基于两类寄存器互为缓存方法的DSP寄存器分配溢出处理优化算法

邱亚琼 胡勇华 李 阳 唐 镇 石 林

(湖南科技大学计算机科学与工程学院 湖南 湘潭 411201)

摘 要 寄存器是处理器硬件中有限的宝贵资源,这使得寄存器分配成为编译器中最为关键的过程之一。影响寄存器分配效果的关键因素之一是溢出带来的访存开销。针对DSP处理器具有两类通用寄存器的情况,以图着色全局寄存器分配方法为基本方法,提出两类寄存器间的一种互补利用策略和相应的寄存器溢出优化算法。该策略改进了传统图着色方法,通过生命周期分析的结果,将同类寄存器分配候选者之间的冲突关系和不同类寄存器分配候选者之间的冲突关系区分开来,并把它们表示在一张无向图中。与传统的图着色算法相比,改进的算法能充分考虑不同类寄存器之间的相互约束关系,减少寄存器溢出时的访存操作,从而有利于提高代码的性能。

关键词 寄存器分配,编译器,图着色方法,寄存器溢出,优化

中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2019.06.029

Optimization Algorithm of Complementary Register Usage Between Two Register Classes in Register Spilling for DSP Register Allocation

QIU Ya-qiong HU Yong-hua LI Yang TANG Zhen SHI Lin

(School of Computer Science and Engineering, Hunan University of Science and Technology, Xiangtan, Hunan 411201, China)

Abstract Register allocation has become one of the most important optimization techniques for compiler for that registers are limited and valuable resources in hardware architecture of computer. One of the key factors affecting the results of register allocation is the access and storage costs incurred from spilling signed registers. For DSP architectures with two classes of general-purpose registers, this paper proposed a complementary utilization strategy between the registers and a corresponding register spilling optimization algorithm on the basis of graph coloring register allocation method. Through distinguishing the interference between candidates of the same register class from those of different register classes, an undirected graph is built by improving the analysis for variables' live ranges. Compared with the conventional graph coloring register allocation, the improved algorithm fully consideres the interferences among the register allocation candidates for two register classes, thus achiving less memory access operations in register spilling and higher code performance.

Keywords Register allocation, Compiler, Graph coloring algorithm, Register spilling, Optimization

1 引言

寄存器分配技术是编译器中最为关键的优化技术之一。研究人员经过大量的研究,发现寄存器分配是NP(NP-complete)完全问题[1-2],即使处理器只有一个寄存器也是如此[3]。然而,寄存器作为计算机体系结构中读写速度最快的存储空间,同时也是最稀缺和宝贵的存储资源[4-5],其使用效率的高低将直接影响处理器的性能[6]。

20世纪80年代,Chaitin等[7]创造性地将寄存器分配问题转化成对一张寄存器冲突图的染色问题[8-9]。图着色是解决寄存器分配问题最有效的方法,但是Chaitin等的启发式算法也存在不足,即数据溢出极大地增加了代码复杂度和能耗[10],这在该算法中没有被考虑和解决。传统的溢出处理办法是将溢出候选者(通常是符号寄存器)存放到存储器,待到合适的时机并且有空闲的寄存器时,再将数据恢复到寄存器。当处理器中只有一种类型的通用寄存器时,这一方法较为有效。这种方法虽然缓解了寄存器资源不足的问题,但缺点是代码执行速率显著降低,不利于降低能耗。

近年来,随着微电子与微处理器技术的进步和处理器应用的发展,DSP处理器得到了快速发展,一些先进的DSP处理器中存在两类或两类以上的通用寄存器[11-12]。尽管一条普通指令的操作数只能是其中的某一种寄存器,但可以通过

到稿日期:2018-04-17 返修日期:2018-07-16 本文受国家自然科学基金(61308001),湖南省自然科学基金(2017JJ3087)资助。

邱亚琼(1991-),女,硕士,主要研究领域为DSP编译;胡勇华(1981-),男,博士,教授,主要研究领域为DSP编译,E-mail:hyhyt@126.com(通信作者);李 阳(1995-),女,硕士,主要研究领域为DSP编译;唐 镇(1994-),男,硕士,主要研究领域为DSP编译;石 林(1986-),男,主要研究领域为图形处理器与分布式计算。

专门的数据传送指令在不同类的寄存器之间传送数据。针对这种硬件特征，本文提出了一种基于全局图着色的寄存器分配溢出[13]优化策略。本策略对传统寄存器溢出处理方法进行了改进优化，通过对生命周期的分析，将不同类别的寄存器分配约束关系限制在一张无向图中，在不同类别的寄存器候选者的生命周期不重叠的前提下实现寄存器资源的互补利用。

2　问题分析

在图着色寄存器分配方法中，溢出处理的基本方法是：在待溢出网（一种典型寄存器分配的候选者叫“网”，网是寄存器分配的对象，指相交的定值使用链[14]）的定值之后和使用之前插入相应的溢出指令和恢复指令，将数据溢出到存储空间。为了使整个程序因为溢出所花费的溢出代价最小，需要计算插入相应的溢出指令和恢复指令的代价，在修剪冲突图的过程中按溢出代价由小到大的顺序存放到栈中，这样在进行寄存器指派（着色）的过程中，溢出代价大的优先分配寄存器，在发生溢出时，就可选择溢出代价较小的网。

当寄存器资源不足时，进行溢出处理可以很好地解决这个问题。但是溢出不仅增加了目标代码的代码量[15]，也增加了对存储器的访问和存储压力。本文以减弱这些不利影响为目标，考虑先进的 DSP 处理器中存在两类或两类以上通用寄存器的情况，在进行符号寄存器溢出时增加对两类寄存器的互补利用的处理，以减少以上不利影响。本文优化算法利用在寄存器之间传递数据比在寄存器和存储器间传送数据速度快的优势，在发生溢出时，将数据暂时缓存到另一类寄存器中，直到对被溢出的数据进行运算前再恢复该数据。值得注意的是，这里只是临时将数据传送到其他类别寄存器进行缓存，如果要使用被缓存的数据进行其他处理，仍然要先将数据传送回某个本类别寄存器。此外，我们假定两类寄存器的位宽是相同的。因为本文算法利用的是不同类别寄存器的空闲期，所以不会引起另一类寄存器溢出。

本文算法以传统的图着色寄存器分配方法为基础，改进的溢出处理包括以下几个方面：1）改进邻接矩阵和邻接接表。邻接矩阵是节点间冲突关系的一种表示方法。邻接表是一个链表，其中表元素记录了网的溢出代价、指派成功的寄存器等信息。本文优化算法中，记录项添加了在两类寄存器间进行数据传送的传送代价，以及网和另一类的各个寄存器的冲突关系。添加的信息在确定可借用的另一类寄存器时是很重要的。2）计算两类寄存器间数据传送的传送代价。传统的寄存器分配算法在计算溢出代价时都是通过到存储器的溢出指令和恢复指令（load/store 指令）的权重来计算的。我们假设有专门的传送指令模板：将数据从某一个 A 类寄存器传送到某一个 B 类寄存器的指令助记符模板是 MOV $a->b$，将数据从该 B 类寄存器传送到该 A 类寄存器的指令助记符模板是 MOV $b->a$。如果存在满足互补利用原则的另一类寄存器，则数据可以缓存到这一类的某个寄存器中。这样的溢出和恢复是以传送指令来实现的，此时溢出代价应该以 MOV 指令的权重来计算。3）判断网之间的冲突关系。网之间的冲突关系是决定指派哪一个寄存器的判断依据，但是传统的寄存器分配只判断同类别网之间的冲突关系。在本文优化算法中，因为要判断可用的它类寄存器，而能够让溢出网互补利用的它类寄存器必须是活跃区间不重叠的，所以在判断冲突关系时要考虑网和它类物理寄存器网之间的冲突关系。4）数据溢出目标空间的判断。在实际情况中，溢出到另一类寄存器的传送代价不一定比溢出到存储器的溢出代价小，因此需要进行更精确的计算比较来获得尽可能大的收益。

3　算法实现

3.1　寄存器溢出处理分析

本文中寄存器分配方法以图着色全局寄存器分配方法为基础，在寄存器分配过程中，需要目标代码的基本信息和分析目标代码得到的数据流和控制流信息。我们假设已知 *UD* 链（使用定值链）、*DU* 链（定值使用链）、网、网的活跃区间等，本文的溢出优化算法的顶层结构包括如下几个步骤：

1）*Judge_NeedSpillRegisters*()，判断是否需要进行寄存器溢出。只有在存在寄存器溢出的情况下，才进行后续步骤的处理。

2）*Find_SpillWebs*()，找到所有待溢出的网。

3）*Get_SpillWebsWithOtherTypeRegistersConflictShip*()，获取待溢出网与它类寄存器的冲突关系，网的生命期和寄存器活跃区间不重叠，则表示这个它类寄存器是可借用的，在邻接表的记录项中用标志 $flag=0$ 来标记。

4）*Determine_OtherTypeUsableRegister*()，确定可借用的它类寄存器的类别和名称。当有空闲的它类寄存器且当前待溢出网到此寄存器的传送代价最小时，此寄存器就可以供溢出网使用，即将当前待溢出网缓存到相应的它类寄存器。

5）*Gen_SpillCode*()，生成溢出指令和恢复指令。找到合适的它类寄存器来缓存待溢出网，以生成相应的溢出和恢复传送指令，在合适的位置插入相应的指令；如果没有这样的它类寄存器作为数据溢出的缓存，则将待溢出网直接溢出到存储器中，这与传统的溢出处理方法是一致的。

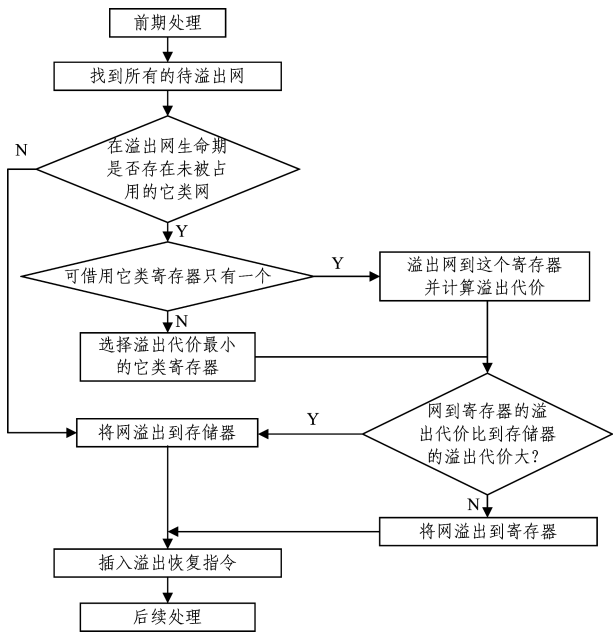

图 1　单个寄存器溢出处理流程

Fig. 1　Flowchart of register spilling processing

在寄存器溢出优化算法中，首先要构建待溢出网与不同类别寄存器之间的冲突关系，这个关系采用冲突图来表示。这里采用邻接表来表示冲突图，邻接表的一条记录中除包含传统邻接表记录外，还包括待溢出网到它类寄存器的溢出代价，以及待溢出网的相邻节点。具体的寄存器溢出处理流程如图 1 所示。

3.2 分析网的冲突关系

通常节点之间的冲突关系都是存在于同一类寄存器之间，即任何量都可以存放在任何寄存器中，这时只要简单地找出这个图的 R 色着色，并指派不同的颜色到不同的寄存器即可。但是实际情况不一定如此，当处理器存在两类或两类以上指定了不同功能的寄存器集合时，应对不同种类的寄存器分配分别进行处理。传统的邻接矩阵元素为一个布尔变量，这个变量的值只表示相同类寄存器之间的冲突关系，而不同类寄存器之间不存在冲突。本优化方法中所有的网按类别不同以及生命周期的重叠情况表示它们之间的冲突关系，变量的值可分别取 -1，0，1。在判断两个网的冲突关系前，须先获取它们的类别信息，新建邻接矩阵的关键步骤如下。1）判断物理寄存器的网之间的冲突关系。默认它们是不冲突的，因为即使是同时活跃的物理寄存器的网，其寄存器也已经确定，不会存在冲突。2）判断物理寄存器的网和符号寄存器的网之间的冲突关系。如果两个网是同类别的，则对它们进行活跃量分析：如果它们的活跃期重叠，则此时矩阵元素值为 1，表示它们相冲突；如果它们的活跃期不重叠，则矩阵元素值为 0，表示它们不冲突。如果两个网是不同类别的，同样对它们进行活跃量分析：如果两者生命周期重叠，则表示它们的冲突关系的元素值为 -1，说明这两个网之间不能作为互补的溢出缓存；如果两个网生命周期不重叠，则表示它们的冲突关系的元素值为 0，这说明这两个寄存器之间可以作为互补的溢出缓存。3）判断符号寄存器的网之间的冲突关系，这一步的执行过程与步骤 2）是一样的。

改进邻接表。传统的邻接表的记录类包含的数据元素通常有：网的溢出存储单元的地址偏移量（相对于基址）、网溢出到存储器的溢出代价、与网相邻的节点（即相冲突的同类别的网）的集合等。在本文方法改进的邻接表中，对于每一个网，其表的记录项类增加了 4 个元素：与当前网没有冲突的它类寄存器的集合 UsableRegsofOtherClass、与当前网冲突的它类寄存器的集合 ConflictRegsofOtherClass、作为溢出缓存的它类寄存器的名称 UsablePhyRegName 和最终溢出缓存到某一个它类寄存器的传送代价 SpillCostToReg。对于待溢出网，遍历其在邻接矩阵中表示的与剩下的网之间的冲突关系：如果元素值为 0，则判断两个网的类别，如果不同类则将冲突表示为 0 的另一个网添加到 UsableRegsofOtherClass 中；如果元素值为 -1，则将另一个网添加到 ConflictRegsofOtherClass 中。

3.3 计算不同类寄存器间数据传送的传送代价

在寄存器分配方法中，根据具有较小的度数的节点先入栈、且当节点度数相同时先将溢出代价较小的网优先入栈的原则，修剪冲突图，在给图中节点着色前将所有的节点入栈。在基本的寄存器分配方法中，将待溢出网溢出到存储器的溢出代价的计算公式如式(1)所示，这也是将节点压栈时判断入栈先后顺序的依据。其中，def 和 use 分别是网 w 的各个定值和使用；$defwt_s$ 和 $usewt_s$ 是网溢出到存储器的存储指令和恢复指令各自的权重；$depth(def)$ 和 $depth(use)$ 是 def 和 use 所在基本块的循环嵌套深度。

$$cost=defwt_r*\sum_{def\in w}10^{depth(def)}+usewt_r*\sum_{use\in w}10^{depth(use)} \tag{1}$$

不同的处理器体系结构一般都有与其对应的一套指令集，指令集中不同类别的寄存器之间数据传送的指令名称不尽相同，数据传送代价也不相同。在已经获取了某硬件体系结构的指令集信息的前提下，在计算数据传送代价前，建立一个矩阵来表示数据缓存时的传送指令信息，矩阵元素分别记录指令名称和指令执行时间。要选择传送代价最小的寄存器作为数据缓存的候选，建立这样的二维矩阵来确定溢出的它类寄存器是很方便的。不同类寄存器间数据传送的溢出代价如式(2)所示：

$$r\_cost=defwt_s*\sum_{def\in w}10^{depth(def)}+usewt_s*\sum_{use\in w}10^{depth(use)} \tag{2}$$

其中，$defwt_r$ 和 $usewt_r$ 是待溢出网缓存到另一类寄存器的传送指令和恢复指令各自的权重。

3.4 确定待溢出网溢出缓存的另一类寄存器

在确定了待溢出网与它类寄存器的冲突关系之后，查找并筛选出可借用的它类寄存器，并确定记录项 *UsableOtherPhyReg*。判断集合 UsableRegsofOtherClass 是否为空，若为空，则将网溢出到存储器；否则，选择某一个合适的另一类寄存器来缓存溢出网。计算出待溢出网到此空闲寄存器的传送指令的权重，从而计算出数据传递的溢出代价。假设已经计算出当前网到其他类别寄存器的传送代价 r_cost，并存放在集合 CostofDiffrentClassReg 中。要注意一点，寄存器传送代价不一定小于存储代价，如果寄存器之间数据的溢出代价和恢复代价比溢出到存储器的代价大，那么仍然应该将此网溢出到存储器中。

算法 1 确定溢出的它类目标寄存器

```
Determine_UsableOtherTypeRegister()
begin
i,j,MinSpillCost:interger
if(web[i].UsableRegsofOtherClass.length)
  MinSpillCost:=CostofDiffrentClassReg
    (web[i].UsableRegsofOtherClass.head())
  for j:=1 to web[i].UsableRegsofOtherClass.length
    do
    if(MinSpillCost < CostofDiffrentClassReg
        (web[i].UsableRegsofOtherClass[j]))
        continue
    fi
    if(MinSpillCost > CostofDiffrentClassReg
        (web[i].UsableRegsofOtherClass[j]))
```

```
        if(web[i]. UsableRegsofOtherClass[j]. Num)
        MinSpillCost=CostofDiffrentClassReg(
        web[i]. UsableRegsofOtherClass[j])
      fi
    fi
  od
  if(MinSpillCost < CostofRegToStorage)
    return web[i]. UsableRegsofOtherClass[j]
  fi
  fi
  end//Determine_OtherTypeUsableRegister
```

4　优化效果分析

为了展示上述方法的有效性，我们假设某处理器具有如下寄存器资源特性[16]：将数据从 A 类寄存器写入存储器需要 4 个节拍，而从存储器读取数据到 A 类寄存器需要 6 个节拍，A 类寄存器到 B 类寄存器的数据传送需要 2 个节拍，B 类寄存器到 A 类寄存器的数据传送需要 2 个节拍；验证程序设定可用的物理寄存器有 R_{A1}，R_{A2}，R_{A3}，R_{A4}，R_{B1}，R_{B2}，R_{B3}，R_{B4}，但在程序运行开始时 R_{B4} 寄存器已经被用来存放数据段地址，直到程序运行结束。给定代码实例为：实现复数 $f1=2+i$ 与复数 $f2=5+2i$ 相乘，并输出结果 f。程序示例源码 C 及符号寄存器替换后的代码 S 如表 1 所列。

表 1　程序示例源代码

Table 1　Source program

	输入的源程序代码 S	对应的 C 语言代码
$t1$	$s1<-2$	$a=2$
$t2$	$s2<-1$	$b=1$
$t3$	$s3<-5$	$c=5$
$t4$	$s4<-2$	$d=2$
$t5$	$s5<-s1*s3$	$e=a*c$
$t6$	$s6<-s2*s4$	$f=b*d$
$t7$	$s7<-s1*s4$	$g=a*d$
$t8$	$s8<-s2*s3$	$h=b*c$
$t9$	$s9<-s5-s6$	$n1=e-f$
$t10$	$s10<-s7+s8$	$n2=g+h$
$t11$	$printf("f=\%d+\%d\backslash n", s9, s10)$	$printf("f=\%d+\%d\backslash n", n1, n2)$

对于这段代码，我们进一步假定各变量的网的寄存器类型均为 A 类寄存器。对该代码进行冲突关系分析，可以得到各个符号寄存器的网以及物理寄存器的网的冲突关系表示的冲突图，如图 2 所示。

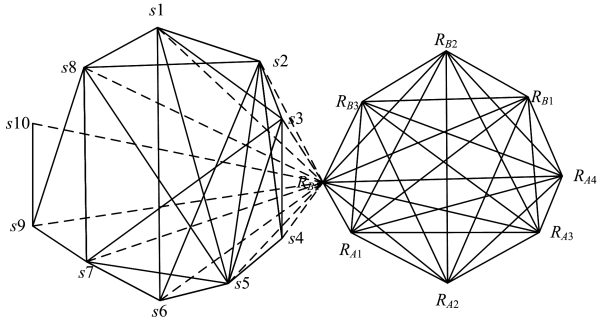

图 2　源代码变量的冲突图

Fig. 2　Conflict graph of source code variables

在图 2 中，节点的邻接节点最多为 4 个，所需要的 A 类寄存器至少为 5 个，现有的 A 类寄存器资源显然是不足的，因此要进行寄存器的溢出。由式(1)可计算网的溢出代价，具体的溢出代价的计算值如表 2 所列。

表 2　网溢出到存储器的溢出代价

Table 2　Cost of spilling to storage

网	溢出代价 *cost*
$s1$	4+6+6=16
$s2$	4+6+6=16
$s3$	4+6+6=16
$s4$	4+6+6=16
$s5$	4+6=10
$s6$	4+6=10
$s7$	4+6=10
$s8$	4+6=10
$s9$	4+6=10
$s10$	4+6=10

接下来，修剪冲突图，采用乐观启发式方法依次删除图中的节点，并将节点存放在一个栈中，符号寄存器的网的最终入栈结果如图 3 所示。

图 3　符号寄存器的网的最终入栈结果

Fig. 3　Result of popping-stack of symbol register webs

然后将这些节点弹出栈并指派寄存器。$s1$，$s2$，$s3$，$s4$ 能够成功指派，这里溢出 $s5$ 和 $s6$ 是较合适的。由式(2)分别计算 $s5$ 和 $s6$ 溢出到 B 类寄存器的溢出代价，结果如表 3 所列。

表 3　溢出网缓存到 B 类寄存器的溢出代价

Table 3　Spilling cost transfering web to register of class B

溢出网	溢出到 A 类寄存器的溢出代价
$s5$	2+2=4
$s6$	2+2=4

因为溢出代价都一样，所以当有多个寄存器可供待溢出网进行数据缓存时，可以根据寄存器的编号来选择编号最小的那一个作为溢出目标寄存器，这里选择溢出 $s5$ 到 R_{B1}，溢出 $s6$ 到 R_{B2}。

值得注意的是，虽然在大多数情况下寄存器间的数据传送代价小于数据溢出到存储器的代价，但逻辑上这不是绝对的，在进行数据溢出时，仍然要进行溢出代价的比较。比较 *cost* 和 *r_cost* 的大小，以确定两个待溢出网都溢出到寄存器，而不是存储器。

确定了溢出网和溢出缓存的它类寄存器之后，插入相应的溢出指令和恢复指令，此时的程序示例如表 4 所列，即分别在 $t6$ 和 $t7$ 时刻插入溢出指令，在 $t11$ 和 $t12$ 插入恢复 $s5$ 和 $s6$ 的指令。

将网溢出之后，各个网之间的冲突关系发生了变化，保证了相邻节点能够指派不同的寄存器。通过对比分析发现，将它类寄存器作为溢出节点的缓存可节省 6 个指令周期。本优化算法中，随着目标代码的数据溢出的增多和代码复杂性的增加，将会得到更大的寄存器资源利用率。

表 4 插入指令后的代码

Table 4 Code after instering spilling and resyoring instruction

	输入源码	插入指令后的代码
$t1$	$s1<-2$	$s1<-2$
$t2$	$s2<-1$	$s2<-1$
$t3$	$s3<-5$	$s3<-5$
$t4$	$s4<-2$	$s4<-2$
$t5$	$s5<-s1*s3$	$s5<-s1*s3$
$t6$		$R_{B1}<-s5$
$t7$	$s6<-s2*s4$	$s6<-s2*s4$
$t8$		$R_{B2}<-s6$
$t9$	$s7<-s1*s4$	$s7<-s1*s4$
$t10$	$s8<-s2*s3$	$s8<-s2*s3$
$t11$		$s5<-R_{B1}$
$t12$		$s6<-R_{B2}$
$t13$	$s9<-s5-s6$	$s9<-s5-s6$
$t14$	$s10<-s7+s8$	$s10<-s7+s8$
$t15$	printf("$f=\%d+\%d\backslash n$", $s9,s10$)	printf("$f=\%d+\%d\backslash n$", $s9,s10$)

下面以银河飞腾 Matrix 处理器为例，使用其标量处理单元，通过对比两种排序算法的代码来分析本文算法的效果。假设可用的通用寄存器只有 8 个，分别为 $R_0,R_1,R_2,\cdots,R_7$。此外，处理器有 16 个基址寄存器 $A_{R_0}-A_{R_{15}}$。比较对 n 个数据进行冒泡排序和选择排序对应的汇编程序可知，本文算法与传统的溢出到存储器的算法各自所需的执行时间以及本文算法能够节省的时间如表 5 所列。可以看出，计算数据量越大，用本文方法进行溢出处理所节省的时间越多。

表 5 测试结果对比

Table 5 Comparison of test results

	优化前的指令执行时间	优化后的指令执行时间	节省的指令执行时间
冒泡排序	$31n^2+11n+2$	$29n^2+9n+2$	$2n(n+1)$
选择排序	$22n^2+26n+10$	$22n^2+24n+8$	$2(n+1)$

结束语 本文通过分析寄存器溢出问题，提出针对具有两类通用寄存器的寄存器溢出优化策略。与传统的溢出数据存放方法相比，在进行寄存器溢出处理时，不再单纯地将数据存放到存储器中，而是同时考虑将一种寄存器类别对应的待溢出寄存器分配候选者存放在其他类别的寄存器中的可能性。在传统的图着色方法的基础上，改进了生命周期分析、寄存器分配候选者之间的冲突关系的描述模型和溢出处理方法。利用两类寄存器之间的空闲时间片段，提高了寄存器这一稀缺资源的使用率。该策略提高了代码的执行效率，节省了代码执行的时间成本。

参 考 文 献

[1] AHO A V, SETHI R, ULLMAN J D. Compilers, Principles, Techniques[M]. Boston: Addison wesley, 1986.

[2] RAU B R, LEE M, TIRUMALAI P P, et al. Register allocation for software pipelined loops[C]// Acm Sigplan Conference on Programming Language Design & Implementation. 1992: 283-299.

[3] REN K, YAN X L, SUN L L, et al. ASIP register allocator based on improved graph coloring algorithm[J]. Journal of Zhejiang University, 2010(12): 2309-2313. (in Chinese)
任坤，严晓浪，孙玲玲，等. 基于改进图染色算法的 ASIP 寄存器分配器[J]. 浙江大学学报(工学版)，2010(12)：2309-2313.

[4] JIANG J, WANG C, YU H M. A Local Register Allocation Optimization Strategy[J]. Computer Applications and Software, 2013(12): 215-217. (in Chinese)
姜军，王超，尉红梅. 一种局部寄存器分配的优化策略[J]. 计算机应用与软件，2013(12)：215-217.

[5] BRIGGS P, COOPER K D, KENNEDY K, et al. Coloring, heuristics for register allocation[J]. ACM SIGPLAN Notices, 1989, 24(7): 275-284.

[6] CHAITIN G J. Register allocation & spilling via graph coloring [J]. ACM Sigplan Notices. ACM, 1982, 17(6): 98-105.

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

[8] ACM Transactions on Programming Languages and Systems [M]. Association for Computing Machinery, 1979.

[9] BRIGGS P, COOPER K D, TORCZON L. Coloring register pairs [J]. Acm Letters on Programming Languages & Systems, 1992, 1(1): 3-13.

[10] BRIGGS P, COOPER K D, TORCZON L. Improvements to graph coloring register allocation[J]. Acm Transactions on Programming Languages & Systems, 1994, 16(3): 428-455.

[11] PENG Z S, DUAN Y Y, WANG B. Introduction of Mainstream Processor Architecture and Intel Microarchitecture Development [J]. Information System Engineering, 2017(3): 29-31. (in Chinese)
彭圳生，段妍羽，王赟. 主流处理器架构及英特尔微架构发展分析[J]. 信息系统工程，2017(3)：29-31.

[12] WANG X, FU J W, HE H. Instruction Processing Method Based on ARM Instruction Set in General DSP[J]. Microelectronics and Computers, 2016, 33(9): 10-14. (in Chinese)
王旭，付家为，何虎. 基于 ARM 指令集的通用 DSP 中指令相关处理方法[J]. 微电子学与计算机，2016，33(9)：10-14.

[13] NICKERSON B R. Graph coloring register allocation for processors with multi-register operands[C]// ACM Sigplan 1990 Conference on Programming Language Design and Implementation. ACM, 1990: 40-52.

[14] WOLFE J. Optimizing Super-compilers for Supercomputers [M]// Optimizing Super-compilers for Supercomputers. MIT Press Cambridge, MA, USA, 1990.

[15] HUANG L, FENG X B. Combined instruction scheduling and register allocation techniques[J]. Application Research of Computers, 2008, 25(4): 979-982. (in Chinese)
黄磊，冯晓兵. 结合的指令调度与寄存器分配技术[J]. 计算机应用研究，2008，25(4)：979-982.

[16] Instruction Set Manual of YHFT-Matrix2 DSP[G]. Institute of Microelectronics, National University of Defense Technology. 2013. (in Chinese)
YHFT-Matrix2 DSP 指令集手册[G]. 国防科学技术大学计算机学院微电子所. 2013.