第 27 卷 第 9 期
2004 年 9 月
计 算 机 学 报
CHINESE JOURNAL OF COMPUTERS
Vol. 27 No. 9
Sept. 2004

优化 RSE 开销的过程间栈寄存器分配

刘 旸 张兆庆
(中国科学院计算技术研究所 北京 100080)
(中国科学院研究生院 北京 100080)

摘 要 安腾®处理器引入了硬件控制的寄存器栈,寄存器栈引擎能够自动地改变寄存器栈帧指针,对栈寄存器进行保存和恢复,从而有效地减少跨越过程调用时的寄存器值的保存和重新载入.每个过程使用的栈寄存器数量可以通过 alloc 指令显式地指定.通常的过程内寄存器分配方法给过程分配最大需要数量的栈寄存器.但过多的栈寄存器使用会引起寄存器栈溢出/载入.如果频繁出现这样的寄存器栈溢出/载入,将严重影响程序执行性能.该文提出了一种创新的算法,能够有效地减少 RSE 代价.该算法已经在开放源码编译器 ORC 中得到了实现.实验表明,SpecINT2000 在使用该算法后性能普遍提高,perlbmk 的性能提高了 14%,而 crafty 也有 3.2%的性能提高.

关键词 寄存器栈;寄存器栈引擎;寄存器栈溢出/载入
中图法分类号 TP302

Inter-Procedural Register Allocation for RSE Optimization

LIU Yang ZHANG Zhao-Qing
(*Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100080)
(*Graduate School of Chinese Academy of Sciences*, *Beijing* 100080)

Abstract In Itanium® architecture, a hardware managed register stack is introduced, register stack engine (RSE) can change the register stack frame pointers and spill/fill registers automatically. This mechanism can reduce load/store operations of register across call sites efficiently. The number of stacked registers used by a procedure could be specified by alloc instruction explicitly. Traditional intra-procedural register allocation algorithm will allocate max stacked registers required by a procedure but no more than the total number of stack registers. But high stack register pressure will lead to frequent register stack spill/fill. If this event happens frequently, the performance will be seriously harmed. This paper proposes an innovative algorithm, which could reduce the RSE cost efficiently. This algorithm is already implemented in ORC. Experimental results show that the performance is improved obviously when this algorithm is applied, especially for perlbmk, it has 14% performance improvement and crafty also has 3.2 % performance improvement.

Keywords register stack; register stack engine; register stack spill/fill

1 引 言

寄存器分配问题,就是从需要使用寄存器的变量中选择出一个子集,放置到有限的寄存器中,从而达到尽量减少内存和寄存器间的传送开销,也就是使程序在实际运行中执行的加载/存储操作减低到最少的目标.研究者们已经对这个问题进行了深入

收稿日期:2002-12-27;修改稿收到日期:2004-05-28.本课题得到国家自然科学基金(69933020)和英特尔公司资助.**刘 旸**,男,1973 年生,博士研究生,主要研究方向为指令级并行、寄存器分配、多线程和低功耗编译技术.E-mail: ly@ict.ac.cn.**张兆庆**,女,1938 年生,研究员,博士生导师,主要研究方向为先进编译技术及相关编译环境.

广泛的研究,并且提出了很多有效的算法[1~5].因为寄存器数量是有限的,所以对于某些没有分配到寄存器的变量,只能在使用它时把它从内存取出,执行完操作以后又把它重新存到内存里.这样的处理叫做溢出处理(spill).

除了因为过程内寄存器分配时某些变量不能得到寄存器而导致的溢出处理以外,还有很多其他原因引起的加载和存储操作.例如,在变量 x 的活跃区间跨越过程调用点时,被调用过程可能会使用到变量 x 使用的寄存器 r.因此,在过程被调用之前,需要对跨越调用点的活跃区间使用的寄存器进行保存;而过程返回以后,又需要把这些寄存器从内存重新载入.研究者们针对该问题进行了很多研究,并提出了一些软件的解决方法[6,7].

安腾®(Itanium®)处理器[8]选择了一种硬件实现方法来解决这个问题.在安腾®处理器中,共有128 个通用寄存器,它们被划分为两个部分:r0～r31 是静态寄存器,它们对于所有过程都是可见的;r32～r127 是栈寄存器,除了用于参数传递的寄存器以外,它们仅对某个特定的过程是可见的.仅对某个特定的过程可见的寄存器一起形成一个寄存器栈帧,栈帧的大小是由编译器显式地指定的.发生过程调用时,当前栈指针通过硬件自动移动到为被调用过程新分配的寄存器栈帧;过程返回时,当前栈指针又回到调用者栈帧上.这种机制避免了在跨越调用点时对于寄存器的保存和加载.寄存器栈引擎(Register Stack Engine,RSE)是一个处理器状态机,当被调用过程要求的寄存器栈帧大于现有的可用寄存器数目时,它负责把栈中的部分寄存器拷贝到(或者从后备存储区载入)后备存储区(backing store)中,给被调用过程腾出足够的可用栈寄存器;当函数返回,它又把溢出的栈寄存器从后备存储区载入到寄存器栈中.当发生栈寄存器溢出/载入事件时,程序被阻塞,直到这个过程完成为止.

虽然这个机制大大地减少了跨越过程调用时的加载/存储操作,但是寄存器栈溢出/载入(register stack spill/fill)事件所造成的 RSE 开销也是不容忽视的.实验证明,RSE 开销在有些程序中严重地影响了执行性能.使用 pfmon 1.1 版对 Spec2000INT 的可执行代码进行运行时测试,发现 perlbmk 的执行时间中 RSE 开销竟然占据了 23%,而 crafty 的 RSE 代价也高达 4.8%.快速发展的 CPU 处理速度会让 RSE 开销在整个执行时间中所占的比例越来越大,使得该问题更加严重.

本文提出了一种全新的解决方法.在第一遍编译时,使用通常的寄存器分配,得到每个过程的寄存器使用情况反馈信息.在第二遍编译时的过程间优化阶段,读入反馈信息,并基于过程调用图,给每个过程分配一定的栈寄存器预算使用数量.在后端优化的全局寄存器分配阶段,按照寄存器预算使用数量进行寄存器分配,每个过程使用的栈寄存器最多不得超过预算使用数量.这个算法已经在开放研究编译器 ORC 上实现,实验证明该算法对于控制 RSE 开销和提高程序执行性能十分有效,特别是对于 perlbmk,性能提高了将近 14%,crafty 也有 3.2%的性能提高.

本文第 2 节对寄存器栈的背景知识做了简单的介绍;在第 3 节,我们讨论了国际相关工作;第 4 节分析了实例,对解决的方法进行了讨论;第 5 节描述了算法的详细实现;第 6 节给出了实验数据并对实验数据进行了分析;第 7 节总结了本文并对未来的工作提出了展望.

2　寄存器栈的背景介绍

2.1　寄存器栈和它的动作

栈寄存器对于每个过程来说是局部的.仅对某个给定过程可见的栈寄存器一起组成一个寄存器栈帧.每个寄存器栈帧包括 4 个部分:(1)输入寄存器,input registers;(2)本地寄存器,local registers;(3)输出寄存器,output registers;(4)旋转寄存器,rotating registers.旋转寄存器是为了方便对程序进行软流水优化处理.输入和旋转寄存器都包含在本地寄存器里.寄存器栈帧的大小是本地寄存器+输出寄存器.在 Itanium®处理器中,每个寄存器栈帧最多可以有 96 个栈寄存器.寄存器栈帧的大小可以由编译器通过 alloc 指令显式地指定:

alloc〈target_reg〉 = ar.pfs in,local,out,rot.

寄存器栈引擎负责把寄存器栈帧映射到物理寄存器组的栈寄存器上,这个过程对于编译器是透明的.当一个过程被调用时,栈寄存器被重命名,从而调用者寄存器栈帧中输出寄存器的第一个变成被调用者寄存器栈帧中的 r32.参数通过调用者寄存器栈帧的输出部分传递到被调用者.被调用过程从它的寄存器栈帧输入区读取参数.当被调用者返回时,寄存器再次重命名为调用者的寄存器栈帧.这个机制使得调用者的寄存器栈帧可以被保存在寄存器中而不是储存到内存里去.

这里我们用一个例子来描述寄存器栈的行为.在图 1 中,过程 foo()有一个从 r32 开始的寄存器栈帧,而过程 bar()的寄存器栈帧也是从 r32 开始的.执行代码只能看到和一个给定过程相关联的寄存器栈帧.

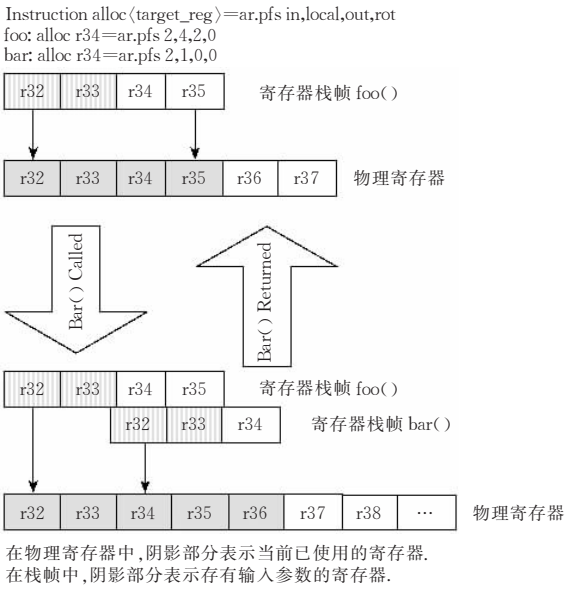


图 1 寄存器栈帧和它的行为

2.2 寄存器栈的载入和溢出

当一个过程被调用时,RSE 会为它分配一个寄存器栈帧,最多有 96 个寄存器.如果没有足够未使用的栈寄存器,它就会溢出一部分已使用的栈寄存器来腾出足够多的空间.溢出的栈寄存器值被保存到一块叫做后备存储区的内存区域中.当一个过程返回并当前过程的寄存器栈帧已经被溢出到后备存储区时,RSE 需要把这个寄存器栈帧从后备存储区重新取出,放到栈寄存器中.这个过程由 RSE 自动执行,对编译器是透明的.

这里用一个例子来描述寄存器载入和溢出的行为.假设有三个过程,foo(),bar()和 par().Foo()调用 bar()而 bar()调用 par().如果 foo()和 bar()用完了所有 96 个栈寄存器,那么当 par()被调用时,foo()的寄存器栈帧将被 RSE 溢出到后备存储区.当 bar()返回时,foo()的寄存器栈帧又被 RSE 从后备存储区载入到栈寄存器里.这个过程会阻塞程序的执行,因此耗费大量的时间.

3 国际相关工作

Douillet 等[9]提出了一种通过在程序控制流的不同路径上插入 alloc 指令来减少 RSE 开销的方法.因为在程序控制流图的不同路径上,需要使用的寄存器数量是不同的,因此,该算法尝试在需要寄存器较少的路径上插入 alloc 指令来改变寄存器栈帧的大小,从而达到减少调用点的 RSE 溢出的目标.但他们的实验数据很不理想,因为 alloc 指令本身也有不少开销,结果导致整体性能下降.David 等[10]对 RSE 开销给出一个量化的估算,并且提出了一种硬件解决方法,该算法本质上是 Alban 等提出算法的一种硬件实现.在模拟器上的实验证明减少了很多 RSE 动作.但是硬件实现的代价很高,并且没有实际的机器运行性能比较.Pro64 编译器也在消除 RSE 开销这个问题上进行了一些工作.在 Pro64 编译器中,栈寄存器被划分为两个部分:栈调用者寄存器和栈被调用者寄存器(stacked caller registers and stacked called registers).栈调用者寄存器指的是寄存器栈帧的输出部分,而栈被调用者寄存器指的是寄存器栈帧的本地部分.如果一个活跃区间跨越过程调用点,那么它就被分配到栈被调用者寄存器,否则被分配到栈调用者寄存器.这样,可以尽量地扩大输出部分,增加调用者和被调用者寄存器栈帧的重叠部分.但是,即使采取了这样的措施,用 Pro64 编译得到 SpecINT2000 的可执行代码运行时仍然有严重的 RSE 问题,perlbmk 的 RSE 仍然达到了 24%.

4 问题的分析

4.1 RSE 开销的测量

假设有一条动态的调用路径 $p_1 p_2 \cdots p_{i-1} p_i$, $p_i \cdots p_n$.当过程 p_i 被调用时,过程 p_j 后在寄存器栈里的 z 个栈寄存器被溢出到后备存储区中.那么,当过程调用结束,并且逐层返回到过程 p_j 时,被溢出的 z 个栈寄存器又会被载入到寄存器栈里.通常情况下,只要程序执行不在调用路径上中途退出,栈溢出(stack spill)寄存器的个数和栈载入(stack fill)寄存器的个数应该是相等的.因此,栈寄存器溢出和栈寄存器载入可以合起来看作一次寄存器栈溢出/载入.

只有被调用过程需要的栈寄存器数量大于当前寄存器栈中可用栈寄存器数量时,才会发生寄存器栈溢出/载入.在调用路径 $p_1 p_2 \cdots p_{i-1} p_i$, $p_i \cdots p_n$.上,设过程 p_{i-1} 被调用并分配寄存器栈帧以后,寄存器栈还剩余 x 个未使用的寄存器,而被调用过程 p_i 需要 y 个栈寄存器,如果 $y>x$,那么至少需要溢出 $z=y-x$ 个栈寄存器.利用 pfmon1.1 版,我们研

究了 Itanium® 处理器上的 RSE 开销和需要溢出/载入的栈寄存器个数 z 之间的关系. 用于实验的程序包括 4 个过程,它们的调用关系是:A 调用 B,B 调用 C,C 调用 D. A,B,C 总共分配了 96 个栈寄存器. 这样,在 D 被调用的时候,剩余的可用栈寄存器为 0. 只要过程 D 需要多于 0 个的栈寄存器,就会发生寄存器栈溢出/载入. 实验中 D 分配的栈寄存器栈帧大小从 1 递增到 96,每次分配的栈帧大小增加 1. 用 pfmon 测量程序当过程 D 分配不同数量栈寄存器时运行的 RSE 开销,结果如图 2 所示. 图中横轴表示过程 D 需要的栈寄存器数量,即需要溢出/载入的栈寄存器个数;竖轴表示时钟周期数. 从图上我们看到 RSE 开销随寄存器栈溢出/载入寄存器的数量增加而线性增加. 这与栈寄存器整个帧溢出的情况并不完全符合. 每个栈寄存器溢出/载入引起的 RSE 代价大约是 1 个时钟周期,除了从溢出 56 个到溢出 57 个栈寄存器时有一个较大的 RSE 开销增加. 假设 S_{latency} 为启动时间, z_i 是过程 p_i 被调用时需要溢出的栈寄存器个数, $diff$ 是根据实验测量得到的一个栈寄存器溢出花费的时间开销,那么我们可以用以下公式来计算 RSE 开销:

$$Dynamic_RSE_Cycles = z_i \times diff + S_{\text{latency}} \quad (1)$$

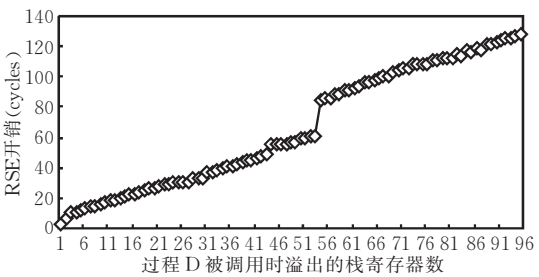


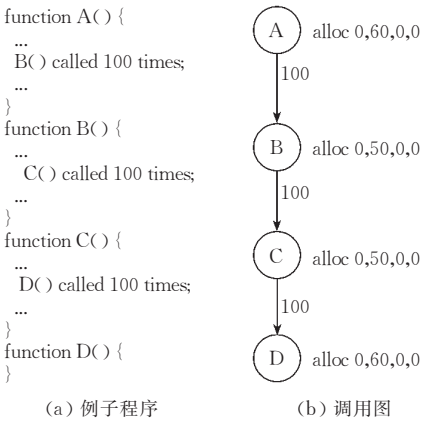
图 2　RSE 开销和溢出的栈寄存器数量之间的关系

4.2　实例分析

通常的过程内寄存器分配会分给每个过程它所需要的最大栈寄存器数量而不考虑 RSE 开销. 因为过程内寄存器分配看不到调用路径上其他过程的栈寄存器使用情况,所以它很难确定一个过程使用多少栈寄存器不会导致寄存器栈溢出/载入.

如果在进行寄存器分配时能够知道调用路径上其他过程的栈寄存器使用数量,就可以知道使用多少栈寄存器会导致寄存器栈溢出/载入. 因此,我们想到使用调用图来帮助寄存器分配. 这里给出一个简单的例子来说明它是如何工作的. 假设编译的目标机器是 Itanium 处理器,总共有 96 个栈寄存器. 例子程序和它的调用图如图 3 所示.

过程旁的指令是对该过程进行过程内寄存器分配时的栈帧分配指令. 在这个例子中,每两个过程间调用边的频率是相等的. 从 4.1 节的实验我们知道多溢出一个栈寄存器,RSE 开销就会增加大约 1 个时钟周期. 图 3 中,过程 B,C,D 每次被调用时分别会溢出 14,50,60 个栈寄存器,根据公式(1),总的 RSE 开销为 12400 个时钟周期. 假设除去 RSE 开销,程序运行时的访存开销是 $t_{\text{spill_cost}}$,那么总的访存开销是 $t_{\text{spill_cost}}+12400$.



(a) 例子程序　　(b) 调用图

图 3　例子和调用图

由前述可知,如果调用路径上使用的栈寄存器数量不超过 96,就不会发生寄存器栈溢出/载入,从而没有 RSE 开销. 因此,为了消除程序的 RSE 开销,容易得到一种简单的算法:

算法 1.　从调用图的根节点开始,按照图中的调用路径进行栈寄存器分配,到用完 96 个栈寄存器时停止.

对图 3 的程序应用算法 1,过程 A 被分配到 60 个栈寄存器,过程 B 得到 36 个栈寄存器. 这样,96 个栈寄存器都被用完,过程 C 和 D 不能得到栈寄存器. 相比于过程内寄存器分配,过程 B,C 和 D 分别少使用了 14,50 和 60 个栈寄存器. 算法 1 消除了所有的 RSE 开销,但是减少栈寄存器的使用必然会导致更多活跃区间被溢出,从而导致访存开销的增加. 为了知道算法 1 是否能够减少总的访存开销,从而提高性能,我们需要知道少使用一个栈寄存器,会增加多少访存操作时钟周期数. 反过来说,就是多使用一个栈寄存器,可以减少多少访存开销.

栈溢出一个寄存器就会导致相同的 RSE 开销,但是多用一个栈寄存器会减少的访存开销通常是不一样的. 这是因为使用该寄存器的活跃区间各不相同,从而使用该栈寄存器可以减少的访存代价也不一样. 为了讨论简单起见,我们假设对于某个特定的过程,多用一个栈寄存器减少的访存开销是一样的.

图 4 给出了 4 个过程中使用的栈寄存器数量和减少的 load/store 操作时钟周期数之间的关系. 图上横轴表示过程里使用的栈寄存器个数,竖轴表示由此减少的 load/store 操作时钟周期数. 在过程 A,B,C,D 中,每多使用一个栈寄存器分别可以减少 200,50,80 和 400 个 load/store 操作时钟周期.

有了以上信息,我们就可以计算对程序应用算法 1 后的性能变化. 因为过程 B,C,D 分别少使用了 14,50 和 60 个栈寄存器,因此会增加总共 28700 个时钟周期的访存操作,从而总的访存开销为 $t_{spill_cost}+28700-12400=t_{spill_cost}+16300$. 相比于最初的总访存开销,性能反而下降了.

很显然,这不是所希望的结果. 我们的目标是减少总的访存开销,从而提高程序性能. 一个栈寄存器的使用是否对性能有提高取决于使用它能够减少总的访存开销,即使用它可以减少的访存操作时钟周期数是否大于它会引起的 RSE 开销. 结合调用图和图 4 提供的信息,算法 1 可以被改进如下:如果一个栈寄存器的使用会引起寄存器栈溢出/载入,但是使用它可以减少的访存操作时钟周期数大于 RSE 开销,则使用该寄存器. 程序应用新的算法,过程 A,B,C 和 D 将各使用 60,36,0,60 个栈寄存器. 这样,只有过程 D 被调用时会引起 60 个栈寄存器溢出,从而总的 RSE 代价为 6000;因为过程 B 和过程 C 分别少使用了 14 和 50 个栈寄存器,访存操作增加了 4700 个时钟周期. 总的访存开销为 $t_{spill_cost}+10700$ 个时钟周期,少于在过程内寄存器分配时的 RSE 代价.

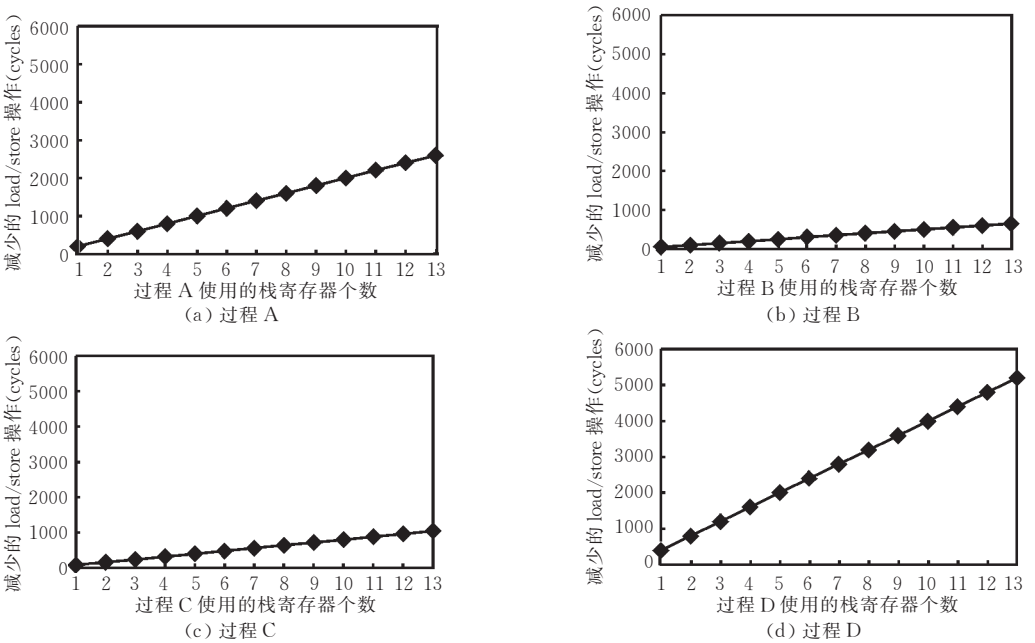


图 4 过程使用的栈寄存器数和由此减少的 load/store 操作时钟周期数之间的关系

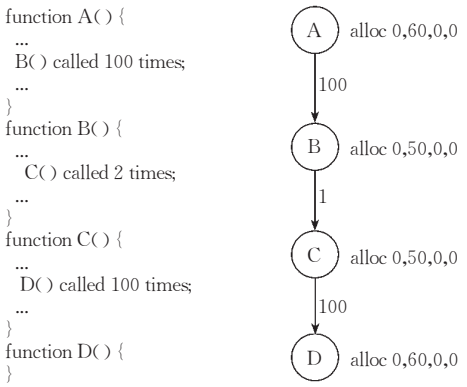
进一步观察程序,可以发现过程 B 中每个栈寄存器的使用只能减少 50 个时钟周期,而过程 D 使用一个栈寄存器却可以减少 400 个时钟周期,但是过程 B 优先使用没有寄存器栈溢出/载入代价的栈寄存器. 很明显,这样不是最好的情况. 如果我们优先把栈寄存器分给过程 D,性能可以进一步得到提高. 对所有过程使用的栈寄存器 r_{ij} 按照 reduced_cycles(r_{ij})进行排序,这里 r_{ij} 表示在过程 p_i 中使用的第 j 个栈寄存器,得到如下序列:

$$r_{D1}r_{D2}\cdots r_{D60}r_{A1}r_{A2}\cdots r_{A60}r_{C1}r_{C2}\cdots r_{C50}r_{B1}r_{B2}\cdots r_{B50}.$$

序列中前 96 个栈寄存器的使用是没有代价的;顺序在 96 以后的栈寄存器 r_{ij} 仍然按照算法 2 中的方法确定是否可以使用. 这样,4 个过程 A,B,C,D 各分配到 60,0,0,60 个栈寄存器. RSE 开销为 2400 个时钟周期,而 B 和 C 因为相比于过程内寄存器分配时各少用了 50 个栈寄存器而增加了 6500 个时钟周期的 load/store 操作. 总访存代价为 $t_{memory_access}+8900$ 个时钟周期,性能比应用算法 1 时有了更多提高.

在上边的讨论中,热点中调用边的频率都是相同的,因此,计算多用一个栈寄存器增加的 RSE 开销比较简单. 但是在实际的应用程序里,调用边的频率通常是不同的. 这里(图 5)我们把过程 B 和过程 C 之间调用边的频率改为 1.

研究在哪些过程被调用时导致了大部分 RSE 开销容易发现,主要的 RSE 开销产生于两个热点:A→B 和 C→D,而 B→C 仅仅导致了 50 个时钟周期

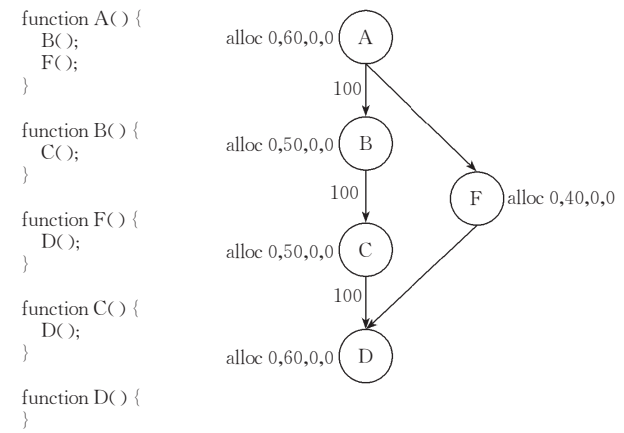


(a) 修改过的程序　　(b) 修改后程序的调用图

图 5　修改过的程序和它的调用图

的 RSE 开销. 如果对热点 A→B 和 C→D 分别应用算法 2,那么增加的 load/store 操作时钟周期和 RSE 代价总共是 1916 个时钟周期. 相比于过程内寄存器分配的结果,性能明显提高了. 下边我们来解释为什么分别对 A→B 和 C→D 应用算法 2 已经近似最大限度地减少了总访存开销. 由以上讨论可以看到,调用频率较低的边只会导致很少的 RSE 开销,如果选择出调用频率高的热点并对它们应用算法 2,可以近似地最大限度减少总访存开销. 我们得到算法 3:在调用图内找出调用边频率近似且有很大 RSE 开销的热点,并应用算法 2 分别对这些热点进行栈寄存器再分配.

上边的所有讨论中的调用图都是一条路径. 对图 3 中的程序略作修改,得到如图 6 所示的程序和调用图.



(a) 更复杂的程序　　(b) 多于一条路径的调用图

图 6　有分支的调用图及其对应的程序

如果我们使用通常的过程内寄存器分配算法,那么在调用路径 A→B→C→D 上会有 12400 个时钟周期. 而在另外一条路径 A→F→D 上仅有 64 个时钟周期的 RSE 开销,占 RSE 总开销的 0.5%左右. 应用算法 3,我们得到热点 A→B→C→D 并进行优化,可以使得总的 RSE 开销会减少到只有 8964 个时钟周期左右.

4.3　问题的描述

通过 4.2 节的讨论,我们看到通过限制过程内某些寄存器的使用,可以有效地提高性能. 从而,我们需要解决的是最大限度地使用栈寄存器,达到尽量减少总访存开销的目的. 这个问题可以被描述如下:

给定一个有调用边频率信息的调用图 G,栈寄存器上限 $bound$(在 Itanium® 处理器中这个上限是 96),平均每个栈寄存器的溢出代价 $stack_spill_cost(r)$ 等硬件相关信息以及过程内寄存器分配时使用的每个栈寄存器 r_{ij} 所能减少的加载/储存操作开销 $reduced_cost(r_{ij})$,给每个过程分配一定数量的栈寄存器,使得总的访问内存开销最小. 这里 r_{ij} 表示过程 p_i 中使用的第 j 个栈寄存器.

5　方法和实现

5.1　方　法

在前边的讨论中,看到在大多数程序中,RSE 的开销是由于某些寄存器使用量大并且调用频率高的热点引起的. 其它部分引起的 RSE 开销可能只占程序总 RSE 开销的很小部分. 如果能够很好地控制这些热点的 RSE 开销,最大限度地减少热点部分的访存开销,那么程序的整体性能应该会有明显的提高. 我们应用调用图和过程内寄存器分配时每个寄存器的使用可以减少的 load/store 时钟周期信息,对每个过程使用的栈寄存器数量进行了重新的分配,结果发现性能有很大的提高空间.

根据前边的信息,我们得到如下进行 RSE 优化的方法:首先在有调用次数信息的调用图上找出 RSE 开销较高的热点. 这里热点是一条调用路径的部分或者整条调用路径,热点上连接过程的调用边频率相近,并且有很高的 RSE 开销. 对所有热点内过程使用的栈寄存器 r_{ij} 按照它的使用可以减少的 load/store 操作时钟周期进行排序. 对于顺序不超过 $bound$ 的栈寄存器使用,因为没有溢出代价,所以都可以使用. 对于顺序超过 $bound$ 的栈寄存器,比较使用它引起的 RSE 开销是否小于减少的 load/store 操作时钟周期,如果这样,那么该寄存器可以使用. 这样,对于每个过程,我们得到一个使用栈寄存器的数量. 在后端进行全局寄存器分配时,就仅使用分给数量的栈寄存器.

5.2 实现框架

图 7 是 ORC 编译器在峰值模式进行编译时各个编译优化阶段的描述. 黑色方框中的编译阶段为 ORC 编译器原有的编译过程. 本文提出的算法包括两遍编译. 点划线所圈的过程是第一遍编译时加入的过程,虚线所圈的是第二遍编译时增加的过程,下边对这些过程按照顺序进行说明:

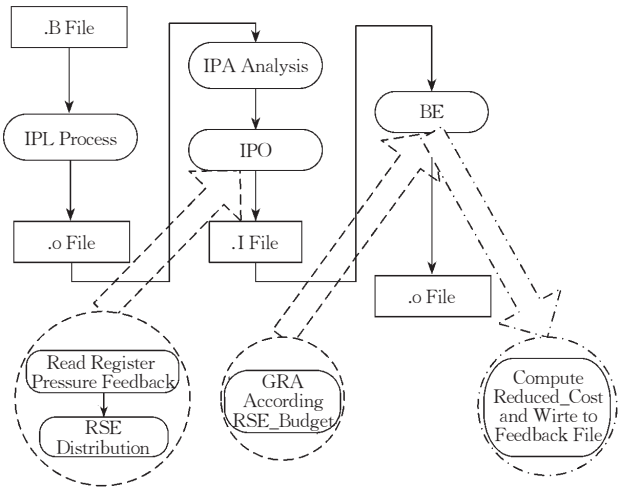


图 7 ORC 的编译过程和算法框架在其中的位置

在第一遍编译过程中,按照标准的过程内寄存器分配算法进行寄存器分配,然后统计每个栈寄存器的使用可以减少的时钟周期,把这些信息保存到反馈信息文件中. 对于一个栈寄存器 r,如果有 n 个活跃区间 l 使用 r,栈寄存器 r 的使用减少的时钟周期如下式所示:

$$reduced_cycles(r) = \sum_{i=1}^{n} spill_cost(l_i).$$

第二遍编译过程包括以下 3 步:

(1)在过程间优化的函数内联完成以后,读入寄存器使用情况的反馈信息.

(2)根据调用图,对于每个过程进行栈寄存器预算使用数量分配.

(3)在后端优化的全局寄存器分配阶段按照预算使用数量进行寄存器分配.

下一节将对栈寄存器预算分配算法进行详细的描述.

5.3 预算分配算法的实现

预算分配算法包括两个步骤:(1)找出导致大部分 RSE 开销的热路径;(2)在热路径上进行栈寄存器预算分配.

首先,在调用图中,找出一个权值最大且未处理过的节点 n 作为种子节点. 权值的计算方法如下:

$$weight(p_i) = called_freq(p_i) \times used_stack_reg_num(p_i).$$

从种子节点开始向后伸展,选择调用频率最大的后继节点 y,并且 y 是未处理过的且满足如下公式:

$$succ(x,y) = \frac{call_freq(x \rightarrow y)}{called_freq(x)} > T$$

且

$$\frac{call_freq(x)}{called_freq(seed)} > T'.$$

类似地定义 $pred(x,y)$,并从种子节点开始向前伸展热点. T 和 T' 是由编译器定义的阈值.

然后,对于在选出路径上使用的所有栈寄存器 r_{ij},按照 $reduced_cycles(r)$ 进行排序. 这里 r_{ij} 表示在过程 p_i 中使用的第 j 个栈寄存器. 逐个检查排序后的序列中每个栈寄存器 r_{ij}. 栈寄存器的数目是一个硬件的特征,这里我们把它定义为 $bound$. 在 Itanium 处理器中,它有 96 个栈寄存器,其 $bound$ 的值就是 96.

如果 r_{ij} 在排序序列中的顺序号 $\leq bound$ 且 p_i 不是一个自递归过程,那么可以使用 r_{ij},并给过程 p_i 的栈寄存器使用预算数加 1. 如果虽然在排序序列中的顺序号 $\leq bound$ 但它是在递归过程中使用的,那么假设每次递归调用都会引起寄存器栈溢出/载入. 比较在递归时引起的 RSE 代价是否高于 $reduced_cycles(r_{ij})$,如果 RSE 代价高就不使用它;反之使用 r_{ij},并对过程 p_i 的栈寄存器使用预算数加 1. 如果 r_{ij} 在序列中的顺序号已经超过 $bound$,那么 r_{ij} 引起的 RSE 代价必须小于 $reduced_cycles(r_{ij})$ 才能被使用.

最后,标记该路径上所有节点为已处理. 重复执行上述步骤直到所有节点都已经被处理过为止.

5.4 算法的时间复杂性

定理 1. 设程序调用图中的节点个数为 n,那么该算法的最坏时间复杂性为 $O(n^2)$.

证明. 根据算法,在遍历调用图选择种子节点和路径的过程中,时间复杂性为 $O(n)$,在最坏情况下,每次选择出的路径仅包含一个节点,那么这样的过程要重复 n 次. 因此,算法的时间复杂性为 $nO(n)$,最后的时间复杂性是 $O(n^2)$.

通常来说,选择的路径总是多于一个节点的,因此真实的时间复杂性会远远小于 $O(n^2)$.

6 实 验

6.1 实验环境

实验的硬件平台是一台有一个 733MHz Itani-

um 处理器的 HP 服务器,装有 2M 二级缓存. ORC 编译器在后端的优化包括条件转换、谓词分析、指令调度以及数据和控制投机等先进优化编译技术. 条件转换、谓词分析和指令调度都是基于域的编译框架进行的[11]. 我们用 SpecINT2000 的 12 个测试例子来测试算法的效果.

6.2　实验结果

图 8 所示为使用本文提出的算法以后,对于 SpecINT2000 的 12 个测试例子性能的影响. 这里我们的编译是在峰值模式进行的,也就是 O3+反馈信息. O3 优化级别表示所有常规的优化都是打开的. 从图上可以看到性能普遍有所提高,特别是 perlbmk,它的性能提高了 14%. Crafty 和 gcc,eon 等都有明显的性能提高,分别是 3.2%,1.2%和 2%. 只有 gzip 的性能略微下降了 1.2%.

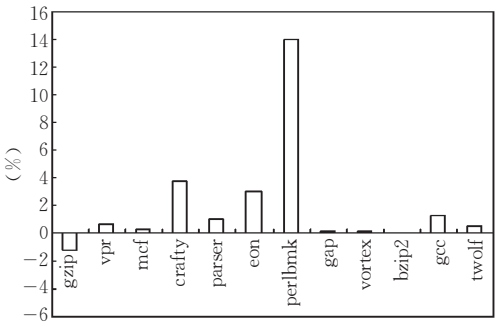


图 8　使用本文算法后 SpecINT2000 性能提高的百分比

6.3　实验结果分析

为了在 RSE 开销和减少栈寄存器使用而导致的溢出处理开销之间取得一个较好的平衡,有效地控制 RSE 代价,我们的算法必须能够做到如下两点:

首先,这个算法能够找出最热的路径,这些路径造成绝大部分的 RSE 开销.

其次,该算法应该能够合理地平衡栈寄存器的使用与否,从而使得整体的开销减小,性能得到提高.

在 SpecINT2000 中选择 perlbmk,crafty 和 gcc 作为分析用的测试例子来证明上述两点. 选择它们是因为 perlbmk 和 crafty 有着非常明显的性能提高,而 gcc 是一个非常大且复杂的程序.

首先通过实验来验证第一点. 如果本文算法找出的最热路径造成绝大部分的 RSE 开销,那么,控制这些路径上的寄存器使用在 96 个左右时,应该可以使得 RSE 开销的绝大部分被消除掉. 在这个实验中我们只选出前十条关键路径,并设定每个栈寄存器的溢出代价为 20,远远大于实际的情况,以尽量减少关键路径上的 RSE 开销,而不考虑增加的溢出处理代价. 实验结果显示(如图 9),crafty 和 perlbmk 的 RSE 几乎完全消失了,而 gcc 的 RSE 也从 1.5%下降到 0.5%. 这个实验证明算法对于找出导致大部分 RSE 开销的关键路径是非常有效的.

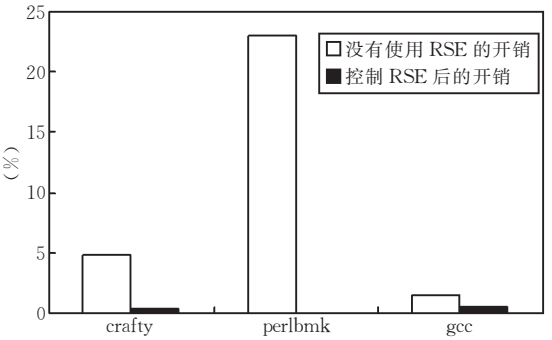


图 9　在关键路径上控制 RSE 后的 RSE 开销与没有使用本算法时 RSE 开销的比较

第二点我们可以通过使用本算法后总访存代价的减少来了解. 从图 10 我们可以看出,使用本算法以后,crafty,perlbmk 和 gcc 的总访存代价分别减少了 3.2%,18%和 0.9%. 这解释了为什么 perlbmk 的性能提高了 13%左右,而 crafty 性能提高了 3.2%. 图 11 显示了使用该算法后 RSE 的开销和没有使用本文算法时 RSE 开销的比较. 我们看到,除了 pelbmk 的 RSE 几乎完全消失以外,crafty 和 gcc 的 RSE 开销都只减少了 40%以上. 因为算法的目标是尽量减少总的访存开销,所以在 RSE 开销和溢出处理代价之间进行了折衷.

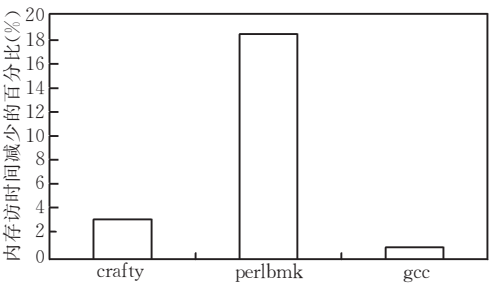


图 10　内存访问时间(包括 RSE 开销)的减少(占总运行时间的百分比)

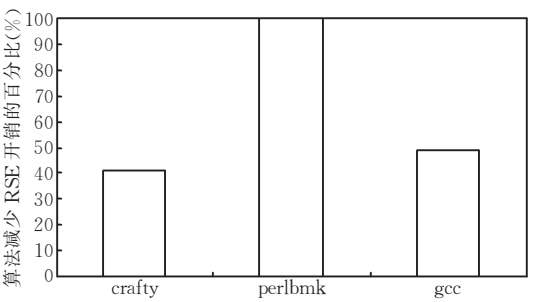


图 11　算法减少 RSE 开销的百分比

7 结论和未来的工作

虽然寄存器栈机制大大地减少了跨越过程调用时的加载/存储操作,但是 RSE 开销也不容忽视.使用 pfmon 1.1 版对 Spec2000INT 的可执行代码进行运行时测试,发现 perlbmk 的执行时间中 RSE 开销竟然占据了 23%,而 crafty 的 RSE 代价也高达 4.8%.快速发展的 CPU 处理速度会让 RSE 开销在整个执行时间中所占的比例越来越大,这使得该问题更加受到研究者的重视.

本文提出了一种全新的解决方法,并在开放研究编译器 ORC 上实现了该算法.实验结果证明算法对于控制 RSE 开销和提高程序执行性能效果明显,perlbmk 的性能提高了将近 14%,crafty 也有 3.2%的性能提高.

进一步的工作打算在过程间分析前对每个过程进行寄存器压力估算和活跃区间分析,从而减少获取反馈信息而需要的第一遍编译过程.

参 考 文 献

1 Chaitin G.. Register allocation and spilling via graph coloring. In: Proceedings of the SIGPLAN 82 Symposium on Compiler Construction, New York, 1982, 98～105

2 Chow F. C., Hennessy J. L.. Register allocation by priority-based coloring. In: Proceedings of the SIGPLAN'84 Symposium on Compiler Construction, NewYork, 1984, 222～232

3 Briggs P.. Register allocation via graph coloring[Ph. D. dissertation]. Rice University, Houston, Texas, USA, 1992

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: Proceedings of ACM SIGPLAN'97 Conference on Program Language Design and Implementation, Las Vegas, Nevada, 1997,296～307

6 Steenkiste P. A., Henessy J. L.. A simple interprocedural register allocation algorithm and its effectiveness for LISP. Transactions on Programming Languages and Systems, 1989, 11(1): 1～30

7 Wall D. W.. Global register allocation at link time. In: Proceedings of the SIGPLAN'86 Symposium on Compiler Construction, New York, 1986,264～275

8 Intel IA-64 System Architecture, Intel Company, 2002

9 Douillet A., Amaral J. N., Gao G. R.. Fine-grain stackedregister allocation for the Itanium architecture. In: Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing(LCPC), College Park, Maryland, 2002

10 David R. *et al*.. Quantitative evaluation of the register stack engine and optimization for future itanium processor. In: Proceedings of the 6th Annual Workshop on Interaction between Compilers and Computer Architectures, Boston, Massachusetts, 2002, 57～67

11 Liu Yang *et al*.. A region-based compilation framework. Chinese Journal of Computers, 2003,25(2):188～194(in Chinese) (刘 旸等.基于域的编译框架.计算机学报,2003,25(2):188～194)

LIU Yang, born in 1973, Ph. D. candidate. His research interest include instruction level parallelism, register allocation, multithread parallelism and low power optimization.

ZHANG Zhao-Qing, born in 1938, professor, Ph. D. supervisor. Her research interest include advanced compilation technology and compilation environment.

Background

The Open Research Compiler (ORC) project is an international cooperated project sponsored by the Institute of Computing Technology, CAS and Intel Company. This project aims at providing research and industry community a flexible, high performance and robust open source compiler targeted to IPF (Itanium® Processor Family) processors of Intel Company. In this project, authors focused on research of optimization methods that could improve the Instruction Level Parallelism efficiently. Authors implemented many Itanium® specific optimizations, such as if-conversion, data and control speculation, instruction scheduling integrated with micro-scheduling, and predicate analysis, etc. All these optimizations are region based. The performance of ORC compiler on Itanium2 for SpecINT2000 is comparable to the Intel product compiler. The research of this paper is trying to improve the register allocation performance.