,Rx);
12.    operatorAfterSelectSubsitutePairs(R2d,Ry);
13. }else if(HalfIdleRPSet !=Ø){
14.    SH=Remove(HalfIdleRPSet);//排除 SH 中含源变量或 RepSet 集合
       内容的寄存器偶对
15.    If(∃[R2e+1,R2e]∈SH){
16.       operatorAfterSelectedSubsitutePairs(Rx,R2e+1);
17.       operatorAfterSelectedSubsitutePairs(Ry,R2e);
18.    }
19. }else{
20.       SH=Remove( NonIdleRPSet );//排除 SH 中含源变量或 RepSet
          集合内容的寄存器偶对
21.       if(∃[R2f+1,R2f]∈SH ){
22.       operatorAfterSelectedSubsitutePairs(Rx,R2f+1);
23.       operatorAfterSelectedSubsitutePairs(Ry,R2f);
24.    }
25. }
26. return true;
```

处理 RightError 模式寄存器偶对源操作数，处理过程如表 7 所列。假设当前处理操作数为 $[R_x,R_y]$，并且该操作数处理模式为 RightError。也就是说，操作数 $[R_x,R_y]$ 中的 R_y 的使用不符合寄存器偶对的使用约束条件。为了减少插入指令的数量，处理过程中优先考虑使用寄存器偶对 $[R_x,R_{x-1}]$ 替换错误寄存器偶对。在处理当前操作数时，如果寄存器 R_{x-1} 已用于替换 R_y 之外的其他寄存器，则将存器偶对操作数 $[R_x,R_y]$ 的处理模式改为 BothError。如果指令 S 的其他源操作数使用了 R_{x-1}，同样也需要将寄存器偶对 $[R_x,R_{x-1}]$ 的处理模式改为 BothError。

表 7　RightError 模式错误寄存器偶对修正处理过程

Table 7　Correct process for RightError mode error registerpairs

```
bool CorrectRightRegisterPairs([Rx,Ry])
1. bool haveUsed=false;
2. foreach Ri∈R{
3.     if(Rx-1∈RepSetRi && (Ri!=Ry)) haveUsed=true;
4. }
5. if( haveUsed ){
6.     if(!CorrectBothErrorRegisterPairs([Rx,Ry])) Return false;
7. }
8. if(Rx-1∈RepSetRy){
9.     Rx-1=Ry;
10. }else{
11.    if(Rx-1∈SLI && Rx-1∈SourceOperandSet){
12.       if( ! CorrectBothErrorRegisterPairs([Rx,Ry]) ) Return false;
13.    }else{
14.       operatorAfterSelectedSubsitutePairs(Ry,Rx-1);
15.    }
16. }
17. return true;
```

处理 LeftError 模式寄存器偶对源操作数，处理过程如表 8 所列。假设当前处理操作数为 $[R_x,R_y]$，其处理模式为 LeftError。该模式操作数的左位 R_x 使用不满足寄存器偶对的使用条件。首先，考虑使用 R_{y+1} 替换 R_x。在处理操作数 $[R_2,R_4]$ 之前，如果 R_{y+1} 已用于替换 R_{y+1} 之外的其他寄存器，则操作数 $[R_x,R_y]$ 的处理模式改为 BothError。在处理当前指令的其他源操作数过程中，如果 R_{y+1} 已经替换 R_x，则直接将当前处理操作数中的 R_x 替换为 R_{y+1}，并且无需插入指令。如果当前处理指令的其他源操作数中存在 R_x 的使用，则将操作数 $[R_x,R_y]$ 的模式改为 BothError 模式，否则选择寄存器偶对 $[R_{y+1},R_y]$ 作为当前操作数的替换寄存器偶对。

处理 Illegal 模式寄存器偶对源操作数，处理过程如表 9 所列。假设当前处理操作数为 $[R_x,R_y]$。该处理模式的操作数左位是编号为奇数的寄存器偶对，右位是编号为偶数的寄存器。该类错误操作数是由于操作数的左位和右位使用的寄存器的编号不连续造成的。因此，可以假设操作数 $[R_x,R_y]$ 中的左位错误进行处理，或者假设操作数 $[R_x,R_y]$ 中的右位错误进行处理。优先考虑选择寄存器偶对 $[R_{y+1},R_y]$ 或 $[R_x,R_{x-1}]$ 作为错误寄存器偶对操作数 $[R_x,R_y]$ 的替换寄存器偶对。在处理当前处理指令的其他源操作数时，如果 R_{y+1} 已经用于替换 R_x，则选取 $[R_{y+1},R_y]$ 作为替换寄存器偶对。同样，在处理当前处理指令的其他源操作数时，如果 R_{x-1} 已经用于替换 R_y，则选取 $[R_x,R_{x-1}]$ 作为替换寄存器偶对。其次，考虑使用半空闲寄存器偶对 $[R_{y+1},R_y]$ 或 $[R_x,R_{x-1}]$ 替换 $[R_x,$

R_y]。半空闲寄存器偶对[R_{y+1},R_y]中 R_{y+1}为空闲寄存器,半空闲寄存器偶对[R_x,R_{x-1}]中 R_{x-1}为空闲寄存器。如果无法找到合适的半空闲寄存器偶对,则在非空闲寄存器偶对[R_{y+1},R_y]或[R_x,R_{x-1}]中选择一个寄存器偶对作为替换寄存器偶对。如果[R_{y+1},R_y]和[R_x,R_{x-1}]中的 R_{y+1},R_{x-1}是当前处理指令的源操作数,则将当前处理操作数的处理模式改为 BothError。

表 8 LeftError 模式错误寄存器偶对修正处理过程

Table 8 Correct process for LeftError mode error registerpairs

```
bool CorrectLeftErrorRegisterPairs([Rx,Ry])
1. bool haveUsed=false;
2. foreach Ri∈R{
3. if(Ry+1∈RepSetRi && (Ri!=Rx)) haveUsed=true;
4. }
5. if( haveUsed ){
6. if(! CorrectBothErrorRegisterPairs([Rx,Ry])) Return false;
7. }
8. if(Ry+1∈RepSetRx){
9. Ry+1→Rx;
10. }else{
11. if(Ry+1∈SLI && Ry+1∈sourceOperandSet){
12. if(! CorrectBothErrorRegisterPairs([Rx,Ry])) return false;
13. }else{
14. operatorAfterSelectedSubsitutePairs(Rx,Ry+1);
15. }
16. }
17. return true;
```

表 9 Illegal 模式错误寄存器偶对修正处理过程

Table 9 Correct process for Illegal mode error registerpairs

```
CorrectIllegalRegisterPairs([Rx,Ry])
1. bool Rj HaveUsed=false,Rk HaveUsed=false;
2. for each Ri∈R{
3.    if(Rx-1∈RepSetRi && (Ri!=Ry)) Rj HaveUsed=true;
4.    if(Rk∈RepSetRi && (Ri!=Rx)) RkHaveUsed=true;
5. }
6. if( Rx-1∈RepSet Ry){
7.    Rx-1→Ry;
8. }else if( Ry+1∈RepSet Rx){
9.    Ry+1→Rx;
10. }else if(Rx-1∉ SLI){
11.   operatorAfterSelectedSubsitutePairs(Ry,Rx-1);
12. }else if(Ry+1∉SLI){
13.   operatorAfterSelectedSubsitutePairs(Rx,Ry+1);
14. }else if(Rj∉SourceOperandSet && Rj∈SLI && ! RkHaveUsed){
15.   operatorAfterSelectedSubsitutePairs(Ry,Rx-1);
16. }else if(Rk∉SourceOperandSet && Rk∈SLI && ! RkHaveUsed){
17.   operatorAfterSelectedSubsitutePairs(Rx,Ry+1);
18. }else{
19.   if(!CorrectBothErrorRegisterPairs([Rx,Ry])) return false;
20. }
21. return true;
```

在错误寄存器偶对源操作数的处理过程中,为错误的寄存器对选取了替换寄存器对后,需要根据错误寄存器 Rerror 和替换寄存器 Rsubstitute 中值的活跃情况进行一系列的处理。首先,将当前处理操作中的 Rerror 替换为 Rsubstitute。然后,在当前指令前插入数据转移指令,将 Rerror 中的值转移到 Rsubstitute 中。最后,更新 SourceOperandSet,SLI 等集合中的内容,以便处理当前处理指令中还未修正的其他源操作数。选定替换寄存器偶对后,修正处理过程如表 10 所列。

表 10 LeftError 模式错误寄存器偶对修正处理过程

Table 10 Correct process for LeftError mode error registerpairs

```
bool CorrectLeftErrorRegisterPairs([Rx,Ry])
1. bool haveUsed=false;
2. foreach Ri∈R{
3.    if(Ry+1∈RepSetRi && (Ri!=Rx)) haveUsed=true;
4. }
5. if(haveUsed ){
6.    if(! CorrectBothErrorRegisterPairs([Rx,Ry])) Return false;
7. }
8. if(Ry+1∈RepSetRx){
9.    Ry+1→Rx;
10. }else{
11.   if(Ry+1∈SLI && Ry+1∈sourceOperandSet){
12.      if(!CorrectBothErrorRegisterPairs([Rx,Ry])) return false;
13.   }else{
14.      operatorAfterSelectedSubsitutePairs(Rx,Ry+1);
15.   }
16. }
17. return true;
```

operatorAfterSelectedSubsitutePairs 函数调用 IsContain(Rerror)函数判断 Rerror 是否存在于除当前处理源操作数之外的其他源操作数。在控制流进入当前处理指令之前,函数中 BeProtSet 集合的元素需要暂存到其他存储空间。修正处理当前指令中的所有错误寄存器偶对源操作数后,在当前指令的入口处仍然存在空闲寄存器。BeProtSet 集合中的元素则保存到空闲寄存器中,否则保存到内存中。在控制流退出当前处理指令后,需要将 BeProtSet 集合中的元素中的数据从占存的存储空间内恢复。函数中的 SLM 集合是集合 SLI 和 SLO 的交集。SLM 中的元素为源操作出口活跃变量。指令源操作出口指当前处理指令源操作数与目标操作数之间的点。该点活跃的变量称为源操作出口活跃变量。

3.3.2 指令目标操作数寄存器偶对修正算法

指令目标操作数寄存器偶对修正过程中,根据活跃变量分析结果,分析出指令源操作数出口处的各个寄存器空闲的状态信息,将所有寄存器偶对按照空闲状态分别放到空闲寄存器偶对、半空闲寄存器偶对、非空闲寄存器偶对 3 种集合中。然后,结合处理模式选择替换寄存器偶对。目标操作数的处理模式划分与源操作数的划分相同,因此就不再累述。为各个处理模式的目的操作数选择替换寄存器对的方法如下。

(1)BothError 处理模式。首先考虑使用空闲寄存器偶对替换错误的寄存器偶对。如果空闲寄存器偶对集合为空,则选择半空闲的寄存器偶对作为替换寄存器对。如果半空闲寄存器偶对集合为空或半空闲寄存器偶对中的活跃寄存器为当前处理指令的目标操作数,则选择非空闲寄存器偶对替换错误寄存器偶对。

(2)RightError 处理模式。假设当前处理的目的操作数为[R_x,R_y],其处理模式为 RightError。处理此模式的寄存器偶对目的操作数时,优先考虑使用寄存器偶对[R_x,R_{x-1}]替换错误寄存器偶对。如果 R_{x-1}为当前处理指令的目的操作数或者目的操作数中的一部分,则将当前处理的目的操作的处理模式改为 BothError。

(3)LeftError 处理模式。假设当前处理的目的操作数为[R_x,R_y],其处理模式为 LeftError。优先考虑使用寄存器偶对[R_{y+1},R_y]替换错误寄存器偶对。如果 R_{y+1}为当前处理指

令的目的操作数或者目的操作数中的一部分，则将当前处理的目的操作的处理模式改为 BothError。

(4) Illegal 处理模式。假设当前处理的目标操作数为 $[R_x, R_y]$，其处理模式为 Illegal。优先考虑使用寄存器偶对 $[R_x, R_{x-1}]$ 或者 $[R_{y+1}, R_y]$ 替换当前的处理操作数。如果 R_{x-1} 和 R_{y+1} 同时为当前处理指令的目的操作数或者某个目的操作数的一部分，则将当前处理的目的操作数的处理模式改为 BothError。

在确定替换寄存器偶对后，为了处理当前处理指令中还未处理的其他目的操作数和确保处理过程中不发生语义变换，我们需要进行信息更新和指令插入等处理。假设在处理某个目的操作数的过程中使用 Rsubstitute 替换 Rerror。首先，在当前处理指令的后面插入数据转移指令，把 Rsubstitute 中的值转移到 Rerror 中。其次，将 Rerror 的活跃状态设置为不活跃，并且从当前处理指令的目的操作数集合中删除 Rerror。然后，将其添加到当前处理指令的目的操作数集合中。如果 Rsubstitute 的活跃状态为不活跃，则将其活跃状态设置为活跃。在处理当前目的操作数之前，如果 Rsubstitute 是活跃寄存器，则需要在控制流进入当前处理指令之前将 Rsubstitute 中的值暂存到其他存储空间，并在当前指令出口处插入指令，将 Rsubstitute 中原来的值从总暂存空间中恢复。

4　实例分析

为了展示上述方法的优化效果，我们假设某处理器拥有 4 个物理寄存器，编号为 R_0-R_3，其中寄存器 R_1 和 R_0、R_3 和 R_2 分别可以作为寄存器偶对使用。下面以输入代码段 3(如表 11 中的左侧所列)为例来讨论上述算法。按照本文提出的无约束寄存器分配方法得到的寄存器分配后的代码如表 11 中的右侧所列，其中源代码的变量已经按照第一阶段的寄存器分配方案将变量替换为所指派的物理寄存器。

表 11　代码片段 3(左)以及无约束寄存器分配后的中间代码(右)

Table 11　Code fragment 3(left), Intermediate code after weak constrain register allocation(right)

输入代码片段 3	中间代码(无约束寄存器分配后)
I_0 :load $[a_1, a_0]$, address;	I_0 :load $[R_3, R_0]$, address;
I_1 : $b=[a_1, a_0]+[a_1, a_0]$;	I_1 : $R_2=[R_3, R_0]+[R_3, R_0]$;
I_2 : $c=[a_1, a_0]*[a_1, a_0]$;	I_2 : $R_1=[R_3, R_0]*[R_3, R_0]$;
I_3 : $d=[a_1, a_0]$;	I_3 : $R_0=[R_3, R_0]$;
I_4 : $e=b+c$;	I_4 : $R_0=R_2+R_1$;
I_5 : $f=c+d$;	I_5 : $R_3=R_1+1$;
I_6 : $g=e+1$;	I_6 : $R_2=R_0+1$;
I_7 : $[h_1, h_0]=[g, e]*[g, e]$;	I_7 : $[R_1, R_0]=[R_2, R_0]*[R_2, R_0]$;
I_8 : $[i_0, i_1]=[f, g]*[h_1, h_0]$;	I_8 : $[R_1, R0]=[R_3, R_2]*[R_1, R_0]$;

由寄存器偶对的使用约束条件可知，表 12 所列的中间代码中存在不符合寄存器偶对使用约束条件的寄存器偶对。本文将通过指令操作数寄存器偶对修正算法来处理代码中的错误寄存器偶对。在表 12 所列的中间代码中，指令 I_0，I_1，I_2，I_3 和 I_7 中存在错误寄存器偶对。指令 I_0 的目的操作数中存在错误寄存器偶对 $[R_3, R_0]$，并且错误寄存器偶对的处理模式为 Illegal。指令 I_1，I_2 和 I_3 的源操作数中存在错误寄存器偶对 $[R_3, R_0]$，并且错误寄存器偶对的处理模式为 Illegal。指令 I7 的源操作数中存在错误寄存器偶对 $[R_2, R_0]$，并且错误寄存器偶对的处理模式为 LiftError。通过数据流分析获取指令入口处或者指令出口处的活跃寄存器，结果如表 12 所列的右侧集合所示。其中，SLI_{Ix} 表示 Ix 指令入口处活跃物理寄存器集合，SLO_{Ix} 表示指令 Ix 指令出口处活跃物理寄存器集合。然后，结合处理模式选择相应的修正处理方法。最终，生成如表 13 右侧所列的目标代码。与表 13 左侧所列的 Briggs 算法生成的目标代码相比，本文的寄存器偶对分配算法提高了寄存器指派过程的成功率，减少了溢出恢复指令插入的数量，即减少了插入 store 和 load 指令的数量，提升了目标生成代码的运行效率。

表 12　修正过程中 SLI 和 SLO 的分析结果

Table 12　Result of SLI and SLO in corecting procedure

中间代码(无约束寄存器分配后)	
I_0 :load $[R_3, R_0]$, address;	$SLI_{I0}=\{?\}$, $SLO_{I0}=\{R_3, R_0\}$
I_1 : $R_2=[R_3, R_0]+[R_3, R_0]$;	$SLI_{I1}=\{R_3, R_0\}$, $SLO_{I1}=\{R_3, R_2, R_0\}$
I_2 : $R_1=[R_3, R_0]*[R_3, R_0]$;	$SLI_{I2}=\{R_3, R_2, R_0\}$, $SLO_{I2}=\{R_3, R_2, R_1, R_0\}$
I_3 : $R_0=[R_3, R_0]$;	$SLI_{I3}=\{R_3, R_2, R_0\}$, $SLO_{I3}=\{R_3, R_2, R_1, R_0\}$
I_4 : $R_0=R_2+R_1$;	
I_5 : $R_3=R_1+R_0$;	$SLI_{I7}=\{R_0\}$, $SLO_{I7}=\{R_3, R_0\}$
I_6 : $R_2=R_0+1$;	
I_7 : $[R_1, R_0]=[R_2, R_0]*[R_2, R_0]$;	
I_8 : $[R_1, R_0]=[R_3, R_2]*[R_1, R_0]$;	

表 13　本文及 Briggs 寄存器偶对分配算法生成的目标代码

Table 13　Briggs's algorithm and this paper algorithm generates the target code

Briggs 寄存器偶对分配算法	本文寄存器偶对分配算法
I_0 :load $[R_1, R_0]$, address;	I_0 :load $[R_3, R_2]$, address;
J_0 :Store address$_0$, $[R_1, R_0]$;	C_0 : $R_0=R_2$;
J_1 :Load $[R_1, R_0]$, *address*$_0$	C_1 : $R_2=R_0$;
I_1 : $R_0=[R_1, R_0]+[R_1, R_0]$;	I_1 : $R_2=[R_3, R_2]+[R_3, R_2]$;
J_2 :Load $[R_3, R_2]$, *address*$_0$	C_2 : $R_1=R_3$;
I_2 : $R_1=[R_3, R_2]*[R_3, R_2]$;	I_2 : $R_1=[R_1, R_0]*[R_1, R_0]$;
J_3 :Load $[R_3, R_2]$, *address*$_0$	C_3 :store *address*$_0$, R_2;
I_3 : $R_2=[R_3, R_2]$;	I_3 : $R_0=[R_3, R_2]$;
I_4 : $R_0=R_0+R_1$;	C_3 :load R_2, *address*0;
J_4 :*Store address*$_1$, R_0;	I_4 : $R_0=R_2+R_1$;
I_5 : $R_0=R_1+R_2$;	I_5 : $R_3=R_1+R_0$;
J_5 :*Store address*$_3$, R_0;	I_6 : $R_2=R_0+1$;
J_6 :Load R_0, *address*$_1$;	$C4$: $R_1=R_2$;
I_6 : $R_0=R_0+1$;	I_7 : $[R_1, R_0]=[R_1, R_0]*[R_1, R_0]$;
J_7 :*Store address*$_2$, R_0;	I_8 : $[R_1, R_0]=[R_3, R_2]*[R_1, R_0]$;
J_8 :Load R_3, *address*$_2$;	
J_9 : $R_1=R_3$;	
J_{10} :*Load* R_2, *address*$_1$;	
J_{11} : $R_0=R_2$;	
I_7 : $[R_3, R_2]=[R_1, R_0]*[R_1, R_0]$;	
J_{11} :*Load* R_0, *address*$_2$;	
J_{12} : $R_0=R_0$;	
J_{13} :*Load* R_1, *address*$_3$;	
I_8 : $[R_1, R_0]=[R_1, R_0]*[R_3, R_2]$;	

5　实验与讨论

考虑目标机为具有 32 个 32 位通用寄存器的超长指令字体系结构 DSP，它包括指令流控、SPE 以及 SM 3 个执行单元。其中，指令流控单元用于程序流控制，SPE 包含两个乘加单元和一个定点运算单元共 3 个独立的运算部件，SM 主要实现标量数据访存。每个部件在每个指令节拍最多可发射一条指令。

为了测试本文提出的优化算法对生成的目标代码执行效率的影响，考虑 svdcmp，MatrixMultiply，LmsFilter，FFT，

Convolution 和 BubbleSort 这 6 种经典的算法作为测试用例。我们将 6 个测试用例分别采用 Briggs 寄存器分配算法和本文提出的优化算法进行处理，得到相应的目标代码指令执行包数量。各个测试用例的执行包数量对比情况如图 3 所示。图 3 中每种算法的左栏为 Briggs 寄存器分配算法的实验结果，右栏为本文寄存器分配算法的实验结果。从中可以看出，本文算法在一定程度上减少了目标代码的指令执行包的数量。

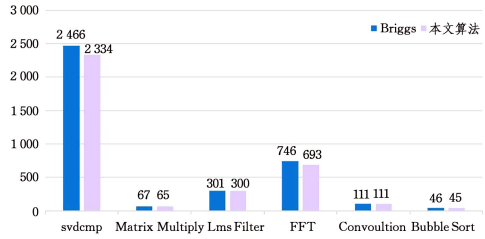

图 3 测试用例的总指令执行包数量情况

Fig. 3 Total number of instruction execution packages for the test case

在测试用例 svdcmp 的目标代码中存在如表 14 所列的指令序列，其中左侧为使用 Briggs 寄存分配算法编译测试用例生成的目标代码，右侧为使用本文算法编译测试用例生成的目标代码。指令 I_3 和 I_5 是为了溢出指令 I_1 目的操作数中的值而插入的。在寄存器分配第一次为指令 I_1 目的操作数中的值指派寄存器时，虽然有多个寄存器供其选择，但是此时不存在合法寄存器偶对，因此指令 I_1 目的操作数中的值需要溢出，从而需要插入指令 I_3 和 I_5。相反，在优化后的寄存器分配算法生成的目标代码中，没有插入对应的溢出指令。对比表 14 左右两侧的目标代码，很容易发现它们的语义是一致的，而且本文优化算法生成的目标代码的质量更高。也就是说，本文提出的寄存器分配算法提升了寄存器指派过程中寄存器使用的灵活性，进而减少了由于寄存器偶对使用约束条件导致的符号寄存器对的溢出，达到了优化目标代码的质量的目的。

表 14 svdcmp 测试用例目标代码片段

Table 14 Object code fragment of svdcmp test case

Briggs 寄存器偶对分配算法	本文优化算法
I_1:FRCPD. M_1 $R_{15}:R_{14}$, $R_1:R_0$	J_1:FRCPD. M1 $R_{29}:R_{28}$, $R_1:R_0$
I_2:NOP 1	J_2:NOP1
I_3:STDW. LS $R_1:R_0$, * $+AR_{15}[0]$	J_3: FMULD. M1 $R_{31}:R_{30}$, $R_1:R_0$, $R_{31}:R_{30}$
I_4:NOP 2	J_4:NOP5
I_5:LDDW. LS * $+AR_{15}[0]$, $R_1:R_0$	
I_6:NOP 5	
I_7:FMULD. M1 $R_{13}:R_{12}$, $R_1:R_0$, $R_1:R_0$	
I_8:NOP 5	

在实验过程中，除了统计了各个生成的目标代码指令执行包情况外，还统计了分别使用 Briggs 寄存器分配算法和本文优化算法编译各个测试用例所需要的时间，结果如表 15 所列。运行优化编译程序的机器配置为 Windows7. 0、Intel Core I3 CPU 及 6GB 内存。在 6 个测试用例中除了 svdcmp 测试用例，其他测试程序使用 Briggs 寄存器分配算法编译所需要的时间短于本文优化算法，其主要原因是本文优化算法比传统的乐观图着色寄存器分配算法多了寄存器偶对修正过程。编译除 svdcmp 外的其他测试用例时，Briggs 算法与本文优化算法中溢出处理过程重复的次数相同。因此，使用本文优化算法编译其他测试用例所需要的时间要稍微长一些，但是增加的时间不多。从表 15 可以看出，使用 Briggs 寄存器分配算法编译 svdcmp 测试用例所需要的时间长于本文优化算法。其原因在于，编译 svdcmp 测试用例时，本文优化算法减少了寄存器分配过程中的溢出处理子过程的运行次数。也就说，如果 Briggs 算法寄存器分配过程中存在寄存器偶对使用约束条件引起的寄存器溢出，则其寄存器分配处理的时间长于本文算法。本文的优化算法是针对 DSP 处理器设计的，而且 DSP 程序具有一次编译长久使用的特点，从而对编译时间的容忍度更大。因此，从寄存器分配处理时间角度来看，本文算法适用于 DSP 程序的编译。

表 15 测试用例编译过程中寄存器分配处理时间

Table 15 Registers allocate time during test case compilation

(单位:ms)

算法	svdcmp	MatrixMultiply	LmsFilter	FFT	Convolution	BubbleSort
Briggs	8052.2	18	118.2	654.8	42	10
本文算法	5379.6	21.6	133.4	681	48.4	12.2

结束语 现代高性能 DSP 中广泛支持寄存器偶对的使用。本文针对寄存器偶对的使用约束条件，提出了一种基于弱约束指派的 DSP 寄存器偶对分配算法，为指令的寄存器偶对操作数提供了一种更加灵活的寄存器偶对指派方法。同时，作为该指派方法的补充，本文提出了寄存器偶对操作数修正方法，它们两者一起构成了一个指派时无约束的 DSP 寄存器偶对分配算法。该算法解除了指令寄存器偶对操作数在寄存器指派时必须指派寄存器偶对这一要求，提升了指令操作数对寄存器的使用灵活性。本文算法能够减少由于寄存器偶对使用约束条件导致的符号寄存器对的溢出，提升了目标代码的质量。实验结果表明，本文提出的优化算法对多数典型的 DSP 算法是有效的。

参 考 文 献

[1] LORENZ M, WEHMEYER L, THORSTEN D. Energy aware compilation for DSPs with SIMD instructions[C]// Proc. of the Joint Conf. on Languages, Compilers and Tools for Embedded Systems. 2002, 37(7): 94-101.

[2] NI Y H, CHEN W W, WANG L, et al. Optimization of Register Allocation Strategy for MLC STT-RAM[J]. Computer Science, 2018, 45(S1): 562-567.

[3] EISL J, MARR S, WÜRTHINGER T, et al. Trace Register Allocation Policies: Compile-Time vs. Performance Trade-Offs [C]// Proc. of the 14th International Conf. on Managed Languages and Runtimes. 2017: 92-104.

[4] POLETTO M, SARKAR V. Linear scan register allocation[J]. ACM Trans. on Programming Languages and Systems, 1999, 21(5): 895-913.

[5] WIMMER C, FRANZ M. Linear scan register allocation on SSA form[C]// IEEE/ACM International symp. on Code Generation and Optimization. 2010, 170-179.

[6] KANANIZADEH S,KONONENKO K. Improving on Linear Scan Register Allocation[J]. International Journal of Automation and Computing,2018,15(2):228-238.
[7] CHAITIN G J. Register allocation and spilling via graph coloring[C]// Symp. on Compiler Construction. 1982:98-105.
[8] CHAITIN G J,AUSLANDER M A,CHANDRA A K,et al. Register allocation via coloring[J]. Computer Languages,1981:47-57.
[9] BRIGGS P,COOPER K D,TORCZON L. Improvements to graph coloring register allocation[J]. Trans. on Programming Languages and Systems,1994,16(3):428-455.
[10] WU S N,LI S K. A Hyrid Evolutionary Algorithm for Register Allocation of Embedded Processors [J]. Computer Science,2007,34(8):278-280.
[11] NICKERSON B R. Graph coloring register allocation for processors with multi-register operands[J]. Programming Language Design and Implementation,1990,25(6):40-52.
[12] BRIGGS P,COOPER K D,TORCZON L. Coloring register pairs [J]. ACM Letters on Programming Languages and Systems,1992,1(1):3-13.
[13] SMITH M D,HOLLOWAY G. Graph-coloring register allocation for irregular architectures[J]. Submitted to PLDI,2000,1:1-8.
[14] Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation[G]. 2019.
[15] Processor Programming Reference (PPR) for AMD Family 17h Models 01h,08h,Revision B2 Processors[G]. Advanced Micro Devices. 2019.
[16] YHFT-Matrix2 DSP instruction set manual[G]. Institute of Microelectronics. College of Computer Science and Technology, National University of Defense Technology. 2013.
[17] ARM946E-S (Rev1) System-on-Chip DSP enhanced processor Product Overview[G]. ARM Limited. 2000.

TANG Zhen, born in 1994, postgraduate. His main research interests include DSP compilation and so on.

HU Yong-hua, born in 1981, Ph.D, professor, supervisor. His main research interests include DSP compilation and so on.

(上接第 574 页)

[2] YU Y,JONES J A,HARROLD M J. An Empirical Study of the Effects of Test Suite Reductionon Fault Localization[C]// Proceedings ofInternational Conference on Software Engineering. Leipzig,Germany,2008:201-210.
[3] HAO D,ZHANG L,PAN Y,et al. On Similarity-Awareness in Testing-Based Fault Localization[J]. Automated Software Engineering,2008,15(2):207-249.
[4] ZHANG X,GU Q,CHEN X,et al. A Study of Relative Redundancy inTest-Suite Reduction While Retaining or 1mproving Fault-LocalizationEffectiveness[C]// Proceedings of ACM Symposium on Applied Computing. Sierre,Switzerland,2010:2229-2236.
[5] 遗传算法系列(3)交叉算法[EB/OL]. http://www.cppblog.com/feng/archive/2008/06/18/53870.html.
[6] CHEN G L,WANG X F,ZHUANG Z Q,et al. Genetic Algorithm and Its Application[M]. People Post Press,2001.
[7] HAN X M,WANG L M. Artificial Immune Algorithm and Its Application [M]. Electronic Industry Press,2013.
[8] MA J,SHI G. Theory and Application of Artificial Immune Algorithm [M]. Northeastern University Press,2014.
[9] ABREU R,ZOETEWEIJ P,VAN GEMUND A J. On the Accuracy of Spectrum-based Fault Localization [C]// Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION. Windsor,USA,2007:89-98.
[10] ABREU R,ZOETEWEIJ P,VAN GEMUND A J. An Evaluation of Similarity Coefficients forSoftware Fault Localization [C]// Proceedings of Pacific Rim International Symposium on Dependable Computing. Riverside,USA,2006:39-46.
[11] HAN X M,WANG L M. Artificial Immune Algorithm and Its Application [M]. Electronic Industry Press,2013.
[12] MA J,SHI G. Theory and Application of Artificial Immune Algorithm [M]. Northeastern University Press,2014.
[13] 聚类之高斯混合模型与 EM 算法[EB/OL]. https://www.cnblogs.com/Luv-GEM/p/10851395.html.

ZHANG Hui, born in 1982, Ph.D, lecturer. Her main research interests include fault localization and software testing.