3 种提高软件流水有效性的算法:比较和结合*

李文龙[1+], 陈 彧[1], 林海波[2], 汤志忠[1]

[1](清华大学 计算机科学与技术系,北京 100084)

[2](Intel 中国研究中心 编译组,北京 100080)

Three Algorithms for Improving the Effectiveness of Software Pipelining: Comparison and Combination

LI Wen-Long[1+], CHEN Yu[1], LIN Hai-Bo[2], TANG Zhi-Zhong[1]

[1](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

[2](Compiler Group, Intel China Research Center, Beijing 100080, China)

+ Corresponding author: Phn: +86-10-62773730, E-mail: liwenlong00@mails.tsinghua.edu.cn, http://www.tsinghua.edu.cn

Received 2004-04-16; Accepted 2004-11-22

Li WL, Chen Y, Lin HB, Tang ZZ. Three algorithms for improving the effectiveness of software pipelining: Comparison and combination. *Journal of Software*, 2005,16(10):1822−1832. DOI: 10.1360/jos161822

Abstract: Software pipelining is a loop scheduling technique that extracts instruction level parallelism by overlapping the execution of several consecutive iterations. One of its drawbacks is the high register requirements, which may lead to software pipelining failure due to insufficient static general registers in Itanium. This paper evaluates the register requirements of software-pipelined loops and presents three new methods for software pipelining loops that require more static general registers than those available in Itanium processor. They reduce register pressure by either reducing instructions in the loop body or allocating stacked non-rotating registers or rotating register in register stack to serve as static registers. These methods are better than the existing techniques in that they further improve performance gain from software pipelining by increasing software-pipelined loops. These methods have been implemented in open research compiler (ORC) targeted for Itanium processor, and they perform well on loops of the programs in NAS Benchmarks. For some benchmarks, the performance is improved by more than 21%.

Key words: software pipelining; static variant; static register; Itanium; loop unrolling; register allocation

摘 要: 软件流水是开发循环程序指令级并行性的技术,它通过并行执行连续的多个循环体来加快循环的执行速度.在软件流水中,循环体的重叠增加了寄存器需求,导致寄存器压力增大,当目标处理机所提供的寄存器不足时,软件流水可能失败.在 Itanium 处理机上评估了 NAS 和 SPEC2000 基准程序中的软件流水循环的寄存器需

* Supported by the National Natural Science Foundation of China under Grant No.60573100 (国家自然科学基金)

作者简介: 李文龙(1977−),男,辽宁鞍山人,博士,研究员,主要研究领域为计算机体系结构,指令级并行算法;陈彧(1981−),男,博士生,主要研究领域为指令级并行算法;林海波(1978−),男,博士,主要研究领域为计算机系统结构,指令级并行算法和多线程;汤志忠(1946−),男,教授,博士生导师,主要研究领域为计算机系统结构,指令级并行算法,并行编译技术.

求,发现静态寄存器不足是造成软件流水失败的主要原因,提出了 3 种增加软件流水个数、提高软件流水有效性的算法:限制循环展开因子的算法(register sensitive unrolling,简称 RSU)、堆栈寄存器分配算法(stacked register allocation,简称 SRA)以及变量类型转换的算法(variable type conversion,简称 VTC).RSU 根据静态寄存器需求确定一个合理的展开因子,增加了软件流水的成功率;SRA 和 VTC 分别使用空闲的堆栈寄存器和旋转寄存器来充当静态寄存器,提高了寄存器的利用率.在面向 Itanium 处理器的开放源码编译器 ORC(open research compiler)上实现了这 3 种算法,通过 NAS 程序的测试比较了这 3 种算法的有效性,同时对它们的结合应用进行了研究和实验.

关键词: 软件流水;静态变量;静态寄存器;Itanium;循环展开;寄存器分配
中图法分类号: TP302 文献标识码: A

随着应用程序对高性能处理机需求的增长,现代处理机对一些并行编译技术提供了硬件支持,比如基于显示并行指令计算体系结构(explicitly parallel instruction computing,简称 EPIC)的 Itanium 处理机,它对软件流水提供了硬件支持,这包括相应的旋转寄存器、条件执行以及特殊的循环分支指令等.

软件流水[1]是一种重要的指令调度技术,它通过并行执行连续启动的循环体来加快循环程序的执行速度.在软件流水算法中,模调度[2]是一类重要的启发式,它要求所有循环体的调度结果相同.经过模调度的软件流水循环,其相邻循环体的启动间隔为常数,称为启动间距.模调度在 20 世纪 80 年代初被首次提出,此后一直成为软件流水调度算法的研究热点,并在某些产品编译器中得以实现[3,4].要使软件流水后的循环能在目标机上执行,必须进行寄存器分配,将指令中的变量用目标处理机上的寄存器来表示,相关的、已成熟的寄存器分配算法见文献[5].

在软件流水中,因为循环体的重叠导致任意时刻的活变量增多,因而软件流水增加了循环的寄存器需求,导致寄存器压力增大.当目标处理机所提供的寄存器个数少于软件流水循环所需的数量时,通常进行以下几种处理:(1) 增大循环启动间距重新调度,启动间距的增大意味着软件流水性能的降低;(2) 在调度中插入 spill 和 refill 操作,将寄存器中的内容保存到内存或者从内存恢复,spill/refill 操作的引入增加了软件流水调度的复杂性,同时因为它们属于 memory 操作,占用内存带宽,所以这种方式会降低存储系统的性能,最终可能会导致程序性能下降;(3) 放弃软件流水,也就是说,因为寄存器不足,软件流水宣告失败.放弃软件流水意味着某些循环失去了软件流水优化的机会.

软件流水优化技术本身可以提高循环程序的性能,但是其增大的寄存器压力限制了它的应用,降低了软件流水技术的有效性.本文在基于 EPIC 体系结构的 Itanium 处理机上,对 NAS 和 SPEC2000 基准程序中的软件流水循环的寄存器需求进行了统计,分析了软件流水失败的主要原因,并针对因静态寄存器不足而导致软件流水失败的问题提出了 3 种解决算法,在开放源码编译器 ORC 上实现了这些算法,比较了 3 种算法的有效性(能够解决的软件流水失败循环个数和所能实现的加速比),并对它们的结合应用进行了研究和实验.

本文首先介绍软件流水的相关概念和实验框架.第 2 节对 NAS 和 SPEC2000 基准程序中的软件流水循环的寄存器个数进行统计,这包括通用(整数)寄存器和浮点寄存器,以及它们各自所包含的静态寄存器和旋转寄存器.第 3 节讨论软件流水失败的主要原因,提出 3 种解决因静态寄存器不足所导致的软件流水失败问题的算法.第 4 节给出这 3 种算法以及它们结合应用的实验结果,并对部分结果进行分析.最后给出相关工作以及结论.

1 相关概念和实验框架

1.1 软件流水和模调度

软件流水是一种有效的循环优化方法,它通过并行执行来自多个循环体的指令加快循环的执行速度.在软件流水中,相邻循环体的启动间隔称为启动间距(initiation interval,简称 II).软件流水要实现启动间距为 II 的调度序时,必须满足相关限制和资源限制.

模调度[2]是一类重要的软件流水调度启发式.它将一个循环体分为等长的段(stage),不同循环体的不同段

并行执行,执行每段所需的周期数为启动间距 II.由于模调度算法的简单性和低复杂性,很多处理器都对模调度的软件流水提供了硬件支持,比如 Cydrome 公司的 Cydra-5 处理器[6]和基于 IPF 体系结构的 Itanium 处理器[7].

1.2 寄存器分配

在循环中需要分配寄存器的变量可分为循环常量和循环变量两种.循环常量在循环的执行过程中被重复引用,但不被重新定值,故一个循环常量只需分配一个寄存器,这是与调度算法和体系结构无关的.

循环变量在循环执行过程中被多次定值,因此对于同一个循环变量,不同的循环体对应不同的变量值.在软件流水中,相邻循环体中循环变量的生存期可以重叠.根据体系结构的不同,变量生存期的定义也不同,本文中所指的变量生存期从定值操作开始起,到最后一个引用操作开始止.

因为循环体的重叠,导致了循环软件流水后,同一个变量的不同实例其生存期可能重叠,为了避免后续循环体中该变量的实例覆盖前面循环体中定义的值,研究者们提出了相应的寄存器分配算法[5].为了防止变量在引用之前被下一个循环体中的操作重写,一种解决方法是循环体展开多次,然后对同一变量的不同实例赋予不同的寄存器,使变量生存期小于展开后的软件流水启动间距,该方法称作模变量扩展(modulo variable expansion,简称 MVE).另外一种解决方法是通过硬件提供的旋转寄存器机制[6,7],由硬件完成寄存器的重命名.

1.3 实验框架

本文以 NAS 和 SPEC CPU2000 Benchmark 中所有的可进行软件流水的最内层循环为研究对象.这些循环中不含函数调用,也没有条件分支.条件分支通过 IF-conversion 消除.

所有的测试程序均用开发源码编译器 ORC(open research compiler)[8]进行编译.ORC 是 Intel 公司和中国科学院计算技术研究所共同开发的基于 IPF(Itanium processor family)体系结构的开放源码的研究型编译器,它的前身是 SGI 公司的 Pro64 编译器.ORC 采用了基于区域(region based)的编译技术.ORC 支持多种优化级别(–O0~–O3),这里将–O3 作为优化选项.在进行软件流水之前,ORC 首先对循环进行预处理,然后分别采用 Huff 的 Lifetime-Sensitive Modulo Scheduling 模调度算法[2]和 Rau 的 best-fit 寄存器分配算法[5]进行模调度和寄存器分配.

ORC 编译生成的代码可在基于 EPIC 体系结构的 Itanium 处理器[7]上运行.Itanium 是第 1 代基于 EPIC 处理器,每周期最多可发射 6 条指令,支持条件执行(predication).Itanium 提供了 128 个通用寄存器、128 个浮点寄存器、64 个条件寄存器、8 个分支寄存器,以及若干应用寄存器等.通用寄存器被分为静态寄存器(32 个)和堆栈寄存器(96 个)两部分.在堆栈寄存器中可以指定旋转寄存器的大小,但必须为 8 的整数倍.浮点寄存器也分为两部分:32 个静态寄存器和 96 个旋转寄存器.

本文的实验平台是 HP workstation i2000 工作站,它的配置如下:

处理器	1 Itanium @ 733 MHz
高速缓存	16K L1 DCache 16K L1 ICache 64K L2 Cache 2M L3 Cache (off die)
内存	1G SDRAM
总线频率	133 MHz

2 软件流水的寄存器需求

本节给出软件流水循环的通用和浮点寄存器需求,所得到的值是在模调度和寄存器分配之后统计出来的.

2.1 通用寄存器

Itanium 处理器提供了 128 个通用寄存器,其中 32 个为静态寄存器,其余 96 个为堆栈寄存器,从堆栈寄存器中可以分配 8 的整数倍个数的旋转寄存器.在 Itanium 中,有 3 种类型的变量需要分配通用静态寄存器,它们的定义如下:

定义 **1**(特殊变量). 需要占用系统专用寄存器的变量.在 Itanium 中,专用寄存器包括全局指针 gp(global

pointer)、堆栈指针 sp(stack pointer)、帧指针 fp(franme pointer)寄存器等.

定义 **2**(循环常量). 在循环体中只有引用没有定值的整数变量.

定义 **3**(地址后增变量). 地址后增指令中用于指定存储地址的变量.

Itanium 体系结构提供了地址后增类型的指令[9],比如指令

ld4 *r*1=[*r*2],4,

该指令从 *r*2 指定的内存中读取 4 字节的数据后,*r*2 中的值将自动加 4,再将结果保存到 *r*2 中.*r*2 所代表的变量称为地址后增变量,该指令为地址后增形式的指令.由于地址后增指令的特性,*r*2 的定值和引用均在同一条指令中,对于这种类型的变量,只能为其分配静态寄存器.

图 1、图 2 给出了 NAS 和 SPEC2000 中软件流水循环的通用静态寄存器需求统计(图中每点(*x*,*y*)表示所需寄存器少于等于 *x* 的循环所占的百分比).从变量的分布可知,软件流水循环中的特殊变量和循环常量很少.在 NAS 的所有 372 个可以软件流水的循环中,只有 5 个循环含有 1 个特殊变量,在所有的循环中,最多仅包含 6 个循环常量,大约 98%的循环只有 2 个以下的循环常量.统计结果表明,占用通用静态寄存器的绝大多数变量是地址后增变量,NAS 中有 10%的循环包含 20 个以上的地址后增变量.由于 Itanium 所提供的 32 个通用静态寄存器中,有 4~5 个属于特殊寄存器,或为系统保留,所以实际可分配的通用静态寄存器只有 27~28 个,NAS 中有 6.2%的循环需要 27 个以上的静态寄存器,所以这些循环在进行软件流水时会由于通用静态寄存器不足而导致失败;在 SPEC2000 中,99.87%的循环需要 24 个或更少的通用静态寄存器,仅有 0.13%的循环需要 27 个以上的通用静态寄存器,也就是说,SPEC2000 中因通用静态寄存器不足而导致软件流水失败的循环不足 1%.

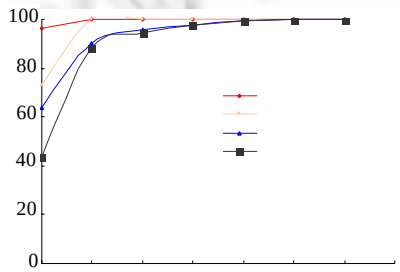


Fig.1 Cumulative distribution of variants requiring static general registers in NAS

图 1 NAS 中通用静态寄存器累积折线图

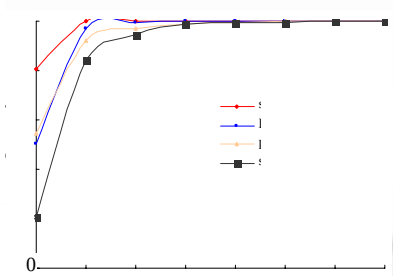


Fig.2 Cumulative distribution of variants requiring static general registers in SPEC2000

图 2 SPEC2000 中通用静态寄存器累积折线图

只有循环变量需要分配旋转寄存器,其所需的旋转寄存器个数与调度算法和寄存器分配算法有关.在 Itanium 中最多有 96 个通用旋转寄存器,其数目是可变的,但必须为 8 的整数倍.图 3、图 4 给出了 NAS 和 SPEC2000 基准程序中软件流水循环所需的通用旋转寄存器的累积折线图.

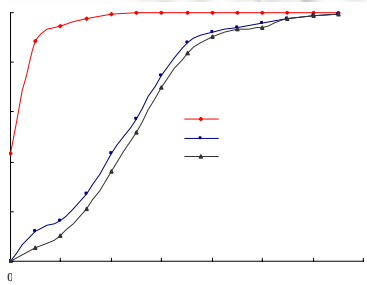


Fig.3 Cumulative distribution of general registers in NAS

图 3 NAS 中通用寄存器累积折线图

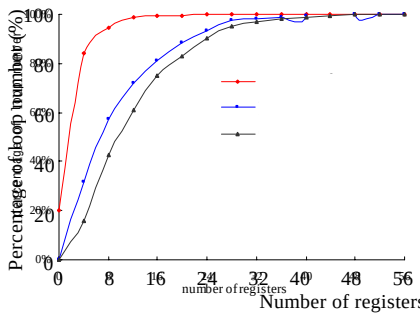


Fig.4 Cumulative distribution of general registers in SPEC2000

图 4 SPEC2000 中通用寄存器累积折线图

与 Itanium 所提供的物理寄存器相比,NAS 的循环所需的通用旋转寄存器并不算多(最多 64 个),90%的循环只需 32 个或更少的通用旋转寄存器,所以 NAS 中没有出现因通用旋转寄存器不足而导致软件流水失败的情况.SPEC2000 的循环在软件流水之后,98%的循环只需要 32 个或者更少的寄存器,所有 SPEC2000 中的软件流水循环至多需要 56 个旋转寄存器,因而 SPEC2000 中的循环在软件流水时,不会出现旋转寄存器不足.

2.2 浮点寄存器

Itanium 中浮点寄存器的结构比较简单,128 个浮点寄存器被分为 32 个浮点静态寄存器和 96 个浮点旋转寄存器.图 5、图 6 给出了 NAS 和 SPEC2000 中软件流水所需的浮点寄存器统计结果.

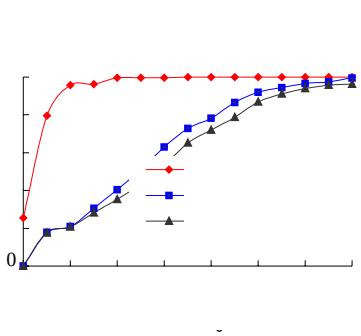


Fig.5 Cumulative distribution of floating point registers in NAS

图 5 NAS 中浮点寄存器累积折线图

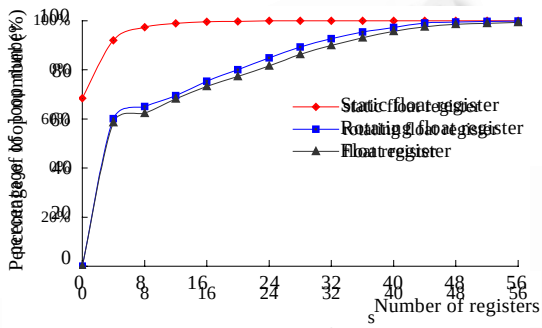


Fig.6 Cumulative distribution of floating point registers in SPEC2000

图 6 SPEC2000 中浮点寄存器累积折线图

NAS 循环中最多需要 28 个浮点静态寄存器,少于可用的物理寄存器个数(32 个),其中 99.7%的循环只需 24 个或更少的静态寄存器.浮点旋转寄存器最多所需的个数为 68,少于物理寄存器个数(96 个),97.4%的循环只需 52 个或更少的旋转寄存器. SPEC2000 循环中最多需要 24 个浮点静态寄存器,其中 99.6%的循环只需 16 个或更少的静态寄存器.浮点旋转寄存器最多所需的个数为 68,97.2%的循环只需 44 个或更少的旋转寄存器.统计结果表明,浮点寄存器没有导致软件流水失败.

2.3 其他寄存器的需求

Itanium 提供的大量寄存器除了通用和浮点寄存器以外,还有条件、应用等寄存器.实验结果统计表明,这些寄存器都是足够的.

3 软件流水失败的解决算法

3.1 软件流水失败的原因

在 NAS 和 SPEC2000 基准程序中,并不是所有满足软件流水条件的循环最终都能得到软件流水的优化,软件流水成功与否受下面两个因素制约.

3.1.1 循环体过大

当循环体中指令数量超过 130 个时,无环列表调度和软件流水调度对循环的性能贡献相差不多,但是在开销方面,软件流水的编译开销很高,同时寄存器压力也很大.因此,当循环中的操作个数超过 130 时,编译器放弃对循环的软件流水优化,而选用其他无环调度优化技术.

3.1.2 通用静态寄存器不足

上节我们分析了软件流水循环的寄存器需求,发现 6.2%的 NAS 循环会因为寄存器不足而软件流水失败.

表 1 统计了循环软件流水失败的个数,可以看出,静态寄存器不足是导致软件流水失败的主要原因(76.7%).表 2 给出了 23 个软件流水失败循环的特性,可以看出,这些循环至少还需要 4 个或 4 个以上的通用静态寄存器.

Table 1 Statistic of SWP failure in NAS benchmarks

表 1 NAS Benchmarks 中的软件流水失败统计

Cause of SWP failure	Number of loops with SWP failure	Percentage in SWP failure (%)	Percentage in all loops (%)
Loop too big	7	23.3	1.9
Insufficient general static register	23	76.7	6.2

Table 2 Statistic of loops in NAS due to insufficient general static register

表 2 NAS 中因通用静态寄存器不足而软件流水失败的循环统计

Benchmarks	Required general static register	Available general static register	Difference between required and available	Number of OP in loop body
BT	32	28	4	123
BT	32	28	4	123
CG	48	28	20	63
FT	32	28	4	39
LU	34	28	6	76
MG	41	28	13	58
MG	52	27	25	127
MG	60	27	33	121
MG	46	27	19	105
MG	32	27	5	95
MG	32	27	5	104
MG	32	28	4	43
MG	41	28	13	59
SP	35	28	7	48
SP	32	28	4	75
SP	34	28	6	77
SP	34	28	6	77
SP	32	27	5	81
SP	32	27	5	81
SP	32	27	5	81
SP	32	27	5	81
SP	32	27	5	81
SP	32	27	5	81

在接下来的几节里,我们将详细讨论 3 种解决静态寄存器不足的方法.

3.2 寄存器敏感展开因子算法——RSU(register sensitive unrolling)

现代编译器通常综合使用多种循环优化技术,循环展开就是其中的一种.循环展开可以使软件流水实现分数值的启动间距,提高资源的利用率.因此,在现代编译器中,通常在软件流水之前进行循环展开,以优化软件流水的调度结果.

但循环展开不可避免地会增加寄存器压力,展开次数过多甚至会使软件流水因可用寄存器不足而失败.通用静态寄存器是软件流水失败的主要原因,而地址后增变量占用了绝大多数的通用静态寄存器.因此,可以通过限制循环展开因子来减少需要通用静态寄存器的地址后增变量个数,这在一定程度上会提高软件流水的成功率,其算法如下:

令 N_a 等于可用的通用静态寄存器个数,N_d 等于循环中特殊变量的个数,N_i 等于循环中的循环常量个数,N_b 等于循环中地址后增变量个数,K_{old} 为原来的循环展开因子,则

$$K_{\max}=\begin{cases}+\infty & \text{if } N_b=0\\ \dfrac{K_{old}}{\min(K_{old},K_{\max})} & \text{otherwise}\end{cases} \tag{1}$$

$$K_{new}=\begin{cases}K_{old} & \text{if } K_{\max}=0\\ \min(K_{old},K_{\max}) & \text{otherwise}\end{cases} \tag{2}$$

$K_{\max}$ 是根据通用静态寄存器的需求确定的展开因子.由于循环常量和特殊变量的个数在展开前后没有变化,而地址后增变量的个数随着展开因子成比例变化,所以依据寄存器压力确定的展开因子是根据地址后增变量个数来确定的.

在式(2)中,如果 $K_{\max}$ 为 0,则表示展开之前循环的通用静态寄存器需求数量就已经超过了目标处理机所能

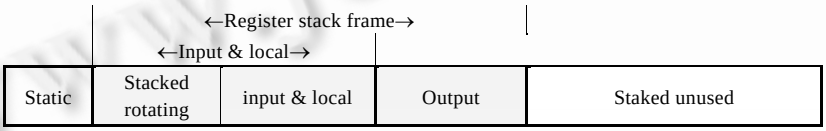
提供的个数,故保留原有的循环展开因子,以供后续处理;否则,取 K_{old} 和 $K_{\max}$ 的最小值,使得展开后的循环仍有足够的通用静态寄存器可用,以便进行后续的软件流水优化.

3.3 堆栈寄存器分配算法——SRA(stacked register allocation)

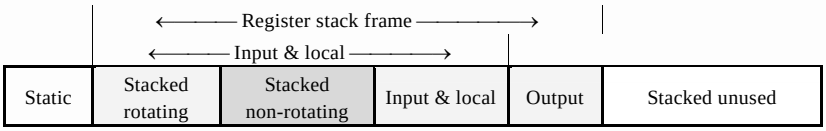
在 Itanium 中,通用寄存器 GR32-GR127 形成一个寄存器堆栈,由函数调用和返回机制进行自动管理.在寄存器堆栈中的每个帧分成两个动态大小的区域,一个区域保存输入参数和局部变量,另一个保存输出参数.硬件上不区分输入寄存器和局部寄存器.在函数调用时,硬件自动对寄存器重命名,调用者的输出寄存器对应着被调用者寄存器堆栈帧的输入部分;在函数返回时,寄存器自动恢复到前一个状态,因此在函数调用时要保存输入寄存器和局部寄存器.

在寄存器堆栈帧中,有一部分寄存器被指定为旋转寄存器.旋转寄存器总是从 GR32 开始,其个数为 8 的整数倍,至多可以分配 96 个旋转寄存器.

图 7 给出了应用 SRA 算法前后的寄存器堆栈使用模型.由于软件流水循环的旋转寄存器需求不是很高,因此通用寄存器堆栈中总有一些未用的寄存器.SRA 算法试图在堆栈寄存器中寻找未用的寄存器,并将它分配给需要静态寄存器的变量(如地址后增变量).实际上,SRA 算法在堆栈旋转寄存器的上面分配非旋转的堆栈寄存器给这些需要静态寄存器的变量,同时将寄存器堆栈帧的其余部分调整到一个较高的区域.这样就可以在静态寄存器和堆栈寄存器之间达到动态平衡,有效地提高软件流水的成功率和堆栈寄存器的利用率.



(a) Before SRA



(b) After SRA

Fig.7 Register stack usage model before and after SRA

图 7 SRA 算法前后的寄存器堆栈使用模型

SRA 算法的流程图如图 8 所示.

```
Perform_Loop_Optimizations() {
    CG_LOOP_Optimize() {
        Perform_SWP() {
            Modulo_Schedule();
            Allocate_Loop_Variants();
            //set MAX rotating reg number
            REGISTER_Reserve_Rotating_Registers();   SWP_Emit() {
                SWP_Rename_TNs() {
                    Get_Non_Rotating_Register_TN();
                }
            }
        }
    }
    SWP_Fixup() {
        SWP_Fixup_Rotating_Registers();
        //rename stacked non-rotating reg
        SWP_Fixup_Non_Rotating_Registers();
    }
    REGISTER_Request_Stacked_Rotating_Register();
    REGISTER_Request_Stacked_Non_Rotating_Register();
}
```

Fig.8 Outline of SRA algorithm

图 8 SRA 算法的流程图

在确定软件流水所需的旋转寄存器个数时,编译器是以编译单元为单位,而不是以循环为单位的.如果一个编译单元中含有多个循环,则堆栈寄存器中旋转寄存器的大小应取这些循环所需旋转寄存器个数的最大值.所以,如果在旋转寄存器之上为需要静态寄存器的变量分配堆栈寄存器,而在分配堆栈寄存器时并不知道实际所需的旋转寄存器数量,那么需要在整个编译单元处理之后进行重命名,这是 SRA 算法的一个难点.

3.4 变量类型转换算法——VTC(variable type conversion)

从前面的分析知道,Itanium 提供的 96 个旋转寄存器足够给所有的软件流水循环使用.当通用静态寄存器不足时,旋转寄存器仍有很多剩余.如果能将这些未用的旋转寄存器分配给那些需要通用静态寄存器的变量使用,那么软件流水的个数将会进一步增多,寄存器的利用率也会提高.

SRA 算法给未分配寄存器的地址后增变量分配堆栈非旋转寄存器,该算法没有减少循环的通用静态寄存器需求.这里提出了 VTC 算法,该算法通过将变量的类型从地址后增类型转换成循环类型,然后对转换后的循环变量分配未用的旋转寄存器,从而减少了循环的通用静态寄存器需求,增加了可以成功软件流水的循环数量,另外还提高了旋转寄存器的利用率.

对于地址后增形式的指令 ld4 *r*1=[*r*2],4(其中 *r*2 对应的变量为地址后增变量),VTC 将这条指令转换成下面形式的指令[9]:

ld4 *r*1=[*r*2]

adds *r*2=*r*2,4

这两条指令的语义与 ld4 *r*1=[*r*2],4 完全相同,此时 *r*2 对应的变量为循环变量,可以为其分配旋转寄存器来替代原来分配的通用静态寄存器.

VTC 算法通过将指令 REG1=OPCODE [REG2],Imm 转换成

REG1=OPCODE[REG2]和 ADDS REG2=REG2, Imm,

或指令 REG1=OPCODE [REG2],REG3 转换成

REG1=OPCODE[REG2]和 ADD REG2=REG2,REG3

来实现变量类型的转换.其流程如图 9 所示.

```
Perform_Loop_Optimizations() {
  CG_LOOP_Optimize() {
    Perform_SWP() {
      // calculate the number of base post-increment variants need to be converted
      Num_Convert = Get_Converted_Number();
      // calculate the number of OP in loop body
      Op_Count = Op_Num(loop);
      // Judge whether the loop becomes too big after conversion
      IF (Op_Count+Num_Convert≥OPS_LIMIT) THEN Return;
      // convert Num_Convert base post-increment variants to loop variants
      Convert_Variable(Num_Convert);
      Modulo_Schedule();
      Allocate_Loop_Variants();
      REGISTER_Reserve_Rotating_Registers();   //set MAX rotating reg number
      SWP_Emit()
      SWP_Rename_TNs()
    }
  }
  SWP_Fixup();
  REGISTER_Request_Stacked_Rotating_Register();
}
```

Fig.9 Outline of VTC algorithm

图 9 VTC 算法的流程图

需要转换类型的地址后增变量个数,其计算如下:

$$Required\_Num=N_d+N_i+N_b-N_a \tag{3}$$

$$Num\_Convert=\mathrm{Max}(0,Required\_Num) \tag{4}$$

在式(3)中,N_d 等于循环中特殊变量的个数,N_i 等于循环中的循环常量个数,N_b 等于循环中地址后增变量个

数,N_a等于可用的物理通用静态寄存器个数.

在 VTC 中,因为指令转换额外引入了一条指令,因此在应用 VTC 算法时,需要判断转换后循环体是否过大(即循环体指令数量超过了编译器规定的上限),如果过大则放弃转换.

4 实验结果

在 ORC 编译器上实现了本文所提出的 3 种算法(RSU,SRA 和 VTC),在 NAS Benchmarks 上,对这 3 种算法的有效性(因为 SPEC2000 中软件流水失败循环的比例不到 1%,故未对其进行测试)进行了测试.表 3 统计了这 3 种算法能够解决的软件流水失败循环个数.本文的所有实验结果都是采用–O3 编译选项运行程序获得的,并未使用 Profile 和 Inter-Procedural Analysis.

Table 3 Number of loops with SWP failure solved by RSU, SRA and VTC algorithms

表 3 各种算法能够解决的软件流水失败循环个数

Benchmarks	Default	RSU	SRA	VTC
BT	2	0	2	2
CG	1	1	1	1
EP	0	0	0	0
FT	1	1	1	1
IS	0	0	0	0
LU	1	0	1	1
MG	8	8	7	6
SP	10	10	10	10

在表中,Default 列对应着在没有应用这 3 种算法之前,各 benchmark 由于通用静态寄存器不足所导致的软件流水循环失败个数.后面的 3 列分别代表各种算法能够解决的软件流水失败循环个数.

在 ORC 原先的实现中,有 23 个循环因通用静态寄存器不足而失败,应用 RSU,SRA 和 VTC 后,仍有部分循环因寄存器不足而失败.RSU 算法有选择性地限制展开因子,从而降低了静态寄存器的需求,但是仍有 3 个循环失败,这是因为这 3 个循环在没有展开的时候,静态通用寄存器就已经不足.SRA 算法通过分配堆栈非旋转寄存器给需要静态寄存器的变量使用,增加了软件流水的循环个数.在 SRA 算法的实现中,规定能够分配的堆栈非旋转寄存器的上限为 32 个,如果分配过多,可能会导致程序的其他部分寄存器不足,增大 RSE 的开销,故在实现的时候做了这样的规定.对于 23 个因静态寄存器不足而导致软件流水失败的循环而言,SRA 算法解决了 22 个失败循环,只有 MG 中的循环未能解决,这是因为此循环需要的堆栈非旋转寄存器超过了 32 个.VTC 算法通过转换减少了地址后增变量的个数,但是仍有 2 个软件流水失败的循环未能解决,这是因为 VTC 转换后增加了循环体中的操作个数,导致循环体过大,因而软件流水仍旧失败.

图 10 比较了 3 种算法的性能,RSU 算法通过评估静态通用寄存器压力来计算展开因子,使平均加速比提高了 1.6%.SRA 算法和 VTC 算法没有改变循环展开因子,只是提高了寄存器的利用率,VTC 提高了 2.7%,SRA 提高了 4.7%.

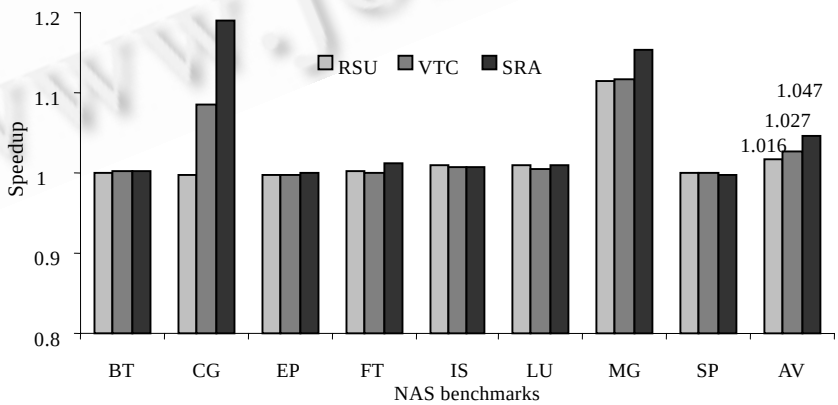


Fig.10 Performance of RSU, SRA and VTC algorithms

图 10 3 种算法的性能

从前面的实验结果可知,3 种算法都没有完全解决所有的因静态通用寄存器不足而导致的软件流水失败问题.如果将 3 种算法结合应用,其结果如何呢?表 4 统计了 3 种算法的结合应用能够解决的软件流水失败循环个数,图 11 给出了 3 种算法结合应用后的性能.

Table 4 Number of loops with SWP failure solved by combination of RSU, SRA and VTC algorithms

表 **4** 各种算法结合应用能够解决的软件流水失败循环个数

Benchmarks	Default	RSU+VTC	RSU+SRA	VTC+SRA	SRA+VTC
BT	2	2	2	2	2
CG	1	1	1	1	1
EP	0	0	0	0	0
FT	1	1	1	1	1
IS	0	0	0	0	0
LU	1	1	1	1	1
MG	8	8	8	8	8
SP	10	10	10	10	10

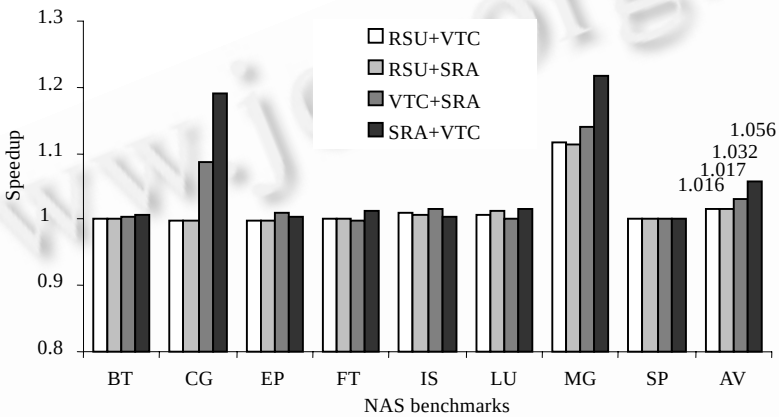


Fig.11 Performance of combination of RSU, SRA and VTC algorithms

图 11 3 种算法结合应用的性能

RSU+VTC 表示首先应用 RSU 算法,然后应用 VTC 算法,“+”表明算法应用的顺序.实验结果表明,算法的结合应用解决了所有因静态寄存器不足而引起的软件流水失败问题.3 种算法分别通过不同的方式来解决软件流水失败问题,它们的结合应用很好地弥补了各自的缺陷,程序的性能进一步得到了提高.

实验结果表明,SRA 算法和 VTC 算法的结合(先应用 SRA 算法,然后应用 VTC 算法)最有效,解决了所有因静态寄存器不足而引起的软件流水失败问题,同时性能贡献也最大,平均加速比为 5.6%.从软件流水中获益最大的程序是 mg,仅通过 8 个循环的软件流水,就可以得到 21.6%的加速比.而其他程序的加速比则并不明显.一方面可能是由于这些循环在执行时间上所占的比例不够大,另一方面则可能是由于软件流水的效率不是很高.

5 相关工作和结论

5.1 相关工作

关于软件流水和寄存器之间的关系,先前的学者们也做了一些研究工作.Mangione-Smith 等人[10]以一些较小的循环为例分析了软件流水和指令级并行对寄存器的影响.西班牙的 UPC(universitat politècnica de catalunya)在软件流水和寄存器方面的研究尤为活跃.Josep 等人[11]通过对 Perfect Benchmarks 中的循环进行量化分析,研究了流水线深度、功能部件个数以及循环优化技术对寄存器的影响.他们的研究指出,所需寄存器较多的循环,在程序执行时间中所占的比重也较大.文献[2]提出了在调度过程中考虑寄存器压力,即一个操作的放置位置要根据其前继(predecessor)和后继(successor)操作的位置来确定,这样所实现的调度结果与其他未考虑寄存器压力的模调度算法相比,在寄存器需求方面会小一些.文献[22]提出了一种称为寄存器队列的硬件来最

小化软件流水的寄存器压力.该硬件通过结合旋转寄存器堆和寄存器连接技术,为生成高效的软件流水调度提供了支持.文献[13]提出了面向嵌入式处理机的软件流水算法.该算法在软件流水调度过程中不仅考虑以往算法所考虑的相关和资源限制,同时也加入了嵌入式处理机的限制,包括代码大小和寄存器压力.文献[14]提出了在软件流水调度的过程中考虑寄存器压力的软件流水算法.它类似于 Huff 的生存期敏感算法[2],只是调度空间的搜索不一样,Huff 的搜索空间只是 II 数值大小的范围,而该算法搜索整个调度空间,所确定的调度位置更有利于降低寄存器的压力.

5.2 结 论

通过对 Itanium 中软件流水失败问题的分析,可以得出以下结论:

1) 在 Itanium 体系结构中,通用静态寄存器是软件流水成败的主要原因;

2) RSU 算法通过适当地限制循环展开因子,较为有效地解决了软件流水失败的问题,并且提高了应用程序的执行速度;

3) SRA 算法和 VTC 算法动态的调整了静态寄存器和动态地寄存器之间的比例,提高了寄存器的利用率,比 RSU 更有效地解决了软件流水失败问题,且性能也优于 RSU.在这 3 种算法中,SRA 相对于 RSU 和 VTC 更加有效.

4) 将 SRA 和 VTC 结合起来应用可以取得更好的效果,解决了所有的因通用静态寄存器不足导致的软件流水失败问题,同时性能优于其他方法以及它们的组合.

References:

[1] Allen VH, Jones RB, Lee RM, Allan SJ. Software pipelining. ACM Computing Surveys, 1995,27(3):367−432.

[2] Huff RA. Lifetime-Sensitive modulo scheduling. In: Budd TA, ed. Proc. of the ACM SIGPLAN'93 Conf. on Programming Language Design and Implementation. New York: ACM Press, 1993. 258−267.

[3] Dehnert JC, Towle RA. Compiling for the Cydra 5. Journal of Supercomputing, 1993,7(1−2):181−228.

[4] Dulong C, Krishnaiyer R, Kulkarni D, Lavery D, Li W, Ng J, Sehr D. An overview of the Intel IA-64 compiler. Intel Technology Journal, 1999.

[5] Rau BR, Lee M, Tirumalai PP, Schlansker MS. Register allocation for software pipelined loops. In: Allen R, ed. Proc. of the ACM SIGPLAN'92 Conf. on Programming Language Design and Implementation. New York: ACM Press, 1992. 283−299.

[6] Dehnert JC, Hsu PY, Bratt JP. Overlapped loop support in the Cydra 5. In: Hennessy J, ed. Proc. of the 3rd Int'l Conf. on Architectural Support for Programming Languages and System. New York: ACM Press, 1989. 26−38.

[7] Intel Corporation. Intel Itanium™ Architecture Software Developer's Manual. Volume 1: Application Architecture. Intel Corp., 2001.

[8] Roy J, Sun C, Wu CY. Tutorial: Open research compiler for Itanium processor family (IPF). In: Proc. of the 34th Annual Int'l Symp. on Microarchitecture. New York: ACM Press, 2001.

[9] Intel Corporation. Intel Itanium™ Architecture Software Developer's Manual. Volume 3: Instruction Set Reference. Intel Corp, 2001.

[10] Mangione SW, Abraham SG, Davidson ES. Register requirements of pipelined processors. In: Kennedy K, Polychronopoulos CD, ed. Proc of Int'l Conf. on Supercomputing. New York: ACM Press, 1992. 260−271.

[11] Josep L, Eduard A, Mateo V. Quantitative evaluation of register pressure on software pipelined loops. Int'l Journal of Parallel Programming, 1998,26(2):121−142.

[12] Smelyanskiy M, Tyson GS, Davidson ES. Register Queues: A new hardware/software approach to efficient software pipelining. In: Hurson AR, ed. Proc. of the 2000 Int'l Conf. on Parallel Architecture and Compilation Techniques. IEEE Press, 2000.

[13] Akturan C, Jacome MF. RS-FDRA: A register sensitive software pipelining algorithm for embedded VLIW processors. In: Madson J, Henkel J, Hu XBS, eds. Proc. of the 9th Int'l Symp. on Hardware/Software Codesigh. New York: ACM Press, 2001.

[14] Altemos G, Norris C. Register pressure responsive software pipelining. In: Lamont GB, ed. Proc. of the 2001 ACM Symp. on Applied Computing. ACM Press, 2001.