基于RISC-V的嵌入式多指令集处理器设计及实现

成元虎,黄立波,崔益俊,马 胜,王永文,隋兵才
(国防科技大学计算机学院,湖南长沙 410073)

摘 要: 软件生态是限制RISC-V指令集架构发展的主要因素之一.让RISC-V处理器可以直接运行ARM Thumb二进制代码能在一定程度上缓解其在嵌入式领域中的软件生态问题.本文基于二进制翻译,通过硬件支持ARM Thumb的标志位、分支指令、条件执行,在RISC-V处理器上以较低的面积和功耗开销实现了对ARM Thumb程序的支持并获得了较好的性能.通过运行Embench基准程序套件,该处理器翻译运行ARM Thumb程序的平均性能能够到达直接运行RISC-V程序性能的75.5%.相较于仅使用二进制翻译支持ARM Thumb,该处理器运行ARM Thumb程序的性能提升了3.1倍,面积开销则下降了7.8%.

关键词: RISC-V; ARM Thumb; 体系结构; 多指令集; 微处理器; 二进制翻译
中图分类号: TP302 **文献标识码:** A **文章编号:** 0372-2112(2021)11-2081-09
电子学报URL: http://www.ejournal.org.cn **DOI:** 10.12263/DZXB.20201350

Design and Implementation of Embedded Multiple-ISA Processor Based on RISC-V

CHENG Yuan-hu, HUANG Li-bo, CUI Yi-jun, MA Sheng, WANG Yong-wen, SUI Bing-cai
(*College of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan* 410073, *China*)

Abstract: Software ecology is one of the most critical factors restricting the development of RISC-V instruction set architecture. Allowing the RISC-V processor to directly run the ARM Thumb binary code can solve its software ecological problem in the embedded field to a certain extent. Based on the binary translation, this article realizes support for the ARM Thumb program and achieves comparable performance on the RISC-V processor with the lower area and power consumption overhead by using hardware to optimize ARM Thumb flag bits, branch instructions, and conditional execution. For the Embench benchmark suite, the average performance of the processor running ARM Thumb programs can reach 75.5% of directly running RISC-V programs. Compared with using only binary translation to support ARM Thumb, hardware optimization performance is improved by 3.1 times and hardware overhead is reduced by 7.8%.

Key words: RISC-V; ARM Thumb; architecture; multiple-ISA; microprocessor; binary translation

1 引言

软件生态缺乏是新指令集架构(Instruction Set Architecture, ISA)发展的主要障碍之一.通过二进制翻译(Binary Translation, BT)[1]技术将一个成熟指令集架构的二进制代码翻译为新指令集的二进制代码,可以在一定程度上减小这个问题带来的影响.

凭借开源和设计上的优势,新兴的RISC-V指令集发展迅速,其操作系统、编译器等基础软件已具有一定的规模,但应用软件依然缺乏.让RISC-V处理器可以直接运行基于ARM Thumb编译的程序能在一定程度上缓解其在嵌入式领域中的软件生态问题.

在桌面和服务器系统中,软件二进制翻译系统是实现程序跨指令集运行最常用的方法.例如,苹果公司两代Rosetta[2]系统分别实现了程序从Power PC到X86以及从X86到ARM的跨指令集执行;FX!32[3]实现了X86程序运行在Alpha处理器上.

然而,将软件二进制翻译系统运用于嵌入式系统中存在一些局限性.首先,由于嵌入式设备对芯片的面积和功耗都有严格的要求,嵌入式系统中常常没有足够的存储空间和环境来运行一个动态二进制翻译系统;其次,由于优化空间有限,动态二进制翻译运行非

收稿日期:2020-11-30;修回日期:2021-03-03;责任编辑:覃怀银
基金项目:国家自然科学基金(No. 62090023, No. 61872374)

原生指令集程序的性能仅能达到原生指令集程序性能的10%~50%[4,5]，性能差距较大. 最后，虽然静态二进制翻译系统拥有更大的优化空间[6]，但它需要在运行程序之前将所有指令翻译为目标指令集，例如LLBT[7].

另一种研究较少的方法是通过硬件支持多指令集，即多指令集处理器. 目前，多指令集处理器在嵌入式领域中主要用于支持存在包含关系的两个指令集. 因为较小的指令集使用更小的指令长度编码，能够减小程序的体积，从而节约存储空间[8]. 比如ARM公司Cortex系列部分处理器同时支持32位ARM指令集和16位Thumb指令集.

显然，硬件支持多个指令集不可避免地会带来额外的面积和功耗开销. 多指令集处理器有多种实现方法[9~13]，对于实现程序跨指令集运行而言，主要考虑硬件二进制翻译. 已有研究显示，为了达到较好的性能其带来了超过50%的硬件开销[9,14]. 因此，将其运用到嵌入式中需要研究降低硬件开销的方法.

另外，由非原生指令到原生指令的翻译比也是硬件二进制翻译关注的重点. 因为较大的翻译比会导致运行非原生指令集程序的性能远低于运行原生指令集程序的性能. 例如，直接将ARM指令翻译为MIPS指令时，指令数量会增加2~4倍[14]，导致MIPS处理器翻译运行ARM程序的性能较低.

本文基于硬件二进制翻译的方式在RISC-V处理器上实现了对ARM Thumb指令集程序的支持. 文章解决了如何将ARM Thumb指令集的指令和寄存器映射到RISC-V的RV32I基础指令集上；提出了硬件支持ARM Thumb标志位、分支指令、条件执行的方式以提高RISC-V处理器翻译运行ARM Thumb程序的性能. 基于开源的RISC-V处理器Zero-riscy[15]我们实现了一个同时支持RISC-V和ARMv6-M(ARM Thumb指令集的一个子集)指令集的多指令集处理器. 实验结果显示，Zero-riscy运行由ARM Compiler编译的ARMv6-M代码的性能超过直接运行GCC编译的RISC-V代码的70.0%.

2 支持ARM Thumb

2.1 硬件二进制翻译

在软件二进制翻译系统中，源指令集架构程序代码的代码发现、代码定位、自修改代码是需要解决的关键问题[7,16]. 但基于硬件二进制翻译的多指令集处理器则不存在这些问题，因为所有源代码在内存中的内容和地址都没有任何修改. 图1是基于二进制翻译的多指令集处理器的逻辑框图，其仅在传统单指令集处理器的基础上增加了一个二进制翻译器，用于将非原生指令翻译为原生指令，并完成部分预译码的功能.

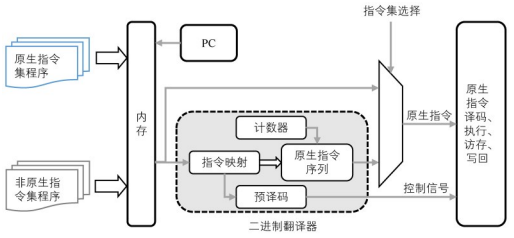

图1 基于二进制翻译的多指令集处理器逻辑框图

实际上，在传统处理器中有两个位置可以放置硬件二进制翻译器. 一个位置是在取指单元和存储器之间，此时，取指单元得到的指令是经过翻译的原生指令，这种方式的优点是对传统微处理器结构没有任何影响，缺点则是增加了程序计数器的复杂度. 这种方式适用于在一个支持较大指令集的处理器中支持一个较小的子指令集，因为总能将后者中的一条指令翻译成前者中的一条指令，避免了上述缺点.

另一个位置则是在取指单元和译码单元之间. 在将指令送入译码单元之前将非原生指令翻译为原生指令. 这种方式的缺点在于对传统微处理器的结构有一定的修改，但优势在于对程序计数器的控制更简单，当一条非原生指令翻译为多条原生指令时，只需暂停取指单元即可.

不同于存在包含关系的两个指令集，两个独立的指令集间的指令翻译，普遍存在一条指令翻译为多条指令的情况. 因此，更适合将二进制翻译器放在取指单元和译码单元之间.

2.2 翻译ARM Thumb

二进制翻译器的主要功能是将非原生指令翻译为原生指令，即为一条ARM Thumb指令找到一条或多条RISC-V指令来完成相同的功能. 为了完成ARM Thumb指令的翻译，二进制翻译器需要完成两个工作：指令映射和寄存器映射.

指令映射的主要目标是使用原生指令正确地实现非原生指令的功能，指令映射的质量则决定了多指令集处理器翻译执行非原生程序的性能. ARM Thumb和RISC-V都属于精简指令集，它们的指令功能相似，但是也存在一些细微的差别. 接下来，我们将以加法指令为例说明如何实现指令映射.

ARM Thumb指令集中的ADDS R_d, R_n, R_m和RISC-V指令集中的ADD R_d, R_n, R_m的主要功能都是将R_n和R_m寄存器中的内容相加，结果保存到寄存器R_d中，但ARM Thumb的ADDS指令还会额外改变ARM Thumb架构中的标志位，以供后面的指令使用；而RISC-V的ADD指令则没有该作用. 因此，只使用RISC-V的ADD指令不能正确地实现ARM Thumb ADDS指令的所有

功能.

指令序列1展示了由ARM Thumb ADDS指令翻译得到的RISC-V指令. 其中,第一条RISC-V ADD指令用于实现ARM Thumb ADDS指令的加法功能;中间7条指令用于完成ARM Thumb的标志位判断及保存[17];最后一条指令用于将加法的结果保存到目标寄存器R_d中. 之所以需要最后一条指令,是因为ADDS指令存在R_d和R_n或R_m相同的情况,如果直接将结果保存到R_d中,则可能会将源操作数覆盖,影响标志位的判断.

寄存器方面,ARM Thumb包括13个通用寄存器(R_0~R_{12}),1个栈指针寄存器(SP, R_{13}),1个链接寄存器(LR, R_{14})和1个程序计数器(PC, R_{15}),它们都是32位寄存器;而RISC-V的RV32I基础指令集架构中包含32个通用寄存器,以及一个专门的程序计数器. 因为RISC-V RV32I指令集架构中的逻辑寄存器数量多于ARM Thumb,因此无需借助内存就可以实现寄存器映射.

因为RISC-V中的R_0寄存器始终为0且无法修改,所以不能直接将ARM Thumb的R_0寄存器映射到RISC-V的R_0寄存器. 因此,我们将ARM Thumb的寄存器R_0~R_{12}映射到RISC-V的R_{16}~R_{28},SP映射到R_{29},LR映射到R_{30},PC则映射到R_{31}. 也就是说,ARM Thumb寄存器编号加16即为相应的RISC-V寄存器编号.

指令序列1 ARM Thumb ADDS翻译得到的RISC-V指令

```
01  ADD R15, Rn, Rm      // 执行加法
02  SLTI R1, R15, 0      // 判断并保存N标志位
03  // 判断并保存Z标志位
04  SLTU R2, R0, R15
05  XORI R2, R2, 1
06  // 判断并保存C标志位
07  SLTU R3, R15, Rn
08  // 判断并保存V标志位
09  SLTI R5, Rn, 0
10  SLT R6, R15, Rm
11  XOR R4, R5, R6
12  // 将结果保存到目的寄存器中
13  ADDI Rd, R15, 0
```

2.3　条件标志位

ARM Thumb指令集中的大多数指令需要改变或使用条件标志位,这些标志位包括:符号标志(N),零位标志(Z),进位标志(C)以及溢出标志(V). 如前所述,ARM Thumb ADDS指令因为会改变N、Z、C、V标志位而需要多条RISC-V指令来完成. 因此,如何处理标志位对翻译运行ARM Thumb程序的性能有极大的影响.

ARM Thumb标志位可以通过软件或硬件实现. 正如指令序列1中所示,软件方式共需要7条指令完成4个标志位判断及保存,其中N、Z、C、V分别保存在R_1~R_4寄存器的最低位中. 硬件方式则需要在RISC-V处理器中添加专门的标志位判断和保存逻辑,在执行需要改变标志位的指令时,自动完成标志位判断和保存. 虽然硬件方式会带来额外的硬件开销,但我们认为它依然是更好的实现方式. 因为其不需要额外的指令进行标志位的判断与保存,能够大大减少由ARM Thumb指令翻译得到的RISC-V指令条数的数量.

2.4　分支指令

对于RISC-V这类没有标志位的指令集,分支指令是通过直接比较两个操作数的大小决定是否跳转的;而对于ARM Thumb这类支持硬件标志位的指令集,则是通过先将两个操作数进行某种运算,将运算产生的标志位保存到对应的标志位寄存器中,再根据运算产生的标志位决定是否跳转.

和标志位的产生及保存一样,也可以通过软件和硬件的方式来实现ARM Thumb标志位的比较从而实现分支指令. 在软件方式下,如果标志位被存储在一个专门的状态寄存器中,那么一条ARM Thumb分支指令可能需要9条RISC-V指令来完成标志位比较及分支跳转. 指令序列2是用来实现ARM Thumb有符号大于则跳转指令(其需要判断N标志位是否等于V标志位,Z标志位是否等于0)的RISC-V指令序列. 虽然将标志位存储在RISC-V的通用寄存器中可以减少其所需的指令条数,但其依然需要4条RISC-V指令才能完成.

硬件方式则需要在RISC-V处理器中添加根据ARM Thumb条件码完成标志位比较的硬件逻辑. 在不增加RISC-V自定义指令的情况下,可以将RISC-V的BEQ指令(判断两个寄存器中的值是否相等,如果相等则跳转)的R_{SI}域用来保存ARM Thumb的条件码. 然后,在执行单元中通过硬件逻辑根据条件码和标志位判断分支是否发生. 这样,所有ARM Thumb分支指令都只需要一条RISC-V BEQ指令就可完成.

考虑到分支指令在程序中所占的比例,如果每条分支指令都需要多条指令实现,将会对程序的性能有较大的影响,因此,有必要添加这部分硬件逻辑来减少ARM Thumb分支指令的翻译比.

2.5　条件执行

ARM Thumb和RISC-V指令集架构另一个显著区别在于ARM Thumb支持条件执行,而RISC-V不支持. 与32位ARM指令集架构不同,在ARM Thumb指令集中有一条专门的条件执行的指令IT(If Then),每条IT指令之后有一个IT block,IT block中的指令就是条件执行的指令. 如果一条指令的执行条件不满足,那么这条指令将不会被执行,而会作为一条空转指令(NOP). ARM Thumb中有一个8位的寄存器EPSR. IT用来支持

指令序列2 软件实现ARM Thumb有符号大于跳转指令的RISC-V指令序列

```
1  CSRRCI R1, 0x20c, 0   // 加载标志位到通用寄存器
2  ANDI R2, R1, 0001b    // 获取V标志位
3  SRLI R3, R1, 3        // 获取N标志位
4  //N==V? R5=0 : R5=1
5  SUB R5, R2, R3
6  SLTU R5, R0, R5
7  // 获取Z标志位
8  ANDI R4, R1, 0100b
9  SRLI R4, R4, 2
10 // (N==V&&Z==0)? R5=0 : R5!=0
11 ADD R5, R5, R4
12 BEQ R5, R0,imm        // 如果R5==0分支跳转
```

条件执行，这个寄存器的高3位用来保存执行条件的高3位，低5位则用来说明IT block中的指令条数（最多可以有4条），以及每条指令执行条件的最低位．每执行完一条指令，EPSR.IT的低5位左移一位，当低5位全为0时，表示当前指令不在IT block中，这条指令始终被执行．

假设标志位已经通过硬件实现，那么在RISC-V处理器中实现条件执行最直接的方法就是在译码的时候判断被译码的指令是否满足执行条件，如果不满足那么就将其译码为NOP指令．但是，这会产生一个问题，如果一条IT block中的ARM Thumb指令需要多条RISC-V指令来完成，而恰恰这条指令的执行条件不满足，那么这些翻译得到的RISC-V指令依然需要全部送入译码部件中完成译码并执行一条空转指令，这就会导致大量的流水线周期被浪费．

为了避免这一浪费，可以将执行条件的判断逻辑放在二进制翻译器中．在一条ARM Thumb指令被翻译成RISC-V指令之前，先判断这条指令是否满足执行条件，只有满足执行条件的指令才进行翻译，否则按照NOP指令处理．通过这一改变，所有不满足执行条件的指令都只会产生一个空转周期．

3 实验

在本节中，基于开源的RISC-V处理器Zero-riscy[15]实现了一个支持ARMv6-M指令集架构的多指令集处理器．Zero-riscy是一个拥有两段流水线的32位顺序处理器核，支持RISC-V的I、M、C和E标准扩展；ARMv6-M则是Cortex-M0处理器使用的指令集架构，是ARM Thumb指令集的一个子集，但不支持条件执行．

3.1 支持ARMv6-M和条件执行

在Zero-riscy的取指单元和译码单元之间加入了一个二进制翻译器，并修改了其算术逻辑单元的实现，通过硬件的方式实现了ARM Thumb标志位及分支指令的判断．在使用硬件方式实现标志位及分支指令时，二进制翻译器除了完成指令的翻译外，还需要承担部分的预译码功能．图2是添加二进制翻译器后Zero-riscy的架构．

表1 软件和硬件两种方式实现标志位和分支指令翻译各类ARMv6-M指令所需的RISC-V指令条数

指令类别	所需RISC-V指令数量	
	软件方式	硬件方式
Move	1/2/4	1/2
Add	1/2/3/8/9/16	1/2/3
Subtract	2/17/18	1/2
Multiply	4	1
Compare	8/16/17	1/2
Logical	4/5	1/2
Shift	8/9	1
Rotate	10	6
Load	1/2/3/*n*	1/2/3/*n*
Store	1/2/*n*	1/2/*n*
Pop/Push	*n*	*n*
Branch	1/4	1
Extend	2	2
Reverse	5/11	5/11

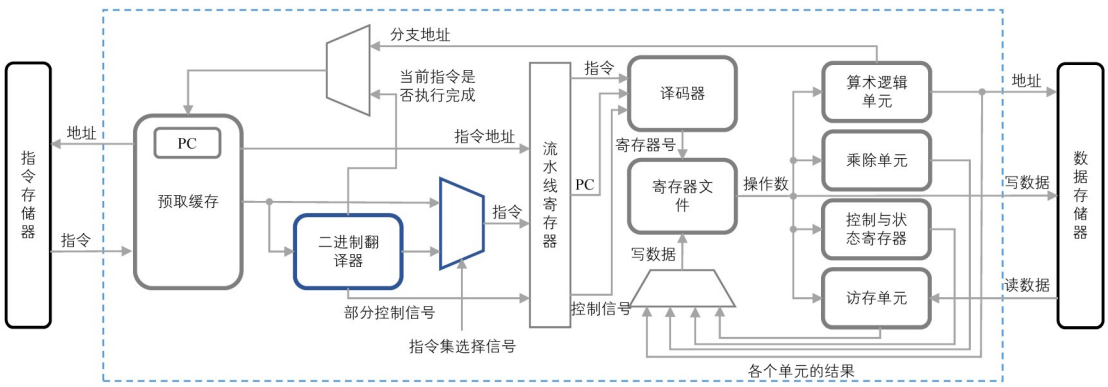

图2 添加二进制翻译器后的Zero-riscy处理器核的架构逻辑框图

如前所述,ARM Thumb指令翻译到RISC-V指令的翻译比是影响Zero-riscy处理器翻译运行ARM Thumb程序性能的关键因素,而使用硬件方式实现标志位及分支指令能够大幅降低翻译比. 表1列出了翻译ARMv6-M指令集中主要指令在软件和硬件两种标志位及分支指令实现方式下所需RISC-V指令的条数. 其中,同一类指令的不同指令可能需要不同的指令条数,比如Move类指令中需要操作程序计数器的指令需要一条额外的AUIPC指令将程序计数器的保存到通用寄存器中.

虽然ARMv6-M指令集本身并不支持条件执行,但是我们基于其支持的指令在Zero-riscy上实现了条件执行的硬件逻辑,用于对我们提出的条件指令的优化方法进行评估.

3.2　性能

我们利用Dhrystone、CoreMark以及Embench基准测试程序对Zero-riscy的性能进行了评估. 通过运行这三个基准程序的RISC-V二进制代码和ARM Thumb二进制代码,能够对Zero-riscy运行RISC-V程序和ARM Thumb程序的性能有一个较为完备的认识. 其中,运行的RISC-V代码由GNU GCC for RISC-V编译器编译产生,ARM Thumb代码则由ARM Compiler编译器编译产生. 这两个编译器是相应架构中最先进的编译器,编译过程中使用性能优先的-O3级优化. 因此,可以认为在本实验中运行的代码都是对应架构性能最高的代码.

Zero-riscy运行Dhrystone和CoreMark程序的性能结果如图3所示,性能按运行RISC-V程序的性能归一化展示. 仅使用二进制翻译支持ARMv6-M时,Zero-riscy翻译运行Dhrystone和CoreMark的ARM Thumb代码的性能分别为直接运行RISC-V代码性能的32.3%和19.7%;使用硬件实现ARM Thumb的标志位和分支指令后,运行Dhrystone和CoreMark的ARMv6-M代码的性能都有较大的提升,达到了70.9%和68.7%,分别提升了2.2倍和3.5倍.

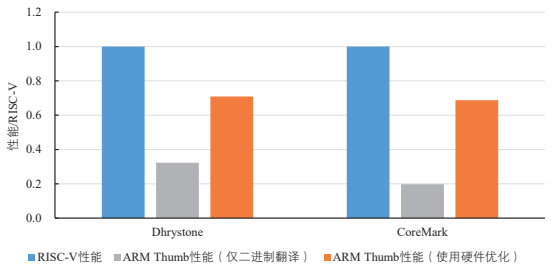

图3　Zero-riscy运行Dhrystone和CoreMark基准程序的性能

运行Embench测试套件的性能如图4所示. 仅使用二进制翻译支持ARMv6-M时,翻译运行Embench测试套件中性能最好的基准程序是matmult-int,为运行RISC-V代码的58.4%;最差的则是aha-mont64,仅为RISC-V的8.9%. 整体上,Embench中19个程序的平均性能仅为直接运行RISC-V代码的24.5%,且只有1个程序翻译执行的性能超过了直接运行RISC-V代码的50%.

使用硬件实现标志位和分支指令后,Zero-riscy翻译运行ARM Thumb程序相对于直接运行RISC-V程序性能最好的基准程序是cubic,达到了RISC-V性能的131.3%;最差的是aha-mont64基准,为RISC-V性能的36.0%,超过了仅使用二进制翻译方法的平均水平. 整体上,Embench的19个程序中,有3个程序翻译执行ARM Thumb代码的性能超过了直接运行RISC-V代码,2个程序超过90%,4个程序超过70%,7个程序的性能在50%~70%,3个程序的性能低于50%. 平均而言,Zero-riscy翻译执行ARM Thumb程序的性能是执行原生RISC-V程序的75.5%,是仅使用二进制翻译的3.1倍.

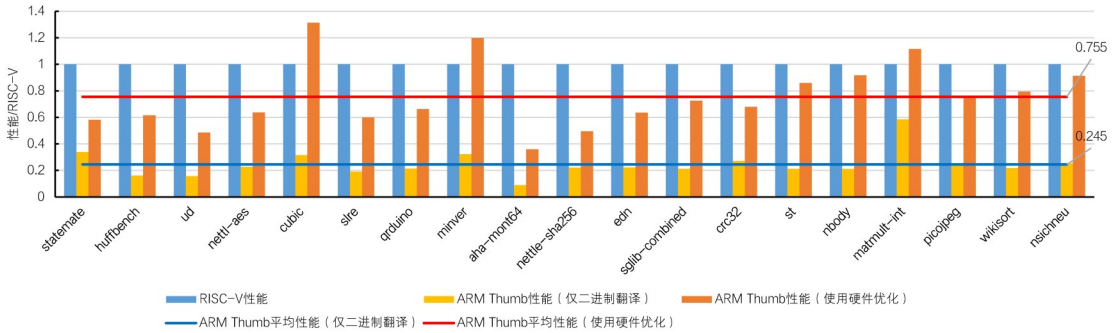

图4　Zero-riscy运行Embench测试套件的性能

为了找到翻译执行性能的差异的原因,我们统计了执行Embench测试套件的指令数量. 图5是Zero-riscy分别执行ARM Thumb和RISC-V的Embench基准程序的指令数量按照RISC-V程序指令数量归一化后的结果. 显然,仅使用二进制翻译时,翻译执行ARMv6-M程序的指令数量上升明显,导致了其性能较低.

ARM Compiler编译的ud程序的指令数量是GCC编译的RISC-V程序的3.2倍,翻译后分别为17.0倍和

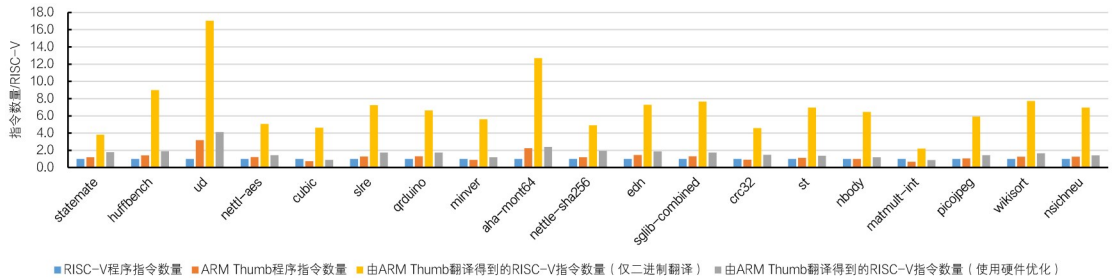

图5 Zero-riscy运行Embench基准程序的指令数量

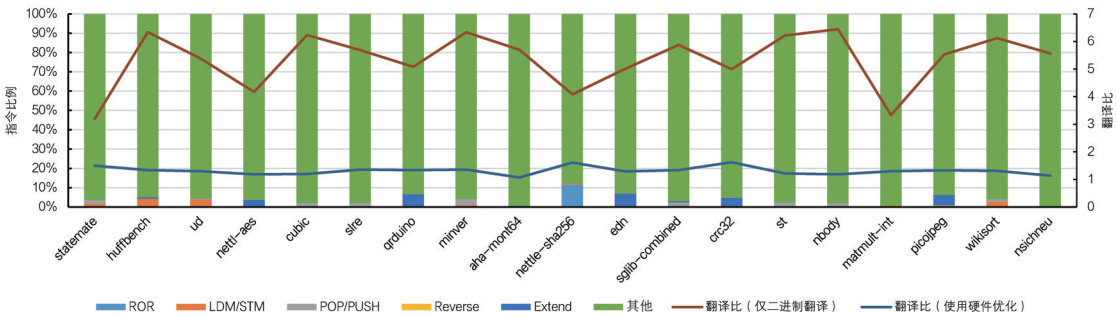

图6 由ARM Compiler编译Embench基准程序得到的代码中各指令类型所占比例及翻译运行该代码的翻译比

4.1倍，最终导致翻译执行ARM Thumb代码的性能较差. 翻译执行效果最差的aha-mont64程序翻译执行的指令数量差距则仅次于ud. 仅使用二进制翻译时，matmult-int基准翻译得到的RISC-V指令数量和直接运行RISC-V程序的指令数量差距最小，因此翻译运行它的性能下降最少.

图5还显示了不同的测试程序的翻译比并不相同，因此我们统计了由ARM Compiler编译的Embench测试套件中所有程序中不同指令所占的比例，如图6所示. 其中，两条折线分别表示仅使用二进制翻译和使用硬件实现标志位与分支指令的翻译比，使用硬件优化后的翻译比相较于仅使用二进制翻译有较大幅度的下降，平均翻译比从5.3下降到1.3.

由表1可知，ROR、LDM/STM、POP/PUSH、Reverse等指令在使用硬件优化标志位和分支指令后也具有较大的翻译比. 使用硬件优化后，Embench中翻译比最大的是nettle-sha256，因为它的代码中ROR指令占比较高. 翻译比最低的则是aha-mont64程序，大约为1.07，因为它的代码里面上述指令所占比例很小；但是，由于其编译得到ARM Thumb指令数量是RISC-V的2.2倍，所以即使有较低的翻译比，其性能下降依然较多.

最后，使用指令序列3的指令对支持条件执行的性能进行了评估. 评估的代码中共有3个IT Block，它们使用不同的执行条件和包含不同的指令类别. 其中，Block 1和Block 3中的指令以Z标志位作为执行条件，Block 2中的指令则以C标志位作为执行条件. 之所以使用指令序列3中的代码评估两种实现条件执行方法的性能，是因为3个IT Block中包含的指令在翻译比上有明显的区别，能够较为全面地说明两种方法在面对不同翻译比指令的性能差别. 由表1可知，Block 1到Block 3中的指令翻译得到的RISC-V指令数量逐渐增加. 值得注意的是，Block 2中的CMPCS指令（比较指令）会改变标志位，其余指令则不会.

在两种支持条件执行的方法下，执行每个IT Block

指令序列3 用于评估条件执行性能的指令

```
1   . . .                    // 使Z=1
2   // ——IT Block 1——
3   ITETE EQ                 // 设置IT Block
4   MOVEQ R3, #1             // 执行
5   MOVNE R3, #0             // 不执行
6   LSREQ R3, R3, #8         // 执行
7   NEGNE R0, R0             // 不执行
8   // ——IT Block 1——
9   . . .                    // 使C=1
10  // ——IT Block 2——
11  ITTE CS                  // 设置IT Block
13  CMPCS R2, R3             // 执行并使C=0
14  ADRCS R5, #17            // 不执行
15  SUBCC R4, R3, #4         // 执行
16  // ——IT Block 2——
17  . . .                    // 使Z=0
18  // ——IT Block 3——
19  ITE NE                   // 设置IT Block
20  RORNE R3, R2             // 执行
21  REV16EQ R3, R2           // 不执行
22  // ——IT Block 3——
23  . . .
```

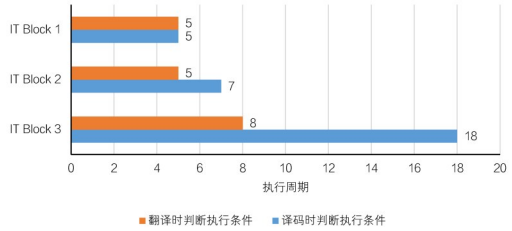

图7　不同条件执行实现方式运行指令序列3的性能

的时钟周期如图7所示. 在执行IT Block 1时,两种方法使用了相同的时钟周期,因为在Block 1中,所有的ARMv6-M指令都能被翻译成1条RISC-V指令,因此在条件不满足时两种方法都只会产生一次空转周期;而在执行Block 2和Block 3时,在翻译时判断执行条件即在二进制翻译器中实现判断逻辑使用了更少的时钟周期,因为在Block 2和Block 3中不满足执行条件的指令需要翻译成多条RISC-V指令. 如REV16EQ指令(在半字内交换字节指令)将会被翻译成11条RISC-V指令,因此,在译码时判断执行条件会产生11个空转周期,而在翻译时判断只会产生1个空转周期,最终导致前者比后者多消耗10个周期. 此时,在二进制翻译器中实现ARM Thumb的条件执行逻辑具有明显的性能优势.

3.3　面积和功耗

为了统计Zero-riscy支持ARMv6-M的面积开销,我们使用综合工具在某先进工艺下对Zero-riscy进行了综合. Zero-riscy各功能单元及总面积如表2所示. 为支持ARMv6-M,Zero-riscy总面积从32802.7 μm^2分别增加到37643.1 μm^2和37266.1 μm^2,增加了14.8%和13.6%. 使用硬件实现标志位和分支指令的方法相较于仅使用二进制翻译带来的面积开销下降了7.8%.

仅使用二进制翻译支持ARMv6-M时,增加的面积开销几乎全部来自将翻译ARM Thumb指令到RISC-V指令的二进制翻译器和取指(IF)单元,因为,这种方法除了在控制与状态寄存器单元(CSR)中增加了一些新的寄存器外,其余功能单元并没有任何变化. 而使用硬件优化后,总面积开销的77.5%来自二进制翻译器(BT),因为我们使用硬件对ARM Thumb的标志位和分支指令进行了实现,对Zero-riscy的译码单元(ID)、执行单元(EX)、控制与状态寄存器都略有修改,因此这些单元的面积也有所增加. 但仅使用二进制翻译时,二进制翻译器的逻辑更加复杂,导致其最终的面积开销更大.

以Dhrystone为例,在100MHz频率及0.81V电压下,我们统计分析了Zero-riscy支持ARMv6-M前后运行程序的功耗信息,结果如表3所示. 仅使用二进制翻译支持ARMv6-M时,Zero-riscy运行RISC-V代码的功耗上升了7.4%,功耗增加的主要来源是二进制翻译器带来的静态功耗;翻译运行ARMv6-M代码的功耗相对于运行RISC-V代码上升了11.0%,相对于原始的Zero-riscy则上升了19.2%.

表2　Zero-riscy支持ARMv6-M前后各功能单元所占面积及其总面积(μm^2)

	IF	BT	ID	EX	LSU	CSR	Top	总面积
不支持ARMv6-M	3527.3	—	8930.5	9629.8	1033.4	9652.1	29.6	32802.7
支持ARMv6-M(仅二进制翻译)	3940.3	4356.4	8863.8	9736.6	1037.9	9678.5	29.6	37643.1
支持ARMv6-M(使用硬件优化)	3978.9	3462.6	8965.5	10003.7	1040.6	9777.8	37.0	37266.1

使用硬件实现标志位和分支指令后,Zero-riscy运行RISC-V代码的功耗上升了8.2%;翻译运行ARMv6-M程序的总功耗则比直接运行RISC-V程序高出14.3%,比原始的Zero-riscy高出23.8%. 功耗开销大于仅使用二进制翻译的原因是ARM Thumb标志位的判断和比较逻辑在运行程序会产生额外的动态功耗.

最后,在使用硬件优化支持ARMv6-M的基础上,通过两种方式实现条件执行分别带来了593.04 μm^2和713.32 μm^2的面积开销. 在二进制翻译器中实现条件执行的判断逻辑增加的面积大于在译码段判断执行条件主要是因为Zero-riscy是一个两段流水线,一条指令的译码和执行在同一个周期完成,在译码段判断执行条件时标志位的判断逻辑可以和分支指令的判断逻辑

表3　Zero-riscy支持ARMv6-M前后的功耗(mW)

	静态功耗	动态功耗	总功耗
原Zero-riscy运行RISC-V程序	2.27	2.33	4.60
仅二进制翻译运行RISC-V程序	2.58	2.36	4.94
仅二进制翻译运行ARM Thumb程序	2.58	2.90	5.48
使用硬件优化运行RISC-V程序	2.58	2.39	4.97
使用硬件优化运行ARM Thumb程序	2.59	3.10	5.69

共享;而在二进制翻译器中实现则必须额外增加一个标志位判断逻辑. 最后,在功耗上,执行指令序列3中的指令时,两种方法并没有太大的差距:在翻译时判断执行条件的功耗略高于在译码时判断执行条件0. 12 mW.

4 结论

本文对通过硬件二进制翻译的方式在RISC-V处理器上支持ARM Thumb指令集进行了研究. 主要讨论了从ARM Thumb到RISC-V指令集的指令映射、寄存器映射、标志位、分支指令以及条件执行的处理方式. 从ARM Thumb指令到RISC-V指令的翻译比是影响RISC-V处理器翻译执行ARM Thumb的关键因素,我们提出了使用硬件实现ARM Thumb中标志位及分支指令的方式来大幅降低翻译得到的RISC-V指令数量. 同时,提出了通过在二进制翻译器中加入执行条件判断逻辑的方式来减少因条件执行指令导致的空转周期. 基于Zero-riscy处理器,我们实现了一个支持ARMv6-M架构的多指令集处理器,通过运行基准程序评估了其运行RISC-V和ARMv6-M程序的性能,并对其进行了综合. 实验结果显示,在13. 6%的面积开销和23. 8%的功耗开销下,Zero-riscy翻译运行由ARM Compiler编译的Embench测试套件的ARM Thumb代码的平均性能能够达到直接运行GCC编译的RISC-V代码的75. 5%,部分基准程序的性能甚至超过了直接运行RISC-V代码的性能. 相对于仅使用二进制翻译实现多指令集处理器,其性能提升了3. 1倍,而面积开销则下降了7. 8%.

参考文献

[1] Sites R L, Chernoff A, Kirk M B, et al. Binary translation [J]. Communications of the ACM, 1993, 36(2):69 – 81.

[2] Apple Inc. About the Rosetta Translation Environment[EB/OL]. https://developer. apple. com/documentation/apple_silicon/about_the_rosetta_translation_environment, 2020.

[3] Chernoff A, Herdeg M, Hookway R, et al. FX!32: A profile-directed binary translator[J]. IEEE Micro, 1998, 18(2): 56 – 64.

[4] Bellard F. QEMU, a fast and portable dynamic translator [A]. Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference[C]. Anaheim, CA, USA: USENIX, 2005. 46.

[5] Hong D Y, Hsu C C, Yew P C, et al. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores[A]. Proceedings of the Tenth International Symposium on Code Generation and Optimization[C]. San Jose, New York: ACM, 2012. 104 – 113.

[6] Altman E R, Kaeli D. Welcome to the opportunities of binary translation[J]. Computer, 2000, 33(3):40 – 45.

[7] Shen B Y, Chen J Y, Hsu W C, et al. LLBT: an LLVM-based static binary translator[A]. Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems[C]. New York: ACM, 2012. 51 – 60.

[8] Krishnan Sivaram, Debbage Mark, Ziesler Sebastian H, Roy Kanad, Sturges Andrew C, Biswas Prasenjit. Processor Architecture for Executing Two Different Fixed-length Instruction Sets[P]. US:US2005262329, 2005-11-24.

[9] Fajardo J, Rutzig M B, Carro L, et al. Towards a multiple-ISA embedded system[J]. Journal of Systems Architecture the Euromicro Journal, 2013, 59(2):103 – 119.

[10] Goetz John W, Mahin Stephen W, Bergkvist John J. Processor Capable of Supporting Two Distinct Instruction Set Architectures[P]. EP:EP0747808, 1996-12-11.

[11] James Walter Rymarczyk, Michael Ignatowski, Thomas J Heller Jr. Multiple-core Processor Supporting Multiple Instruction Set Architectures[P]. US: US8806182, 2014-08-12.

[12] Venkat A, Tullsen D M. Harnessing ISA diversity: design of a heterogeneous-ISA chip multiprocessor[A]. 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA) [C]. AvenueMinneapolis, MN, USA: IEEE, 2014. 121 – 132.

[13] Balkind J, Lim K, Schaffner M, et al. BYOC: a" bring your own core" framework for heterogeneous-ISA research [A]. Proceedings of the Twenty-fifth International Conference on Architectural Support for Programming Languages and Operating Systems[C]. New York: ACM, 2020. 699 – 714.

[14] Capella F M, Brandalero M, Carro L, et al. A multiple-ISA reconfigurable architecture[J]. Design Automation for Embedded Systems, 2015, 19(4): 329 – 344.

[15] Schiavone P D, Conti F, Rossi D, et al. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications[A]. 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS) [C]. Thessaloniki, Greece: IEEE, 2017. 1 – 8.

[16] Smith J, Nair R. Virtual Machines: Versatile Platforms for

Systems and Processes[M]. Amsterdam: Elsevier, 2005.
[17] Andrew Waterman, Krste Asanovi. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA[EB/OL].https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf, 2019.

作者简介

成元虎　男,1996年出生,四川中江人.现为国防科技大学计算机学院硕士研究生.主要研究方向为微处理器体系结构.
E-mail:chengyuanhu@nudt.edu.cn

黄立波(通信作者)　男,1983年出生,湖南邵阳人.现为国防科技大学计算机学院副研究员,主要研究方向为计算机体系结构.
E-mail:libohuang@nudt.edu.cn

崔益俊　男,1995年出生,江苏东台人.现为国防科技大学计算机学院硕士研究生.主要研究方向为微处理器体系结构.
E-mail:cuiyijun18@nudt.edu.cn

马　胜　男,1986年出生,湖南永州人.现为国防科技大学计算机学院副研究员.主要研究方向为计算机体系结构.
E-mail:masheng@nudt.edu.cn

王永文　男,1977年出生,山东泰安人.现为国防科技大学计算机学院研究员.主要研究方向为微处理器体系结构.
E-mail:yongwen@nudt.edu.cn

隋兵才　男,1981年出生,山东烟台人.现为国防科技大学计算机学院副研究员.主要研究方向为微处理器体系结构.
E-mail:bingcaisui@nudt.edu.cn