维普资讯 http://www.cqvip.com

栈寄存器分配优化*)

刘 旸 张兆庆

（中科院计算技术研究所系统结构室先进编译组 北京100080）

摘 要 寄存器栈在减少程序调用时的内存访问上发挥了重要作用。但是，并非任何时候栈寄存器的使用都是没有代价的，有时栈溢出的代价甚至非常高。为了解决这个问题，本文提出了一种解决自递归函数中大量栈寄存器的使用导致过高栈溢出代价的算法，对寄存器分配中的简化过程进行了改进，并提出了一种减轻寄存器压力的优化方法。本算法在开放源码编译器 ORC(Open Research Compiler 是 IA-64开放源码编译器的名称)上得到了实现。在 IA-64上运行的实验结果证明，该算法对于执行频率很高，而且寄存器压力大的自递归函数有很明显的优化效果。

关键词 寄存器栈引擎，栈寄存器，溢出处理，寄存器栈溢出

Optimization Technology for Stacked Register Allocation

LIU Yang ZHANG Zhao-Qing

(Institute of Computing Technology, Chinese Academy of Science, Beijing 100080)

Abstract Stack register plays an important role in reducing memory access at call sites in procedures. But stack registers are not cost free, they have very high cost when overflow happens. In order to solve this problem, this paper proposes an effective algorithm to manage excessive usage of stack registers in self recursive functions, do modification to traditional simplification, and find a optimization method to alleviate register pressure. This algorithm is implemented in ORC[2] compiler. Experiments show that this algorithm is very useful in performance improvement of programs having self-recursive function with high execution frequency and great register pressure.

Keywords Register stack engine, Stack registers, Stack overflow, Spill

1 引言

1.1 相关工作

寄存器分配在现代编译器中对性能有着较大的影响，已出现了很多关于寄存器分配的算法[1~5]。Chaitin[1]首先提出了图着色的寄存器分配方法。在 Chaitin 的算法中，它被划分为以下几个步骤：干涉图构造，简化，图着色和溢出处理。构造的干涉图首先被简化，所有的活跃区间被分成两类：有限制的活跃区间和无限制的活跃区间。然后，对无限制的活跃区间进行着色，也就是分配寄存器；而有限制的活跃区间全部被溢出处理。Fred Chow[2]提出了基于优先级的寄存器分配算法。活跃区间按照它们的优先级排序，具有较高优先级的活跃区间将首先被分配寄存器。同时，活跃区间分裂也被引入到寄存器分配中来。对于有限制且不能分配到寄存器的活跃区间，可以把它分裂，使得活跃区间的其中一部分能够分配到寄存器。Brigg[3,4]通过延迟活跃区间的溢出处理对该算法进行了改进。因为无限制的活跃区间可以被分配到寄存器是充分但不是必要条件，所以，即使是有限制的活跃区间，也是有机会分配到寄存器的。因此，Brigg 并不是溢出处理所有的限制活跃区间，试着给它们分配寄存器；如果不能找到合适的寄存器，那么就溢出处理这些活跃区间。Lueh[5]的文章讨论了寄存器分配中的另一个重要问题，即被调用者保存和调用者保存寄存器的调用代价。在他的文章中，活跃区间有三种选择：使用一个被调用者保存寄存器，使用一个调用者保存寄存器或者不使用寄存器，放在内存中。该文指出在某些情况下，变量被分配到寄存器不如被放置在内存中。

在过去的寄存器分配算法中，因为还没有栈寄存器，所以，也无需考虑栈寄存器溢出的代价。但是，当栈寄存器溢出的代价存在时，这些算法就不能有效地处理这个问题了。第1.2节将对寄存器栈进行描述，1.3节对有寄存器栈存在时，过去的算法在处理这种情况时存在的问题进行讨论。

1.2 寄存器栈

为了减少函数调用导致的访存操作，过去有很多研究工作针对这个问题进行了研究[7,8]，硬件实现的寄存器窗口是一种方法。Intel 的 IA-64体系结构实现了这种寄存器栈，整个寄存器可以被划分为静态寄存器和栈式寄存器两个部分。IA-64共有96个整型栈寄存器，当发生函数调用时，寄存器栈指针会自动移动，而在函数执行完毕，栈又会自动退回[6]。在被调用函数的寄存器使用量超过剩余寄存器数量时(发生栈溢出)，部分栈寄存器会被寄存器栈引擎(Register Stack Engine, RSE)保存到内存中，需要时再从内存加载。这个过程完全由硬件自动完成，不需要软件的干涉。但这是有开销的，我们称这种为有代价的栈寄存器。每个寄存器帧分为三个部分：本地(Local)，输入(Input)和输出(Output)。调用函数的输出和被调用函数的输入寄存器是重叠的。图1表示当函数调用 A->B->C 发生时寄存器栈的情况。

2 问题的分析

从1.2节我们可以看到，通常情况下，栈式寄存器的使用是几乎没有代价的。但情况并非总是如此。用 ORC 编译 Perlbmk 得到执行代码，利用 Pfmon 1.1版对程序执行状况进行测量，我们发现 RSE 代价在整个程序执行时间中占了很大比例(见图6)。ORC 的全局寄存器分配是基于 Briggs 和 Fred 的寄存器分配算法。

*)本文的项目得到国家自然科学基金(69933020)，863项目(2001AA111061)和 Intel 公司的资助。

维普资讯 http://www.cqvip.com

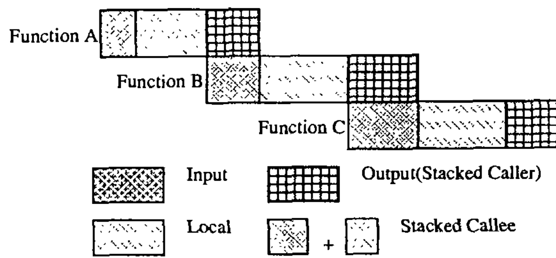

图1　寄存器栈帧的移动

在 ORC 编译器中，为了进一步减轻寄存器压力，增加函数调用时寄存器的重叠部分，提出了栈调用者保存寄存器(Stacked Caller Register)和栈被调用者保存寄存器(Stacked Callee Register)的概念。在图1中，我们可以看到寄存器窗口的输出部分又叫做栈调用者保存寄存器，而本地和输入寄存器合在一起叫做栈被调用者保存寄存器。这是因为，对于某些活跃区间不跨越调用点时，调用者不需要去保存它；一旦活跃区间跨越调用点，使用这些寄存器就需要调用者对它们进行保存，所以叫做栈调用者保存寄存器。而栈被调用者保存寄存器实际上是既不需要调用者，也不需要被调用者保存的，它是通过栈指针的自动移动来实现寄存器的不被重用。这样，在做寄存器分配时，对于不跨越调用点的活跃区间，尽量分配给它们输出部分的寄存器，使得调用函数的栈帧输出部分与被调用函数的栈帧输入部分尽可能重叠，减少寄存器栈指针移动的位移量。而寄存器栈帧的输入和本地部分都是不需要被调用者保存的，可以分配给跨越调用点的活跃区间。

但即使采用了这些算法，栈寄存器溢出仍然给程序的执行性能带来了很大的负面影响。我们可以看到，在有代价的栈寄存器存在时，传统的全局寄存器分配算法会遇到以下问题。

1. 在简化阶段，所有的有限制活跃区间比所有的无限制活跃区间具有更高的优先级。这是因为，没有活跃区间冲突限制的活跃区间总是可以被分配到一个寄存器，因此它们总是在有冲突限制的活跃区间分配寄存器以后才被考虑。即使反馈信息显示这些有限制的活跃区间根本就没有被执行。这在没有栈寄存器的情况下是合理的，但是在有栈寄存器的情况下，这就不是很合理的选择。我们通过一个简单的例子来阐述这个问题。

假设有两个活跃区间 *l1* 和 *l2*，活跃区间 *l1* 是无限制的，而 *l2* 是一个有限制并且跨越函数调用的活跃区间。为了简化问题的讨论，不失一般性，我们假设仅有两个寄存器：*r1* 是一个通用静态寄存器而 *r2* 是一个栈式寄存器。因此，*l2* 被优先分配一个寄存器 *r1*，而 *l1* 只能使用栈寄存器 *r2*。假设 *r2* 的使用代价非常高，我们必须减少可用的栈寄存器数目。因此，*l2* 只能放在内存中了。如果 *l2* 的优先级远远高于 *l1*，或者在极端情况下，*l1* 的执行频率甚至是0。很明显，把 *l2* 放在寄存器里要比把 *l1* 放在寄存器里更有利。

2. 在活跃区间分裂阶段，一旦一个活跃区间被分裂，如果其优先级大于0，它就将立即被分配到一个寄存器。这样做也是不公平的，被分裂的活跃区间并不总是具有比其他活跃区间更高的优先级。这是不公平的，分裂的活跃区间并不总是有比其他仍没有分配到寄存器的活跃区间具有更高的优先级。

3. 对于所有跨越调用的活跃区间，首先选择一个被调用者保存寄存器。但是对于可重计算的变量，一个调用者保存寄存器并不会增加任何开销，甚至在仅跨越一个调用时代价更小。

4. 并非所有变量在寄存器没有代价的情况下就应该获得一个寄存器，有时我们发现对于某些变量，不分配给它寄存器会节省寄存器并使得操作减少。这将在下一节做详细说明。

5. 最明显的 RSE 问题发生在自递归函数里。假设函数 f 是一个自递归函数，使用 r 个寄存器，那么在调用次数为 n，使得 n * r 大于96时，递归的第 n 次以后的调用都会栈溢出 r 个寄存器，RSE 就会被启动并且所有用过的栈寄存器会被保存到内存里。当函数返回时，保存在后备存储区(Backing Store)的值又被加载到栈寄存器里。当函数的寄存器压力很大(r 很大)，而且递归深度很深时，RSE 的代价就变得十分昂贵了。

3　解决优化的关键技术

我们找出了一个在自递归函数中使用栈式寄存器的启发式算法，改进了需要处理栈寄存器溢出时的简化过程；并提出了一个新的方法来优化寄存器分配，它可以降低寄存器压力和把高代价的存储(Store)或者加载(Load)操作从执行频率高的基本块移动到执行频率低的基本块里去。

3.1　自递归函数中无代价寄存器使用量的估算

在实验中，我们发现，尽管自递归函数调用自身多次，但是并非每次递归调用都会导致栈溢出。例如，在 Perlbmk 的 Regmatch 函数，它是一个自递归函数，并且调用次数非常频繁，但是通过插装，发现它的平均调用栈深度不深，大约为5次。它自身调用自己几次以后，就会退回，但是其调用者会频繁调用它。因此，栈寄存器的使用量控制在一定范围内时，每次递归调用都不会引起栈寄存器溢出。在调用图中，找出循环，并且通过对函数进行插装得到它平均每次调用自身的次数，使用如下公式计算没有代价的寄存器使用量预算：

预算＝96/平均调用栈深度

在预算范围内使用的寄存器由于不会导致栈溢出，可以认为是没有代价的，而超过允许范围的寄存器使用则是有代价的。下一节将阐述如果需要使用有代价的栈寄存器时需要考虑的问题。

3.2　在寄存器有代价时的简化算法

在前面我们说明了在不考虑有代价的栈寄存器时，简化存在的问题。我们的方法是，在简化时就认为超过预算的所有栈寄存器都是有代价的，因此所有需要使用栈寄存器的活跃区间都被认为是有限制的，最后还没有分配到寄存器的这些活跃区间最后按照优先级来分配栈寄存器。优先级高的先分配到一个寄存器，而优先级低的则不能被分配到栈寄存器。对于这些需要有代价寄存器的活跃区间，考虑每个活跃区间如果使用栈寄存器时的栈溢出代价和如果栈溢出该活跃区间的代价，并对它们进行比较，如果溢出代价高则不使用寄存器，反之使用寄存器。从实验结果表明每个栈寄存器溢出的代价相当于1.5个时钟周期左右。因此，如果栈溢出的代价高于1.5个时钟周期，使用寄存器就是合算的；否则，不使用反而能使性能较好。

3.3　考虑访存代价的寄存器分配

如果我们不给某些临时变量分配寄存器，可以对某些操作进行优化从而获得更好的性能。这是一个减轻寄存器压力的机会，甚至可以消除一些不必要的操作，从而提高性能。下面举例说明。

有下边的控制流图片段(如图2所示)。

维普资讯 http://www.cqvip.com

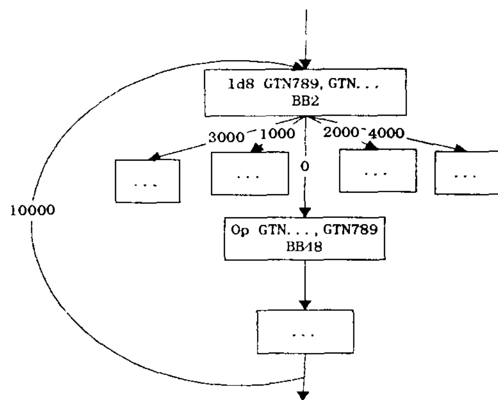

图2 控制流图片段

变量 GTN789是可重计算的，它在基本块 BB2中有一个定义，在基本块 BB48中有一个引用。基本块 BB2是一个 switch 语句的入口基本块，有很多后继基本块。假设2的执行频率很高，而 BB48的执行频率很低，不失一般性，这里假定其执行频率为0。如果变量 GTN789分配到一个寄存器，那么在 BB2中的 Ld8操作就不能被优化掉。但是，如果变量 GTN789被溢出处理，那么在 BB2中 Ld8操作就会被优化掉。这是因为，在 GTN789没有分配到寄存器的情况下，变量 GTN789的活跃区间在基本块边界上被分裂，BB2中的 Ld8操作变成一个没有引用的定义，而 BB48中因为变量 GTN789可以被重计算而重新加载它。这样，BB2中的加载将被删除，从而使得总的访存频率降低。这个方法节省了一个寄存器和访存操作。

如上所述，对于那些可以被重计算的临时变量的活跃区间，计算它的总定义频率和总引用频率。如果其总的定义频率大于总的引用频率，那么就不给该临时变量分配寄存器。这样，基于前边的描述，一个执行频率很高的定义操作就可以被优化掉，到使用时再从内存加载，反而减少了执行时钟周期。详细算法如图3所示。

```
def_freq=0;
use_freq=0
for all bb x in variable y's live range {
  if y is rematerializable {
    if y has a definition in x
      def_freq=def_freq+x→Freq();
    if y has a use in x
      use_freq=use_freq+x→Freq();
  }
  if (def_freq>use_freq)spill(y);
}
```

图3 考虑访存代价的寄存器分配优化算法

4 实验结果

图4的横轴表示栈寄存器溢出的个数，竖轴表示在栈溢出这么多寄存器时的 RSE 需要的时钟周期数。可以看到，随着栈溢出个数的增加，栈溢出的代价呈线性增加。

Spec2000中，有比较明显 RSE 开销的是 Perlbmk。其中 Perlbmk 的 RSE 开销尤其严重，达到了整个运行时间的30%左右。图5表示在没有我们的启发式算法和有我们的启发式算法情况下性能的比较。图中的实验结果是由 pfmon1.1版本测试得到的。右边的条块表示没有做提高时的运行时间，左边表示在进行提高后的性能。从图中可以看到，在进行提高前，RSE 在整个执行时间中占有很高的比例。采用本文描述的算法进行优化以后，Perlbmk 的 RSE 几乎完全消失了，总的执行时间大大减少。

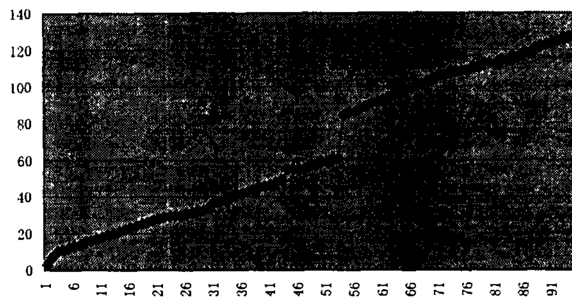

图4 N 个栈寄存器溢出时的代价(时钟周期)

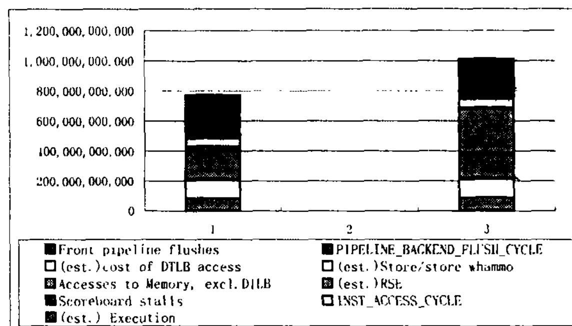

图5 实验结果

结论 栈寄存器在使用数量过多时会造成寄存器栈溢出，这样的代价是很高的，在 Spec2000的 Perlbmk 中这种情况尤其明显。本文提出了一种有效消除递归函数中栈溢出的算法，针对栈寄存器存在代价对简化算法进行了修改，并且通过一些优化来减轻寄存器压力，对某些具有很高 RSE 开销的程序大幅度提高了性能。未来我们可以做更多的实验来得到一个更精确和更通用的栈寄存器分配模型。该模型应该不仅针对于自递归的函数，而且对于因调用栈过深等导致的寄存器栈溢出也能很好地处理。对于大型的应用程序，以及调用栈深度很深的 Java 程序，这样的情况会更加普遍。

参考文献

1 Chaitin G. Register allocation and spilling via graph coloring. In: Proc. of the SIGPLAN 82 Symposium on Compiler Construction (Boston, Mass., June 1982). ACM, New York, 1982. 98~105

2 Chow F C, Hennessy J L. Register allocation by priority-based coloring. In: Proc. of the SIGPLAN 84 Symposium on Compiler Construction (Montreal, June 1984). ACM, NewYork, 1984. 222~232

3 Briggs P. Register Allocation via Graph Coloring: [PhD thesis]. Rice University

4 Briggs P, Cooper K, Torczon L. Improvements to graph coloring register allocation. ACM Transactions on Programming Languages and Systems, 1994, 16(3): 428~455

5 Lueh G, Gross T. Call-cost directed register allocation. In: Proc. ACM SIGPLAN '97 Conf. on Prog. Language Design and Implementation, ACM, ACM Transactions on Programming Languages and Systems. 1997. 296~307

6 Intel IA-64 System Architechture. Intel Company

7 Steenkiste P A, Henessy J L. A simple interprocedural register allocation algorithm and its effectiveness for LISP. Transactions on Programming Languages and Systems, Jan. 1989. 1~30

8 Wall D W. Global Register Allocation at Link Time. In Proc. of the SIGPLAN '86 Symposium on Compiler Construction, New York, 1986