文章编号:1001 - 9081(2006)08 - 1919 - 03

IA-64 软件流水的反流水算法研究

崔平非,庞建民,赵荣彩,崔雪冰
(信息工程大学 信息工程学院,河南 郑州 450002)
(cpf1975@126.com)

摘 要:软件流水是一种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迭代来加快循环的执行速度。而在逆向工程中,软件流水却为逆向翻译带来了困难。为此,基于 IA-64 平台,提出了一种反流水算法,针对循环中包含软件流水的汇编代码进行处理,将其反向转换成语义等价的串行代码,并通过实验验证了该算法的有效性,为在二进制翻译中处理软件流水代码奠定了基础。

关键词:软件流水;反软件流水;模调度;逆向工程
中图分类号: TP311.51 **文献标识码**:A

Study on de-pipeline algorithm for software-pipeline of IA-64

CUI Ping-fei, PANG Jian-min, ZHAO Rong-Cai, CUI Xue-bing
(*College of Information Engineering, Information Engineering University, Zhengzhou Henan* 450002, *China*)

Abstract: Software pipelining is a loop scheduling technique which extracts instruction level parallelism by overlapping the execution of several consecutive iterations. But in reverse engineering, it causes some difficulties to reverse translation. A de-pipelining algorithm based on the IA-64 architecture was proposed. This algorithm reversely converted the optimized assembly code of a software-pipelined loop to a semantically equivalent sequential counterpart. The experimental results have verified the validity of the proposed de-pipelining, which provides the basis for processing software pipelining in the binary translation.

Key words: software pipelining; de-pipelining; Modulo Scheduling; reverse engineering

0 引言

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的迭代来加快循环的执行速度。在软件流水算法中,模调度[2]是一类重要的实现方式,它要求所有迭代的调度结果相同。IA-64 是 Intel 和 HP 公司联合推出的 64 位指令系统,它代表着这两个公司提出的显式并行指令计算(Explicitly Parallel Instruction Computing,EPIC)构架,它支持模调度循环特征[3],利用它可以实现软件流水化过程,使得构造高效且紧凑的代码序列成为可能。

虽然软件流水技术可以有效地提高代码的执行效率,但由于其受到硬件平台的制约,对代码的跨平台移植产生较大影响。在软件逆向工程中,特别是在二进制翻译中,若源机存在对软件流水技术的硬件支持,则在目标代码中将会出现相关流水指令描述。而在目标机中若无支持软件流水的相应硬件,就需要消除代码的硬件相关性,即消除流水特性,将源处理机软件流水后的优化汇编代码反向转换成语义等价且无硬件依赖的串行代码。

对于反流水研究,文献[4]提出了一种基于 TIC62 处理机汇编语言代码的反流水算法。IA-64 与 TIC62 处理机存在着很大的区别,例如 IA-64 中存在循环寄存器、谓词寄存器等硬件支持。

本文提出一种针对 IA-64 处理器中软件流水的反流水算法。通过对软件流水特殊指令的识别来判断一个包含软件流水的循环,运用所提出的反流水算法将其转换成语义等价的串行代码。

1 IA-64 中软件流水机制及相关硬件支持

1.1 软件流水机制

传统的体系结构使用循环展开技术[5]来将循环转化成一种执行更快的等价形式。但循环展开有一些严重的缺点,如当循环次数是由运行中的程序动态确定时,那么只能期望通过非常复杂的编译技术来设计一种展开算法。

与传统的循环展开技术相比,IA-64 使用软件流水技术对一些循环进行优化。下面是计算两个多元素向量的标量积的循环经过软件流水后的代码,用 IA-64 的汇编语言编写:

```
top:
    (p16) ld4   r32 = [r14],4
    (p16) ld4   r36 = [r15],4
    (p17) pmpy4.r   r33 = r33, r37
    (p19) add   r20 = r20, r35
    (p18) sxt4  r34 = r34
br.ctop.sptk.few top;;
```

在软件流水算法中,模调度[6]通过在当前迭代结束之前启动后继迭代的方法,提供了一种增大吞吐量的机制。相邻两个迭代的启动时刻差称为启动间距(Initiation Interval,II)。在模调度中,所有迭代的调度结果相同,而且按照一个固定的

收稿日期:2006 - 02 - 16;**修订日期**:2006 - 03 - 26
基金项目:河南省杰出人才创新基金资助项目(0521000200);国家重大专项子课题
作者简介:崔平非(1975 -),男,河南清丰人,硕士研究生,主要研究方向:软件逆向工程; 庞建民(1964 -),男,河北沧州人,教授,主要研究方向:计算机辅助推理、软件逆向工程; 赵荣彩(1957 -),男,河南洛宁人,教授,博导,主要研究方向:软件逆向工程、分布式计算; 崔雪冰(1972 -),女,河南新乡人,讲师,硕士研究生,主要研究方向:软件逆向工程.

启动间距依次启动。

模调度的结果(如图1)分为三个阶段[5]:装入、核心和排空。在装入阶段,每经过 II 间隔将启动一轮新的循环去填充流水线,直到核心阶段,流水线被完全填充。在核心阶段,每经过 II 间隔一个新的循环体启动且另一个循环体执行结束。离开核心后进入排空阶段,在排空阶段,没有新的循环启动,但流水线中的循环还未执行完,所以必须完成这些循环的执行。如图1中的第4~6行循环将在排空阶段被完成。

周期	1	2	3	4	5	6	阶段
1	ld4			Intiation			
2	pmpy4	ld4		interval			Prolog
3	sxt4	pmpy4	ld4				
4	add	sxt4	pmpy4	ld4			
5		add	sxt4	pmpy4	ld4		Kernel
6			add	sxt4	pmpy4	ld4	
7				add	sxt4	pmpy4	
8					add	sxt4	Epilog
9						add	

图1 模调度

IA-64 中实现软件流水机制使用以下特殊的循环分支指令:br. ctop,br. cexit,br. wtop 和 br. wexit。其中,br. ctop 和 br. cexit 用来构造记数循环,br. wtop 和 br. wexit 用来构造 while 循环[7,8]。本文主要对 br. ctop 循环进行反流水描述。

1.2 IA-64 中软件流水的相关硬件支持

为了配合软件流水技术,IA-64 增加了一些硬件支持:谓词寄存器、应用程序寄存器和旋转寄存器。

1.2.1 谓词寄存器

IA-64 提供了 64 个 1 位寄存器,当成布尔值使用。这类寄存器中,0 对应着逻辑上的 false,1 对应着逻辑上的 true。使用这些寄存器来控制指令是否执行。

1.2.2 应用程序寄存器组

应用程序寄存器组共有 128 个,每个宽 64 位。它们是一些专用的数据/控制寄存器,通常用作某些指令的隐式参数。软件流水中主要用到其中的两个寄存器:ar65 和 ar66,其中 ar65 为 lc 寄存器,它在计算循环分支时是一个隐式的运算数,它决定了循环迭代的次数;ar66 为 ec 寄存器,它是模调度循环分支指令的隐式运算数,指出了循环中流水线的场景数。

1.2.3 旋转寄存器

IA-64 为软件流水提供了旋转寄存器的硬件支持,使得编译器不必显式地进行寄存器重命名。谓词寄存器 p16~p63、通用寄存器 r32~r127 和浮点寄存器 f32~f127 都能标记为旋转,这些寄存器能与特定循环分支配合使用以实现模调度循环。

旋转通用寄存器总是从 r32 开始,alloc 指令能够指派任何 8 的倍数个通用寄存器以一个循环式寄存器集的方式操作。如指令:

```
allocr40 = ar. pfs, 0, 24, 4, 8
```

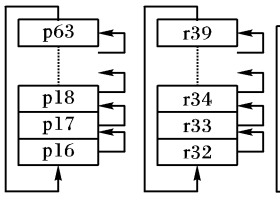
其中的 8 就是说明指派从 r32 开始的 8 个通用寄存器为旋转通用寄存器。

旋转谓词寄存器在进入循环之前,通过一条 mov 指令初始化。如:

```
mov pr. rot = 0x10000
```

其作用是:把源操作数复制到谓词寄存器中,屏蔽了低 16 位,即 p16 的值为 1,p17~p63 的值为 0。

当循环遍历遇到一个软件流水循环指令时,每个循环式寄存器集通过增加寄存器号来承担隐式的重命名工作:来自第一次遍历的 p16 现在是 p17,而 p63 成为第二次遍历的新的 p16;r32 成为 r33,而 r(32 + rot - 1)(其中 rot 为 alloc 指令中所说明的旋转通用寄存器个数)成为新的 r32(图2(a))。



(a) 谓词和通用寄存器旋转

	Ar.lc	ar.ec	p63	rrb
装入阶段	递减	不变	1	递减
核心阶段	递减	不变	1	递减
排空阶段	0	递减	0	递减

(b) 计数循环中寄存器的循环操作

图2 计数循环指令的效果

CPU 硬件采用以下方式来管理寄存器循环:使用一个可分为三部分的寄存器重命名基(Register Rename Base,RRB)寄存器作为当前帧标记(CFM)的一部分。这个 RRB 的谓词和通用寄存器都分别通过执行特殊的转移指令来递减。由于有效的寄存器名总是 RRB 的值与嵌入在一个 IA-64 指令中的寄存器号的和,循环使寄存器 N 中的数据表现为移动到寄存器 $N+1$,以便为按模调度的循环的下一次操作周期作准备。也就是说,在一个周期中的 $RRB+N$ 指向下一个操作周期的 $(RRB-1)+(N+1)$ 所表示的同一个物理寄存器。可以用图2(b)表示计数循环中的这些寄存器的循环操作。

2 反软件流水算法

下面结合001. C 中软件流水循环汇编代码示例来说明我们所研究的反软件流水算法,此例是本课题测试组提供的测试集中 10_16_BitCardStructure. c(此程序是对一个结构体数组进行处理)经 icc 编译后所生成的循环经过软件流水的汇编代码,其中删除了与说明算法无关只为了满足 IA-64 特殊指令束格式的 nop(空操作)指令。对此例运用所提出的反软件流水算法生成语义等价的串行代码(如反流水后语义等价的串行代码所示),其中也删除了 nop 指令。

001. C 中软件流水循环汇编代码示例如下:

```
.b2_1:
  { .mmi
    (p17) addr41 = r38, r37
    (p18) st1[r2] = r39, 4
    (p16) shrr35 = r33, 34 }
  { .mlx
    (p16) movlr42 = 0x04ec4ec4f; ; }
  { .mii
    (p17) ld1r38 = [r2]
    (p16) extrr37 = r36, 31, 1
    (p16) addr32 = r33, r42; ; }
  { .mii
    (p16) subr39 = r35, r37
    (p17) depr42 = r41, r38, 0, 4; ;
    (p16) shrr38 = r33, 35}
  { .mii
    (p16) sxt4r35 = r39; ;
    (p17) depr44 = r40, r42, 4, 2}
  { .mmi
    (p16) shladdr41 = r35, 1, r35
    (p16) shladdr43 = r35, 4, r0
    (p16) subr33 = r38, r37; ; }
  { .mii
```

```
    (p16) subr37 = r41, r43
    (p17) depr38 = r34, r44, 6, 1
    (p16) addr35 = 1, r36}
  { .mfb
    br.ctop.sptk.b2_1;; }
```

反流水后语义等价的串行代码示例如下:

```
.b1_1:
  { .mmi
    shr r35 = r33, 34 }
  { .mlx
    movlr42 = 0x04ec4ec4f;;}
  { .mmi
    extrr37 = r36, 31, 1;; }
  { .mii
    add r32 = r33, r42
    sub r39 = r35, r37;;
    shr r38 = r33, 35 }
  { .mmi
    sxt4r35 = r39;;}
  { .mmi
    shladdr41 = r35, 1, r35
    shladdr43 = r35, 4, r0
    sub r33 = r38, r37;; }
  { .mmi
    sub r37 = r41, r43;;
    add r35 = 1, r36
    add r40 = r37, r36;; }
  { .mii
    ld1 r37 = [r2];;
    dep r41 = r40, r37, 0, 4;; }
  { .mii
    dep r43 = r39, r41, 4, 2;;
    dep r37 = r33, r43, 6, 1;; }
  { .mmi
    st1 [r2] = r37, 4;;
    mov r36 = r35
    mov r33 = r32 }
  { .mib
    br.cloop.sptk.few .b1_1;;}
```

反软件流水算法的主要思想是:根据旋转寄存器的实现特征及谓词寄存器的作用,要消除软件流水循环代码的流水特性,在算法中就要完成调整指令顺序、恢复寄存器号和消除谓词控制的工作。同时还要考虑调整后的指令是否满足IA-64特殊的指令束格式以及是否出现数据依赖等问题。

反软件流水算法的步骤如下(图 3):

1)软件流水循环体识别。将 br.ctop 指令所转移的目的地址所对应指令作为软件流水循环体的入口指令,它与 br.ctop指令及其它们之间的所有指令构成软件流水循环体。

2)重新调整指令顺序。由于要生成串行代码,须对循环体中的指令顺序重新进行调整,调整规则如下:

(1)将无谓词寄存器控制的指令按照它们在循环体中的先后顺序排列,组成调整后的循环体的第一部分。

(2)对带有谓词控制的指令进行调整:按照谓词寄存器号由小至大排列,其中谓词寄存器号相同的指令按照在原循环体中的顺序排列。由此组成调整后的循环体的第二部分。

以上指令调整对 br.ctop 指令不起作用,即 br.ctop 指令仍作为循环体的最后一条指令。

3)恢复旋转寄存器号。要消除流水特性,就必须消除为流水而引入的旋转寄存器的旋转特性,也就是将旋转寄存器恢复为旋转前的物理寄存器。对调整后的循环体第二部分的所有指令进行以下操作:

(1)判断指令中使用的寄存器是否为累加寄存器:满足 add ri = imm,rj 且 i = j - 1(其中 r 表示通用寄存器,i,j 为寄存器号且 i,j≥32,imm 为立即数),则 ri 为累加寄存器。若 ri 为累加寄存器,则在 br.ctop 指令前加入指令 mov r(i + 1) = ri。若不是,继续(2)。

(2)若指令的谓词寄存器 px(其中 p 表示谓词寄存器,x 为寄存器号)满足 x≥16,则该指令中的旋转寄存器 ri(i≥32)恢复为 r(i - x + 16)。

4)消除谓词寄存器。消除循环体中指令的谓词部分。例如指令:(p16) shr r35 = r33,34,消除谓词后成为 shr r35 = r33,34。

5)是否满足指令束模板格式要求。因为 I、M、B、F 型指令与用于译码和执行它们的 I、M、B、F 型的执行部件之间存在着一种所期望的对应关系,而在调整指令顺序时必定会导致指令和原指令束模板出现不匹配,因此要进行指令束格式检查和修改。

从调整后的循环体入口指令开始进行如下操作:

(1)判断是否为 br.ctop 指令,若是,转步骤 6);

(2)该指令若无指令束模板可以匹配,根据指令添加相应的执行单元类型,转(4)。

(3)该指令若有指令束模板可以匹配,判断是否符合相应执行单元要求。若不是,修改模板中执行单元类型。

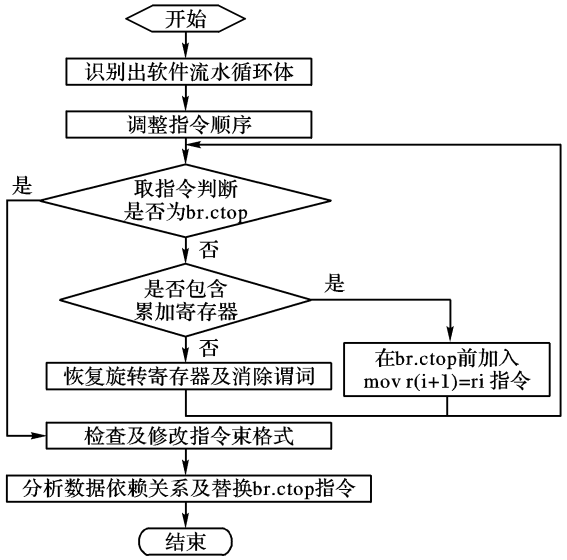


图 3　反软件流水算法流程

(4)判断是否满足指令束模板格式要求,若不是,则在该指令前添加与执行单元对应的 nop 指令。

(5)取下一条指令,转(1)。

6)分析数据依赖关系。首先对循环体中指令构造依赖 DAG(算法描述参见文献[9]),然后依照依赖 DAG 对数据依赖关系进行分析。若指令束中相邻指令以及相邻的两个指令束中指令出现流依赖或输出依赖[9],则对产生依赖的指令组中第一条指令添加停止位。

7)将 br.ctop 指令转换成 br.cloop 指令。最后,将控制软件流水的特殊转移指令 br.ctop 指令转换成与流水无关的条件转移指令 br.cloop 指令。

(下转第 1927 页)

了本文对 Bogart 的改进是成功的,特别是在速度方面。

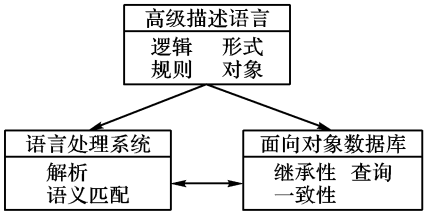


图 3　改进后的 Bogart 组件关系

表 1　翻译程序的结果比较

	传统翻译器		Bogart		改进的 Bogart	
	时间/s	空间/byte	时间/s	空间/byte	时间/s	空间/byte
Bin	63	4 170	33	2 802	23	2 798
Horner	10	3 302	3	2 465	2	2 402
Random	9	5 447	4	2 741	3	2 636
SAPDBMS	18	41 700	9	29 043	6	28 741

4　结语

介绍了把 Re-Engineering 结合到自动翻译器中的方法,目的在于提高目标代码的质量。从前面的描述可以看到,应用Re-Engineering的翻译器可以使代码达到一般的正确性、可读性、易维护性和可复用性等特点,但是它需要建立一个好的抽象模型,并且抽象模型的再实现本身也是很复杂的过程,因此 Re-Engineering 翻译器的功能尚不完善。在此基础上,我们提出了两个改进方案,并通过测试证明了改进的正确性。今后的主要工作是以 Bogart 为蓝本,设计出效率更高的 Re-Engineering 翻译器,使其达到实用化的程度。

参考文献:

[1] WATERS RC. Program Translation via Abstraction and Reimplementation[J]. IEEE Trans Software Engineering, 1988, 14(8): 1207 - 1228.

[2] CHU CW, PATEL S. A Re-engineering Tool for the Reuse of Large Scale Software Systems[A]. Proceedings of The Fifth International Conference on Software Engineering and Knowledge Engineering [C]. 1993. 94 - 101.

[3] BLAIR A. A methodology for the design of knowledge based decision support systems[A]. Proceedings of International Conference on Expert Systems for Development[C]. 1994. 18 - 23.

[4] WILLS LM. Automated program recognition by graph parsing[R]. Technical Report 1358, MIT Artificial Intelligence Lab, 1992.

[5] CHU CW, PATEL S. Software Restructuring By Enforcing Localization and Information Hiding[A]. Proceedings of The International Conference on Software Maintenance[C]. 1992. 165 - 172.

(上接第 1921 页)

3　实验结果

实验环境是一台拥有四个 Intel Itanium2 1.3G 处理器和内存为 2.0G 的服务器,操作系统内核是 linux2.4.21-4.EL,所用的编译器是 ICC 8.0,对本课题测试组提供的测试集中的所有经 ICC 编译后生成的汇编代码中出现 br.ctop 指令的例子进行测试。实验中,首先使用 ICC 编译器对源程序编译成汇编代码,然后运用我们所提出的反流水算法对出现的软件流水循环代码转换成串行代码,再将该汇编程序汇编成可执行文件,最后在 IA-64 上运行此可执行文件以验证该算法的正确性。实验结果完全正确。表 1 列出了软件流水和反流水后循环体中指令束个数和通用寄存器个数的对比结果。

表 1　实验结果

汇编程序	软件流水循环体		反软件流水后循环体	
	指令束个数	分配通用旋转寄存器个数	指令束个数	使用通用寄存器个数
10_16_BitCardStructure.s	8	16	11	11
10_20_Mask_Bits.s	2	8	3	3
10_3_Card_Shuffle.s	7	16	12	10
10_28_Array_Modify.s	2	8	2	4
10_30_Surveydata_Analysis.s (1)	1	8	2	3
10_30_Surveydata_Analysis.s (2)	4	8	7	4

4　结语

软件流水作为一种先进的编译技术,对于提高指令执行的并行性有重要的作用,但也给逆向工程带来了困难,该论文描述的反流水算法,不仅可消除代码的硬件特性,还可以减少循环对寄存器的需求,降低寄存器压力,同时也提高了原代码的可读性。然而对于 while 循环,其难点在于其循环次数受到条件的限制,并且我们在研究过程中发现一些程序中在循环体外会使用循环体内流水后的寄存器。下一步主要工作将针对此类循环研究其反流水算法。

参考文献:

[1] LAM M. Software Pipelining: An Effective Scheduling Technique for VLIW Machines[A]. Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation[C]. Atlanta, Georgia, United States, 1988. 318 - 328.

[2] HUFF RA. Lifetime-Sensitive modulo scheduling[A]. Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation[C]. New York: ACM Press, 1993. 258. 267.

[3] MOSBERGER D, ERANIAN S. IA-64 Linux Kernel Design and Implementation[M]. Beijing: Tsinghua University Press, 2004.

[4] TANG ZZ, LI WL, SU BO. A de-pipeline algorithm for software-pipeline[J]. Journal of Software, 2004, 15(7): 987 - 993.

[5] EVANS JS, TRIMPER GL. Itanium Architecture for Programmers: Understanding 64-Bit Processors and EPIC Principles[M]. Beijing: Tsinghua University Press, 2004.

[6] LADSARIA A. Modulo Scheduling[EB/OL]. http://www.crhc.uiuc.edu/ece411/sp02/Slides02/modulo.pdf, 2002 - 05 - 05.

[7] Intel IA-64 Architecture Software Developer's Manual——Overview of Volume 3: Instruction Set Reference[EB/OL]. http://developer.intel.com/design/litcentr, 2000 - 01.

[8] Intel Itanium Architecture Software Developer's Manuall—— Volume 1: Instruction Set Reference[EB/OL]. http://www.intel.com/design/itanium/manuals, 2002 - 10.

[9] MUCHNICK SS. Advanced Compiler Design Implementation[M]. Beijing: China Machine Press, 2003.