基于流体系结构的 VLIW 二维压缩及并行解压

李功丽[1,2],戴紫彬[1],徐进辉[1],王寿成[1],朱玉飞[1],李 丹[3]
(1. 解放军信息工程大学,河南郑州 450001;2. 河南师范大学计算机与信息工程学院,河南新乡 453002;
3. 中国人民解放军61840部队,北京 100089)

摘 要: VLIW(Very Long Instruction Word)指令因为含有较多的空操作导致严重的代码体积膨胀问题,代码压缩是解决这一问题的有效措施. VLIW 代码压缩需要解决三个关键问题,一是提高压缩率;二是降低解压操作对性能的影响;三是分支目标重定位. 针对流体系结构上的 VLIW 指令特点,提出了二维压缩,对 VLIW 进行垂直与水平两个方向上的压缩,且水平解压可以与代码执行并行,并通过设置堆栈寄存器缓存循环入口地址. 实验结果表明二维压缩有效解决了 VLIW 代码体积膨胀问题,可以使指令存储器的面积减少 36.48%,并使得整个 CISP 系统面积减少了 7.85%.

关键词: 流处理器; VLIW; 二维压缩; 并行解压
中图分类号: TP302 **文献标识码:** A **文章编号:** 0372-2112 (2017)09-2256-07
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2017.09.029

2-D Compression and Parallel Decoding of VLIW Based on Stream Architecture

LI Gong-li[1,2], DAI Zi-bin[1], XU Jin-hui[1], WANG Shou-cheng[1], ZHU Yu-fei[1], LI Dan[3]
(1. *PLA Information Engineering University, Zhengzhou, Henan* 450001, *China*;
2. *College of Computer & Information Engineering, Henan Normal University, Xinxiang, Henan* 453002, *China*;
3. *PLA of Unit* 61840, *Beijing* 100089, *China*)

Abstract: Due to having many NOPs, VLIW (Very Long Instruction Word) exists serious code size expansion problem. As an efficient way to solve this problem, the code compression needs to deal with three key points: improving the compression ratio (CR), simplifying decomposition operations, and relocating the branch target. According to the characteristics of VLIW on stream architecture, a two-dimension (2-D) compression scheme is put forward, where VLIW code is compressed in both vertical and horizontal directions, the horizontal decompression and code execution can be implemented in parallel, and loop entrance addresses are buffered by stack registers. The experiment results illustrate that 2-D compression scheme can resolve code expansion issue effectively. Specifically, it has achieved a 36.48% area reduction of the on-chip instruction memory and a 7.85% area reduction of the CISP system.

Key words: stream processor; VLIW; 2-D compression; parallel decoding

1 引言

流体系结构[1]作为近年来兴起的一种微体系结构,集成了大量的计算资源,并通过超长指令字 VLIW (Very Long Instruction Word, VLIW)开发指令级并行,获得了较高的性能. 一条 VLIW 指令包含固定数目的操作,与流处理器中的功能单元一一对应,但是因为程序本身的并行性有限,以及资源限制和数据相关等因素,VLIW 中并不能完全填充有效操作,而且因为 VLIW 是固定宽度的,所以需要在 VLIW 中填充空操作(No OPeration, NOP). 大量 NOP 的存在造成了 VLIW 代码体积急剧膨胀,这会占用更多的指令存储空间、浪费访存带宽以及消耗更多的能源. Imagine[1]、MASA[2]等流处理器都使用了较大容量的指令存储器,分别是 144KB 和

收稿日期:2016-05-24; 修回日期:2016-9-18; 责任编辑:梅志强
基金项目:国家自然科学基金项目(No. 61404175)

160KB,占到整个芯片面积的 11% 和 27%.

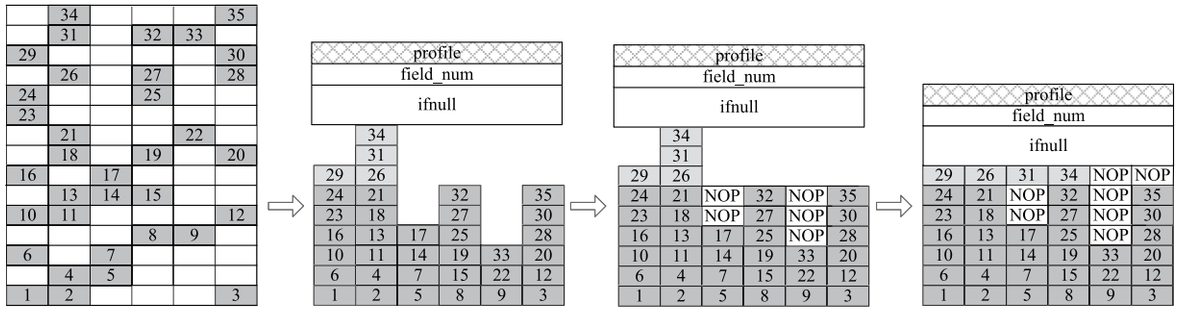
为了减小 VLIW 代码体积,提高 VLIW 的指令密度,可以采用代码压缩技术. 在数据压缩领域广泛使用的字典压缩[3]、哈夫曼编码[4]和算术编码[5]等方法,通过统计不同代码的使用概率,把使用概率高的代码用小位宽的编码替换,达到压缩代码体积的目的. 但是上述方法存在编码计算量大,解压电路复杂等问题. 特别是流处理器上的 VLIW 位宽大,导致解压出一条指令的延时较长.

本文针对流处理器上 VLIW 位宽大且指令中有较多 NOP 的特点,通过删除 NOP 来压缩代码体积. 压缩的首要目标就是如何尽可能地删除 NOP 提高压缩率. 但是随之产生了两个问题,一是压缩后的代码在执行前要先解压,所以要降低解压操作对性能的影响,二是压缩后代码之间的相对位置发生了变化,这需要重新定位分支目标指令. 因此 VLIW 压缩需要解决三个问题,一是提高压缩率,二是减少解压时间,三是目标指令重定位.

国内外针对上述问题进行了一系列研究,并提出了多种解决方案. 文献[6]针对解压缩操作对系统性能造成的影响,把一条 VLIW 指令划分为多个子域后压缩,并设计了分布式指令存储器来存储压缩后的 VLIW. 该方案的优点是不需要对压缩后的代码进行解压还原,而是根据标识信息直接执行压缩后的 VLIW. 文献[7]对每条 VLIW 指令的每个指令槽依次进行压缩,并为每条 VLIW 指令生成相应的标识信息放在指令头部. 这种方法的压缩与解压操作都是逐条指令顺序进行,当指令中的指令槽较多时,每条指令所需要的标识信息会急剧增加,这会抵消压缩效果. 文献[8]则对多个连续的 NOP 提出了一种基于指令后缀的压缩方法,即在每个子域后面增加两位标识码,说明后续有几个 NOP,然后把相应的 NOP 删除,从而达到压缩代码体积的目的. 这在 NOP 连续出现时,可以获得较好的压缩效果. 文献[9]则针对解压缩操作可能对系统性能产生影响的问题,提出了留下部分指令不压缩的方法,这样程序一启动,就可以直接执行未压缩的指令而不需要等待指令解压,从而让指令执行与解压操作并行起来. 但是留下部分指令不压缩势必会影响压缩效果,而且如何计算解压时间与指令执行时间从而划分压缩指令与非压缩指令的比例,文献中并没有给出具体的方案.

2　流体系结构上 VLIW 指令特点

流处理器上运行的流程序是由一系列称为计算核心的 kernel 程序构成,每个 kernel 包括若干条 VLIW 指令,kernel 的构成如图 1(*a*)所示. 图中一行表示的是一条 VLIW 指令,灰色块表示有效操作,白色块表示 NOP. 本文讨论的压缩即是如何把 NOP 删除,以缩减代码体积,从而减少指令存储器的面积,然后在指令执行前再解压恢复.



(*a*) 未压缩的kernel　(*b*) 垂直压缩　(*c*) 填充NOP　(*d*) 水平压缩

图1　二维压缩步骤

本文以课题组开发的流处理器 CISP(Computing-Intensive Stream Processor)为基础,在对流体系结构及运行的 VLIW 代码进行深入分析后,发现流体系结构上的 VLIW 具有如下特点:

(1)指令宽度大. 如 Imagine 流处理器的 VLIW 指令位宽为 576 位,MASA 为 640 位,CISP 为 512 位. 这是因为流处理器为了实现高性能集成了大量的功能单元,每个功能单元都对应 VLIW 中的若干位,导致 VLIW 位宽较大.

(2)VLIW 中的 NOP 数量多. 因为每个空闲的功能单元在 VLIW 中对应的都是 NOP,流处理器中大量的功能单元处理不同应用时,往往有较多的功能单元空闲,从而导致 VLIW 中的 NOP 数量较多,通过对 Reed-Solomon、H.264 等典型流应用的 NOP 进行统计(如图 2 所示),每个子域的 NOP 均在 50% 以上.

(3)特殊的分支指令. 因为流处理器把大部分的片上面积用于功能单元,所以它的控制结构相对简单,流处理器上使用的 KernelC 语言为了简化程序的控制流,只允许使用循环,而不允许使用其它条件分支结构[2].

本文针对流体系结构 VLIW 指令的上述特点,提出了二维压缩方案,在尽可能删除 NOP 提高压缩率的同时,使解压过程与代码执行可以并行,并通过缓存循环

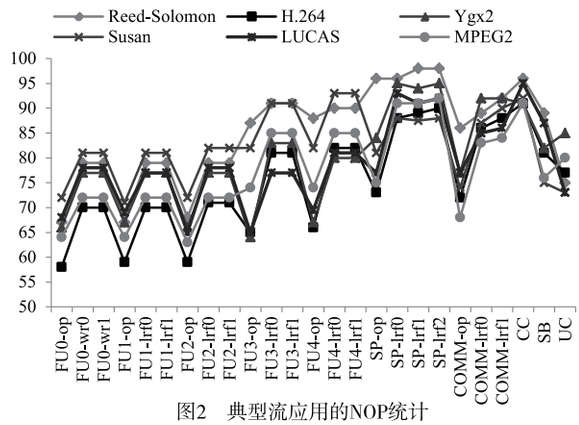


图2 典型流应用的NOP统计

入口地址的方法解决目标指令重定位问题.

3 二维压缩

本文提出的二维压缩分为三部分,分别是垂直压缩、水平压缩以及分支目标重定位.

3.1 垂直压缩

因为 VLIW 指令宽度较大,压缩前先把指令划分成若干子域,即把 kernel 分为若干列,然后把每列中的 NOP 依次删除. 简单的划分方法是把每个功能单元对应的指令作为一个子域,但是这会降低 NOP 检测出的比例,因为判断子域为 NOP 的依据是指令值全为 0,子域划分的越大,则出现所有值同时为 0 的概率越小. 所以根据指令的逻辑特征,对每个功能单元的指令信息作进一步划分.

流处理器的每个功能单元都有自己的局部寄存器(Local Register File,LRF),操作数均来自于 LRF. 基于这种结构,功能单元对应的指令格式如图 3 所示(以双操作数功能单元为例),包括操作码、操作数所在的寄存器号以及要写入数据所在的总线号和写入的寄存器号. 如果为有效操作,则必有操作码和操作数,即操作码和读寄存器字段总是同时为空或非空,可以把它们作为一个子域. 对于写操作,必须有写入的数据和写入的寄存器号,即这两个字段是相关联的,可以把它们作为一个子域. 因为写入的是后续操作需要的操作数,与当前操作无关,同时两个写操作之间亦是独立的,所以根据指令的上述逻辑特征,可以把当前操作与两个写操作划分为三个子域,如图 3 所示. 对指令信息的划分可以提高 NOP 检测出的比例,从而有利于后序的压缩操作.

图3 功能单元对应的指令信息

划分好子域后,需要逐一判定每个子域是否为 NOP. 如果子域为 NOP,则有效操作依次下移,把 NOP 排空. 并设置一个二维数组 ifnull[h][w],记录对应的子域是否为 NOP,其中 h 表示要压缩的 kernel 中所包含的 VLIW 指令条数,w 表示每条指令划分的子域个数,如果第 i 条指令的子域 j 为 NOP,则 ifnull[i][j] =0. 数组 field_num[w]记录每个子域包含的有效操作数目. 经过垂直压缩后的代码如图 1(b)所示.

3.2 水平压缩

因为每个子域的有效操作数量不等,经过垂直压缩后,每列指令不再是等高的. 为了方便指令存储,一种方法是重新填补 NOP 至所有列等高. 对 Reed-Solomon、H.264 等 6 个应用采用该方法后,填充的 NOP 和有效操作的比值如表 1 第二列所示. 可以发现,如果直接填充 NOP 至所有列等高,则填充的 NOP 几乎与有效操作相当,部分应用填补的 NOP 甚至超过了有效操作,这将大大抵消垂直压缩的效果.

基于上述原因,本文不对垂直压缩后的指令填充 NOP 至所有列等高,而是对垂直压缩后的代码进行水平压缩. 水平压缩的目的就是减少存储时要填充的 NOP.

表 1 填充 NOP 与有效操作比值

应用名称	直接填充	二维压缩
Reed-Solomon	1.37	0.18
H.264	1.04	0.36
Ygx2	1.15	0.28
Susan	0.66	0.19
LUCAS	0.95	0.30
MPEG2	0.88	0.32

对垂直压缩后的代码,设置一个参考高度 k. 低于 k 的子域,填充 NOP 至高度 k,如图 1(c)所示;高于 k 的子域,对高于 k 的部分做水平压缩. 因为填充 NOP 与水平压缩均有开销,假设要填充的 NOP 个数与需要水平压缩的有效操作个数之和为 s,确定高度 k 的原则是使 s 最小. 确定 k 值以后,水平压缩不是把经过垂直压缩后处于相同行上的有效操作做水平左移,排空掉 NOP. 因为这样压缩后,解压操作会变得复杂. 本文的策略是依次对每个子域中高于 k 的部分顺序存储. 解压时,因为每个子域中高于 k 的部分是连续存储的,可以一次解压一个子域,从而提高解压速度. 水平压缩的结果如图 1(d)所示.

经过水平压缩后,需要填充的 NOP 比例大幅降低,如表 1 第三列所示.

压缩后的指令包括 field_num、ifnull、水平压缩指令和垂直压缩指令这四类信息,为了识别不同信息,在 kernel 的头部加一个 profile 文件,用以说明四种信息的长度.

3.3 目标指令重定位

从图1(*d*)可以看到,经过二维压缩之后,不仅每条VLIW 指令的地址发生了变化,而且同一条 VLIW 中各个子域的位置也发生了变化,这使得压缩后的目标指令重定位变得困难,而且流程序属于典型的循环密集型应用,执行循环的时间占总执行时间的 70%[10,11]. 所以目标指令重定位的实现速度对流应用的执行效率有较大影响.

为了解决压缩后分支目标指令重定位问题,相关文献也进行了研究[12]. 解决的主要思想是通过构建程序控制流图(Control Flow Graph,CFG)找到分支语句所构成的环路,然后通过划分基本块打破环路. 但存在的问题是:(1)基本块的划分是基于 kernel 的逻辑结构,块的大小并不均匀,可能产生极小块,这势必会影响压缩效果;(2)分支查找表会占用片上面积,而且分支查找会影响代码执行的性能;(3)不能很好的解决分支嵌套的目标重定位.

通过进一步分析流处理器上使用的 KernelC 语言中的循环结构,发现它们可以分为两类,分别是计数循环和条件循环,如图 4 所示:

```
Loop_count (counter)        Loop_while (loopcc)
{loop body };               {loop body };
ins1;                       ins1;
……                          ……
(a) 计数循环                (b) 条件循环
```

图4　KerneC的循环结构

针对循环的这一特点,设置一个先进后出的 LEAM 模块(Loop Entrance Address Module,LEAM). 执行时如果遇到计数循环,则把循环入口指令对应的各个子域的地址存入 LEAM 中,然后寄存器的读指针指向该存储单元,写指针向下移动. 当一次循环执行完毕,读取循环计数器的值,如果需要继续执行循环,则通过读指针读取对应的循环入口地址;如果循环结束,则顺序向下执行.

如果是条件循环指令,当条件成立时,循环体的执行过程与计数循环相同;如果条件不成立,则需要跳过循环体执行 ins1 语句,因为循环体长度 l(未压缩时循环体长度)可在编译时确定,通过 ifnull 标识信息,可以计算出每个子域中有效操作个数,然后与各个子域的当前地址相加,即可以得到 ins1 语句中每个子域的地址. 因为 kernel 程序中大量使用的是计数循环,条件循环一般仅在数据输入输出时使用,所以通过设置 LEAM 模块,可以快速实现绝大部分循环及循环嵌套指令的目标重定位,而且不需要构建分支查找表,只有出现条件循环且条件不成立这一最坏情况时,需要对每个子域重新计算目标地址.

4 解压与执行

4.1 指令加载

当流处理器需要执行某个 kernel 时,先把该 kernel 的 VLIW 代码加载到片上指令存储器中. 加载的过程如下:

(1)首先读取 kernel 头部的 profile 文件,然后根据 profile 确定 field_num、ifnull、垂直压缩指令和水平压缩指令四类信息的长度,然后依次加载四类指令信息.

(2)field_num 数组加载到子域个数(Field Number,FN)寄存器存中,寄存器的位宽为 10-bit.

(3)ifnull 数组加载到子域索引寄存器(Field Index Register,FIR)中,寄存器位宽与子域个数相等.

(4)垂直压缩后的指令结构与未压缩时是一样的,可以直接加载到片上指令存储器中. 因为 VLIW 指令宽度较大,所以指令存储器分 bank(每个 bank 的宽度与子域宽度相同). 指令加载时,每个子域的指令分别加载到对应的 bank 中.

(5)水平压缩的指令,在加载到片上指令存储器之前要先解压,具体的解压过程在下一节介绍.

4.2 并行解压与执行

当垂直压缩后的指令加载完毕后,不需要等待水平解压完成即可以开始执行指令. 指令在执行时,通过如图 5 所示的地址计算单元获得每个子域在对应 bank 中的指令地址.

子域指令计数器(Field PC Register,FPCR):保存的是当前要执行的指令在对应 bank 中的偏移地址,初始值为 0.

子域基地址寄存器(Base Address Register,BAR):指令装载进片上指令存储器时,每个子域在对应 bank 中的起始地址,这个信息保存在 BAR 中.

读取第 i 条指令时,首先读取 FIR 寄存器中第 i 行的 w 个值. 如果 ifnull$[i][j]$($0 \leqslant j < w$)的值为 0,则表示指令 i 的子域 j 是 NOP,则该子域指令的读地址为全 0 值(0 地址寄存器的值为 0,与 NOP 相同);如果ifnull$[i][j]$ = 1,则表示是有效操作,把 FPCR 寄存器中的子域偏移地址与 BAR 中的子域基址相加,即可得到该子域在对应 bank 中的读地址.

在指令执行的过程中,可以同时对水平压缩后的指令进行解压,解压过程如下:

(1)依次读取 FN 寄存器中的值 field_num$[i]$($0 \leqslant i < w$),即每一个子域有效操作个数,然后与 k 值进行比较.

(2)如果 field_num$[i] > k$,则表示该子域中有效操作个数大于 k,需要对 field_num$[i] - k$ 个有效操作进行水平解压. 因为每个子域的宽度值为 field_width$[i]$,该值存储在 FW 寄存器中. 依次读取 field_num$[i] - k$ 个指

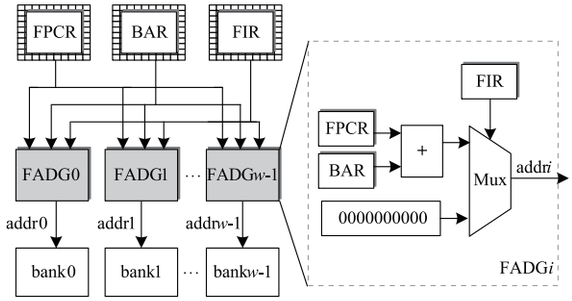


图5 地址计算单元结构图

令存储到该子域所对应的指令存储器 bank 中,bank 的写地址为 kernel 加载到该 bank 时的起始地址加 k.

(3)如果 field_num[i]≤k,则表示该子域没有经过水平压缩,则不需要做水平解压操作.

水平解压电路如图 6 所示. 其中水平解压单元(horizontal decompression unit,HDU)的主要功能是执行比较操作,把读取的 field_num[i]与 k 值比较,根据比较结果确定是否需要从片外存储器中加载指令到 banki 中.

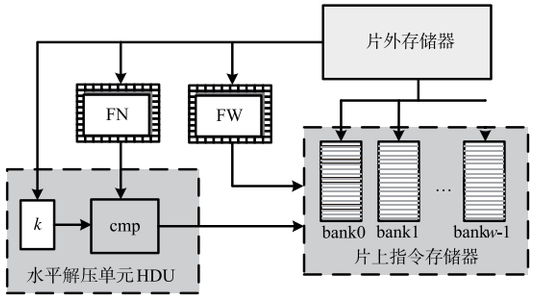


图6 水平解压单元结构图

水平解压之所以可以与指令执行并行起来,一方面是因为水平解压与指令执行分别使用不同的功能单元,两者在执行时不会冲突;另一方面在设置 k 值时限制了水平压缩操作的数目,从而限制了水平解压所需要的时间.

4.3 循环指令的执行

当 kernel 在执行过程中遇到循环指令,如果是计数循环和条件成立时的条件循环,则按如下步骤执行:

(1)把循环入口指令每个子域的地址写入 LEAM 模块,并把 LEAM 的读指针指向该存储单元,写指针向下移动,如图 7-Step1 所示.

(2)当遇到循环嵌套时,同样把内层循环的入口地址存入 LEAM,并把读写指针依次向下移动,如图 Step2 所示.

(3)当循环一次执行完毕,若循环计数器不为 0(如果是条件循环,条件成立时,循环计数器为 1),则从当前读指针指向的存储单元中读取循环入口地址,重复执行循环体;当执行到循环计数器为 0 时,表示要结束当前循环,则把 LEAM 的读写指针均向上移动,如图 Step3 所示.

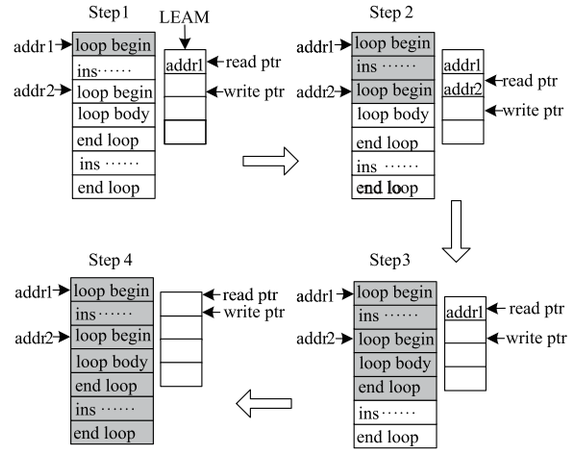


图7 循环执行示意图

(4)重复步骤(3),即可执行外层循环,当外层循环执行完毕,则 LEAM 为空,表示当前循环执行结束,如图 Step4 所示.

通过 LEAM 存储模块,即可以实现 kernel 中的循环和循环嵌套.

如果遇到条件循环指令且条件不成立的情况,则通过 ifnull 数组计算循环体中每个子域有效操作的数目来确定跳转目标指令的地址.

通过对解压与执行过程的描述可以发现,二维压缩不仅大大降低了填充的 NOP 的数目,且垂直压缩后的指令加载到片上指令存储器后即可以开始执行,与此同时可以并行执行水平解压.

5 实验结果与分析

5.1 压缩率

在 CISP 流处理器上映射 6 个流应用,然后对得到的 VLIW 指令进行二维压缩,分别得到片外指令体积与片上指令体积. 压缩率(Compression Ratio,CR)根据式(1)计算,得到的压缩率如表 2 所示:

$$CR=\frac{\text{压缩后代码体积 + 标识信息}}{\text{压缩前代码体积}}\times 100\% \quad (1)$$

经过二维压缩后片外压缩率和片上压缩率均低于 40%,这是因为垂直压缩排空了 NOP,水平压缩仅需填充少量的 NOP,且水平压缩只需要一个 10-bit 的标识信息 k. 同时,通过 LEAM 模块缓存循环入口地址大大减少了循环目标指令重定位所需要的标识信息.

将本文的压缩方法与相关文献进行了比较,结果如表 3 所示. 从表中可以看到,对于相同的流应用,二维压缩的片外压缩率几乎只有文献[6]的一半,这主要是因为二维压缩需要填充的 NOP 数目大大减少. 而对于片上压缩率,二维压缩亦优于文献[6]中的方法. 这主要是因为通过 LEAM 模块存储循环入口地址,减少

了循环指令所需要的标识信息.

表 2　几种典型流应用的二维压缩的压缩率

应用名称	片外压缩率	片上压缩率
Reed-Solomon	22.75%	21.24%
H.264	34.69%	33.36%
Ygx2	27.72%	26.88%
Susan	27.68%	26.29%
LUCAS	29.73%	28.68%
MPEG2	32.19%	30.88%

表 3　不同压缩方法的压缩率比较

	二维压缩	文献[6]	文献[7]	文献[8]
片外压缩率	29.13%	61.44%	—	—
片上压缩率	27.89%	34.58%	52.8%/57.20% *	77%/65.78% *

同时,二维压缩的片上压缩率明显优于文献[7,8]的压缩率. 但是不同应用的 NOP 的数量和分布规律均不同,为了使统计结果更具有可比性,把本文中的 6 个应用分别使用文献[7,8]中的方法进行压缩,得到的平均压缩率为带“ * ”上标所示的数据(因为文献中没有给出压缩应用的具体名称,所以无法使用二维压缩对文献中的应用进行压缩). 可以看到,针对相同的应用,二维压缩的压缩率同样优于其它文献中的方法,这主要是因为本文的二维压缩针对流处理器上的大位宽 VLIW 进行压缩,在尽可能排空掉 NOP 的同时,生成的标识信息较少,从而获得了可观的片上压缩率.

对于解压单元及地址生成单元等硬件电路采用 65nm CMOS 工艺,通过 DC 进行逻辑综合,FN、FW、BAR 和 FPCR 均采用寄存器实现,指令存储器和 FIR 使用 RAM 实现,指令压缩后代码体积大大减少,所以片上指令存储器的容量减少为原来的 50%. 综合结果如表 4 所示. 可以看到,虽然增加了 LEAM 模块、水平解压单元和地址计算单元等硬件部件,但是相对于减少的指令存储器来说,这仍然只是较小的代价,指令存储单元的面积减少了 36.48%,并把整个 CISP 系统的面积减少了 7.85%.

表 4　压缩前后硬件规模对比(单位:逻辑门)

	压缩前	压缩后		面积节省
整个系统	833901	768437		7.85%
指令存储单元	179456	总数	113993	36.48%
		LEAM	9720	
		水平解压	4241	
		地址计算	10304	
		存储器	89728	
指令存储占系统面积	21.52%	14.83%		

5.2　代码执行性能

为了评估二维压缩对代码执行速度的影响,通过代码压缩前后执行速度的加速比来进行衡量,结果如图 8 所示.

可以看到 6 个应用的加速比均在 0.97 ~ 1.02 之间,这说明二维压缩没有因为解压而降低代码的执行速度. 产生这一结果的原因是:

(1)对 VLIW 进行二维压缩后,代码体积减为原来的 30%左右,这会缩短代码的加载时间.

(2)二维压缩方案中代码执行与水平解压是并行的,不需要等待解压操作完成再开始代码的执行. 但是如果需要水平解压的操作较多,可能在执行到水平压缩后的代码时,水平解压操作未完成,此时需要等待. 比如 RS 和 Ygx2 两个应用,因为需要水平解压的操作较多,从而延长了代码执行时间.

(3)流应用中循环较多,即代码经过一次加载和解压后,要循环执行多次. 相对于较长的代码执行时间,加载和解压对执行速度的影响并不明显. 本文只是针对代码体积进行了压缩,没有改变代码执行语义. 而对于循环指令,通过缓存循环入口地址的方法,没有增加循环的执行时间. 所以执行时间在压缩前后没有明显变化.

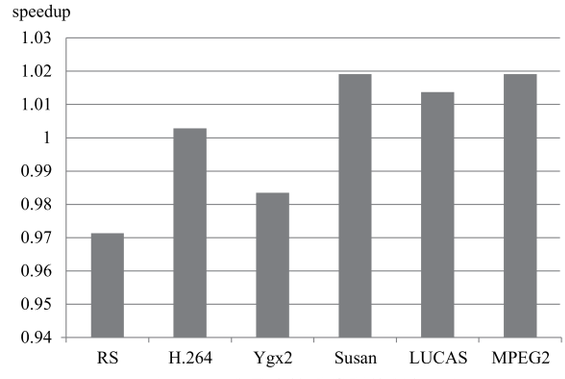


图8　二维压缩前后代码执行加速比

所以二维压缩后代码的执行时间没有明显增加,即二维压缩在获得较好的压缩率的同时,没有降低代码的执行速度.

6　总结

流体系结构上的 VLIW 一直存在因为代码稀疏导致的代码体积膨胀问题. 本文在对已有的压缩方法进行分析总结的基础上,针对流体系结构上的 VLIW 指令特点提出了二维压缩方法. 二维压缩使水平压缩可以与执行过程并行,减少了解压操作对性能产生的影响;通过设计循环入口寄存器,解决了因为压缩导致的复杂的目标指令定位问题. 在压缩过程中,发现每个子域

的 NOP 分布规律并不相同,所以下一步的工作可以针对每个子域的不同特点改进压缩方法和设计指令存储器,从而进一步提高压缩率和降低片上指令存储器的面积.

参考文献

[1] RIXNER S. Stream Processor Architecture[M]. Bostyon, MA:Kluwer Academic Publishers,2002.

[2] 张春元,文梅,伍楠,等. 流处理器研究与设计[M]. 北京:电子工业出版社,2009.

[3] LEE J, YOUN J M, Lee J, et al. Dynamic operands insertion for VLIW architecture with a reduced bit-width instruction set[A]. Proc of the 26th International Parallel & Distributed Processing Symposium (IPDPS)[C]. Shanghai, China:IEEE,2012. 119 – 130.

[4] AZEVEDO W R, MORENO E D. Code compression using Huffman and dictionary-based pattern blocks[J]. IEEE Latin America Transactions,2015,13(7):2314 – 2321.

[5] 刘文松,朱恩,王健,等. JPEG2000 算术编码器的算法优化和 VLSI 设计[J]. 电子学报,2011,39(11):2486 – 2491.
LIU Wensong, ZHU En, WANG Jian, et al. Optimization algorithm and VLSI design of the arithmetic coder in JPEG2000[J]. Acta Electronica Sinica, 2011, 39(11): 2486 – 2491. (in Chinese)

[6] 管茂林,何义,杨乾明,等. 流体系结构指令存储器优化设计研究[J]. 电子学报,2012,40(7):1379 – 1386.
GUAN Maolin, HE Yi, YANG Qianming, et al. Optimized design research of instruction memory for stream architecture[J]. Acta Electronica Sinica, 2012, 40(7): 1379 – 1386. (in Chinese)

[7] 姬忠宁,陈迅,等. 基于指令前缀的专用 VLIW 压缩技术研究与实现[J]. 电子技术应用,2013,39(4):22 – 25.
JI Zhongning, CHEN Xun, et al. Research and implementation of VLIW compressing technology based on instruction prefix[J]. Application of Electronic Technique, 2013, 39(4):22 – 25. (in Chinese)

[8] JIN Taisong, AHN Minwook, YOO Donghoon, et al. Nop compression scheme for high speed DSPs based on VLIW architecture[A]. Proc of IEEE International Conference on Consumer Electronics (ICCE)[C]. Las Vegas, USA: IEEE,2014. 304 – 305.

[9] BONNY T, HENKEL J. LICT: Left-uncompressed instructions compression technique to improve the decoding performance of VLIW processors[A]. Proc of ACM/IEEE Design Automation Conference[C]. San Francisco, California, USA: IEEE, 2009. 903 – 906.

[10] BACHIR M, BRAULT F, et al. Loop unrolling minimization in the presence of multiple register types: a viable alternative to modulo variable expansion[A]. Proc of High Performance Computing and Simulation (HPCS)[C]. Istanbul, Turkey: IEEE, 2011. 207 – 214.

[11] VELKOSKI G, GUSEV M, RISTOV S. The performance impact analysis of loop unrolling[A]. Proc of Information and Communication Technology, Electronics and Microelectronics (MIPRO)[C]. Opatija, Croatia: MIPRO, 2014. 307 – 312.

[12] TU Ji, WANG Zilong, ZHENG Mesong, et al. Code compression in embedded processors using multi-dictionary and pattern blocks[A]. Proc of the 28th Canadian Conference on Electrical and Computer Engineering[C]. Halifax, Canada, IEEE, 2015. 361 – 365.

作者简介

李功丽 女,1981 年生于河南信阳. 信息工程大学博士生,主要研究方向为计算机体系结构、流处理器、高性能计算.
E-mail:ligl522@163.com

戴紫彬 男,1966 年生于河南商丘. 信息工程大学教授,博士生导师. 研究方向为专用芯片设计、可重构芯片、可重构 SoC 设计.

徐进辉 男,1978 年生于江西宁都. 博士,讲师,主要研究方向可重构计算、计算机体系结构研究.

王寿成 男,1992 年生于甘肃金昌. 信息工程大学硕士生,主要研究方向计算机体系结构.

朱玉飞 男,1990 年生于江苏淮安. 信息工程大学硕士生,主要研究方向为专用芯片设计.

李 丹 男,1978 年生于河北唐山. 工程师,研究方向为专用芯片设计.