计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2022,31(9):24−30 [doi: 10.15888/j.cnki.csa.008841]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于链接器的 RISC-V 字加载指令优化①

乌鑫龙[2], 廖春玉[1]

[1](中国科学院 软件研究所, 北京 100190)
[2](北京师范大学珠海分校 计算机学院, 珠海 519087)
通信作者: 廖春玉, E-mail: chunyu@iscas.ac.cn

摘　要: RISC-V 作为精简指令集的代表, 也会反映一些精简指令集的弊端, 程序体积偏大就是其中之一. 在精简指令集 (RISC) 中, 实现一些复杂操作所需要的指令条数普遍会多于复杂指令集 (CISC), 进而导致最后生成的二进制程序体积相较 CISC 程序更大. 并且嵌入式设备的 RAM 和 ROM 普遍较小, 因此在嵌入式场景中, 程序的体积变得尤为重要. 为了在现有压缩指令集的基础上尽可能的优化 RISC-V 程序代码体积, RISC-V 指令集子扩展 Zce 制定了一系列指令. 其中以 LWGP 为代表的一系列指令被用来减少加载/存储字节数据时的指令条数. 本文分析了以 LWGP 为代表的指令对于代码体积的优化原理并且将之实现在 LLD 链接器上, 通过分析使用 LWGP 等指令前后程序体积的变化评估对于二进制程序体积优化的效率并且提出后续改进建议.

关键词: RISC-V; 代码体积优化; LLD; 链接器优化

引用格式: 乌鑫龙,廖春玉.基于链接器的 RISC-V 字加载指令优化.计算机系统应用,2022,31(9):24–30. http://www.c-s-a.org.cn/1003-3254/8841.html

RISC-V Load Instruction Optimization Based on LLD

WU Xin-Long[2], LIAO Chun-Yu[1]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](School of Computer Science, Beijing Normal University, Zhuhai, Zhuhai 519087, China)

Abstract: As a typical example of reduced instruction sets, RISC-V can also reflect some disadvantages of the reduced instruction set computer (RISC), and large program size is one of the problems. Compared with the complex instruction set computer (CISC), RISC generally requires more instructions to implement complex operations and results in a large binary size of the program. Meanwhile, RAM and ROM in embedded devices are generally small. Therefore, it means that the binary size of the program is significant for embedded scenarios. In view of this, the Zce sub-extension of RISC-V has developed a series of instructions to reduce the program size as much as possible. Specifically, the instructions represented by the LWGP are used to reduce the number of instructions when loading/storing bytes. This study analyzes the principle of the LWGP instructions in reducing the code size and implements it on the LLD linker. It also evaluates the efficiency of LWGP in reducing the binary size of the program by analyzing the change in program size before and after using LWGP instructions and puts forward recommendations for improvement.

Key words: RISC-V; code size reduction; LLD; linker optimization

1　引言

由于 RISC-V 指令集架构具有开源、芯片设计友好、开发成本低等特点[1,2], 近年来被越来越多地运用于嵌入式设备中. 同时 RISC-V 指令集作为 RISC 的一

① 本文由“RISC-V 技术及生态”专题特约编辑武延军研究员、宋威副研究员、张科高级工程师以及邢明杰高级工程师推荐.
此工作由第一作者在中国科学院软件研究所 PLCT 实验室实习期间完成.
收稿时间: 2022-03-26; 修改时间: 2022-04-25, 2022-05-16; 采用时间: 2022-06-01; csa 在线出版时间: 2022-07-22

员, 也会不可避免的存在一些精简指令集的弊端, RISC 二进制程序体积偏大的问题就是其中之一. 因为 RISC 指令集只要求实现计算机硬件中最常用且数量有限的基础指令, 所以其中较复杂的操作只能通过基础指令的组合来实现. 因此在完成相同操作的情况下, 相较于直接包含复杂操作指令的 CISC 来说, RISC 程序往往需要执行更多条指令. 尤其是在内存大小受限的嵌入式设备中[3], 二进制程序体积偏大的问题更加突出[1].

本文第 2 节简要介绍了相关减小程序体积的部分方法以及研究, 并且简要介绍 Zce 子扩展对于指令优化的效果. 第 3 节介绍 RISC-V 架构以 Zce 子扩展的优化思路. 第 4 节详细解释了 LSGP 指令优化程序体积的方法. 最后基于 LLD 链接器实现了 LSGP 指令的优化并对优化效率进行分析.

2 相关研究

针对 RISC 程序体积偏大的问题, 目前主流方法之一就是在基础指令集以外额外支持一个“短指令集”. 该指令集用更短的指令宽度编码基础指令集中最常用的指令从而二进制程序体积. 在 ARM 架构中就使用 Thumb 指令集缩减程序体积, MIPS 架构则有 MISP16 指令集承担缩小程序体积的任务[3]. 得益于 RISC-V 指令集可扩展性高的特点, RISC-V 当前也有 C 指令集子扩展被用于同样的目的.

除此之外, Halambi 等人[3] 还通过对 MIPS 指令建模, 使用启发式的方法来估算因为寄存器数量有限导致被分配的堆栈, 计算分析从而更细粒度地选择压缩指令. 在 MISP 16 压缩指令集的基础上更进一步的压缩了 MIPS 二进制程序的体积.

在嵌入式领域的基准测试中, RISC-V 架构的二进制体积相较 ARM 架构增大了约 11%, 即使在使用了 C 子扩展的情况下仍有较大差距[4]. 本文研究的 RISC-V 的 Zce 子扩展[4] 与 C 子扩展同样被用来解决二进制程序体积偏大的问题. 但与 C 子扩展不相同的是, 该扩展除了通过缩减常用指令的长度以外, 还尝试替换频繁使用的固定指令组合从而缩减程序体积, 进一步增加了代码密度. 具体而言, 在 C 子扩展的基础上, Zce 子扩展使得二进制体积比 ARM 架构小约 1.75%. 本文对于 Zce 扩展中以 LWGP 为代表的指令进行研究, 基于 LLD 链接器实现该优化并且评估其优化效率.

3 RISC-V 指令集扩展

RISC-V 指令集由基础指令集和众多扩展指令集组成. 其中基础指令集包含了如整数加减和位运算以及分支跳转指令等, 如表 1 所示. 这些指令足以支撑一个简单的裸机程序或者操作系统的运行.

除此之外, 基础指令集还为 RISC-V 指令集定义了 x0–x31 共 32 个通用寄存器. 每个寄存器都有其对应的用途. 如表 2 所示.

表 1 基础指令集中常用的主要指令

指令分类	指令助记符
Load/Store	LB, LH, LW, LBU, LHU
	SB, SH, SW
Shift	SLL, SLLI, SRL, SRLI, SRA, SRAI
Arithmetic	ADD, ADDI, SUB, LUI, AUIPC
Logical	XOR, XORI, OR, ORI, AND, ANDI
Compare	SLT, SLTI, SLTU, SLTIU
Branches/Jump	BEQ, BNE, BLE, BGE, BLTU, BGEU, JAR, JARL

表 2 通用寄存器使用规范

Name	Mnemonic	Meaning
x0	zero	zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5–x7	t0–t2	Temporary registers
x8–x9	s0–s1	Callee-saved registers
x10–x17	a0–a7	Argument registers
x18–x27	s2–s11	Callee-saved registers
x28–x31	t3–t6	Temporary registers

表 2 中, gp 和 tp 寄存器则较为特殊, 它在程序执行的过程中被当作常量值使用.

3.1 Zce 指令集扩展

与基础指令集不同, Zce 扩展更多的是针对当前已有指令的压缩和优化问题. 它通过减少指令中某些情况下冗余操作数或者替换常用的固定指令组合来缩减单个指令的长度和指令条数.

以基础指令集中逻辑运算指令为例, 逻辑运算中只有与, 或和异或, 非运算则通过将源寄存器异或−1 实现. 图 1 是 XORI 的指令格式

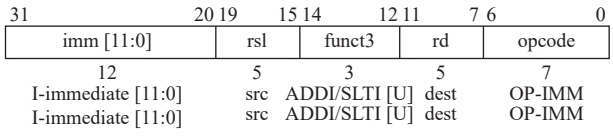

图 1 XORI 指令格式

XORI 指令将 rs1 寄存器中的数和立即数 imm 按位异或运算, 结果写入 rd 寄存器. 在非运算过程中较频繁的会出现 rs1 和 rd 使用同一个寄存器的情况, 因此 Zce 扩展尝试将这种情况下 rs1 和 rd 寄存器合并以节约编码点. Zce 中将非运算 (c.not) 定义为如图 2 格式.

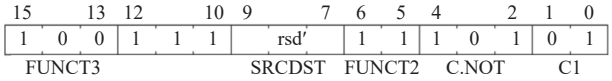

图 2 c.not 指令格式

c.not 指令合并 rs1 和 rd 寄存器为 rsd, 同时因为约定立即数为−1 所以删除了立即数. 将基础指令集实现的 32 位非运算指令 XORI rd, rs1, −1 缩减为 16 位的 c.not 指令.

Zce 扩展中还有一部分指令用于压缩固定的指令组合. 例如 push/pop 指令. 在 RISC-V 汇编中, 函数的开始和末尾都需要保存和恢复堆栈指针和参数或返回值. Zce 以使用 push/pop 指令一次性代替多条 SW/LW 指令的方式缩减指令条数.

LSGP 指令缩减程序体积的方式也与之类似, LSGP 指令仍为 32 bit 指令, 它通过提高硬件的复杂度, 将两条指令合并为一条指令从而减小程序二进制体积. Zce 中使用 GP 寄存器进行优化的指令共 4 条, 分别是 LWGP、SWGP、LDGP、SDGP (下用 LSGP 指代全部 4 条指令). 其与基础指令集的 Load/Store 指令对应. 其中, LDGP 和 SDGP 仅被用于 RISC-V 64 位机器中加载双字长的数据.

4 LWGP 指令优化原理

4.1 LW 指令介绍

前面提到的 RISC-V 基础指令中还定义了字加载/存储指令, 分别是 LW、SW、LD、SD (下多以 LW 指令为例), 它们被用来从给定地址加载字节数据. 其指令格式如图 3 所示. LW 指令使用 rs1 的值为基地址, 将 rs1+offset 处 4 个字节的数据加载到 rd 寄存器中. 这就意味着在程序执行 LW 指令之前仍需使用额外指令将基地址加载到 rs1 寄存器中. 适用于这种情况的有两条指令, 分别是 ADDI 和 LUI. 本文主要研究使用 LUI 指令加载基地址的情况.

代码示例 1. LW 指令的使用

```
lui a0, 512
lw a1, 256(a0)
```

在代码示例 1 中, 两条指令一起被用来加载位于 0x200100 的数据. 由 LUI 现将该数据的高二十位地址加载进 a0 寄存器, 再由 LW 指令将位于此处的数据加载到 a1 中.

为了对这种情况进行优化, RISC-V 引入了一个全局指针寄存器 GP. 这个寄存器的值在链接过程中被确定并且在程序执行过程中保持不变. GP 寄存器主要被用来优化程序中全局变量的访问, 所以在一般情况下, 链接器会将 GP 指针指向 ELF 文件中.sdata 小数据段+0X800 的位置. 当某一个全局变量可以被以 GP 为基地址访问时. 链接器就会删除 LW 指令之前的 LUI 指令以缩减代码体积. 示意如图 4.

但是由于 LW 指令中的偏移量的长度仅有 12 bit, 因此仅能访问到 GP±2 KB 范围内的全局变量. 对于超出该范围的全局变量, 就仍需要 LUI 指令通过其他寄存器传递基地址.

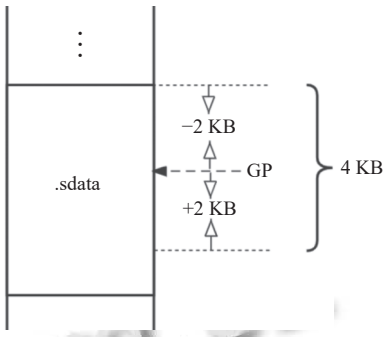

图 3 Load/Store 指令格式

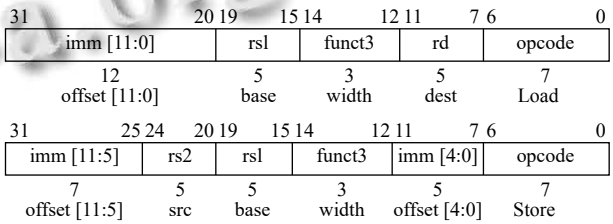

图 4 GP 指针位置示意

4.2 通过 LWGP 提高 LW 指令的访存能力

LW 指令的主要问题是偏移量位宽不足以当前情况. 它只能访问基地址±2 KB 范围内的变量. 所以需要较为频繁的使用 LUI 指令以重新加载新的基地址. 而 LWGP 正是通过增加长偏移量的位宽提高了其访存能力. LWGP 指令格式如图 5.

LWGP 指令事先约定了使用 GP 寄存器作为基址寄存器, 如此可以将 LW 中基地址寄存器 rs1 对应的编

码点分享给偏移量 offset 使用. 这样就可以使得偏移量的宽度从 LW 的 12 bit 扩展到了 LWGP 的 16 bit, 从而使 LWGP 指令可以访问 GP±32 KB 范围内的全局变量. 基于同样的原理, LDGP 和 SDGP 的访存能力更是扩大到了 GP±64 KB 的范围.

31:29	28:25	24:20	19:15	14:12	11:7	6:0	instruction
000	imm [8:2, 10:9]		imm [15:11]	011	rd	0000111	LWGP

图 5 LWGP 指令格式

5 基于 LLD 的 LSGP 指令优化

正如上文所提到的, GP 指针被用来优化全局变量的访问. 然而在程序链接之前, 全局变量的地址还尚未被确定. 因此当前生成的一些汇编指令需要使用标志符预先占位, 如 %hi (symbol) 代表符号 symbol 的高 20 位地址, 这些标志并不能被直接编码到二进制指令中, 所以编译器会使用重定位类型 (如 R_RISCV_LO12_I) 标记这条指令, 表示这条指令还需要链接器做后续处理. 而具体的相应数据会在链接过程中被写入.

5.1 链接器松弛

在链接器松弛过程中, 链接器会从整个可执行程序的视角对于代码进行优化. 链接器会读取并解析文件中所有的重定位信息, 针对每一条重定位信息进行相应的优化处理, 链接器松弛的简要流程如图 6 所示. 每条重定位信息的优化方式取决于该条信息的重定位类型, 不同的重定位类型对应着不同的函数方法. 本文的优化主要设涉及 3 种重定位类型, 分别是 R_RISCV_HI20、R_RISCV_LO12_I 和 R_RISCV_LO12_S.

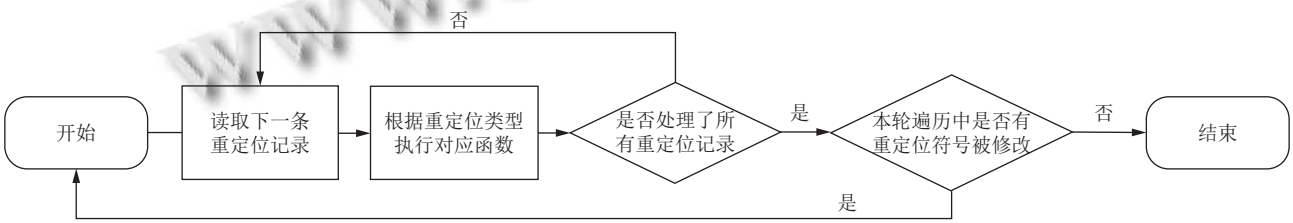

图 6 链接器松弛简要流程

当 LUI 指令被用来加载一个全局变量的高 20 位地址时, 编译器会将该指令用 R_RISCV_HI20 标记. 同时, 该指令通常会和使用全局变量低 12 位的 LW 指令一起使用. 以本文研究涉及到的 LW 和 LUI 的指令为例, 编译器会给 LW 指令标记重定位类型 R_RISCV_LO12_I.

在链接器的松弛阶段中, 链接器会不断重复扫描并尝试优化程序中每一条重定位信息, 直到全部的重定位信息都不能够再次被优化. 其中被用来加载全局变量的 LUI 和 LW 两条指令会被尝试优化成以 GP 寄存器为基地址寄存器的 LW 指令. 基于同样的逻辑, 本文主要讨论的 LSGP 也需要做相似的处理.

代码示例 2. LW 指令使用的重定位类型

```
lui a0, %hi (symbol) # R_RISCV_HI20 (symbol)
lw a0, %lo (symbol) (a0) # R_RISCV_LO12_I (symbol)
```

由于 LSGP 指令格式与其他指令都不相同, 因此并不能被目前已经存在的重定位标记正确处理. 于是在这一阶段我们定义了新的重定位类型来指定 LSGP 指令的优化操作. 同时, 链接器中所有被用到的重定位类型都需要由 psABI 来定义. 但是由于 Zce 扩展仍处在实验阶段, psABI 中没有定义相关的重定位类型, 因此出于实验测试目的, 作者针对 LSGP 临时定义了重定位标记用于指令的优化.

链接器松弛结束后, 每一条被重定位类型标记的指令会被按照这个重定位类型的要求计算地址, 并且填充到对应占位标志的地方.

5.2 链接器上 LSGP 优化的实现

本节中使用 LLD 链接器为例子进行讨论. 由于 LLD 主线针对于 RISC-V 链接器松弛的实现尚不完善, 因此我们使用了一个上游正在 review 的补丁来完善相关功能. 在此基础上进行 LSGP 等指令的生成、优化以及评估工作. 同时, 为了能够单独评估 LSGP 的优化效率, 我们定义了一个-mzce-lsgp 开关, 用来更直观地评估 LSGP 四条指令的优化效率.

编译器会为 LUI 和 LW 指令分别标记重定位类型 R_RISCV_HI20 和 R_RISCV_LO12_I. 在初始阶段,

链接器就会统一提取所有的重定位信息. 因此我们通过遍历重定位标记就可以找到需要被优化的指令. 但值得注意的是, LUI 指令并不仅会和 LW 被一起使用. 也会和例如 ADDI 等其他指令一同被用来加载绝对地址. 因此我们在判断 LUI 指令是否可以被优化的时候还需要提前读取并判断下一条指令是否属于字加载指令.

在此之后还需针对 R_RISCV_LO12_*进行优化. 为了避免造成额外的影响, 首先需要判断当前指令是否为 LW/SW/LD/SD 的其中之一, 之后计算当前指令使用的全局变量是否位于 LSGP 指令要求的地址范围内. 过程中要注意保存 rd 寄存器的值来确保优化前后的功能不会改变. 整体实现逻辑伪代码如示例代码 3 所示. 链接器在将 LW 修改成 LWGP 的过程中并不会对偏移量 offset 参数进行赋值, 它的值将会在链接器优化阶段结束后被统一调整.

代码示例 3. 优化 LUI 和 LSGP 指令

```
1. for each rel in relocations
2.   if rel.type is R_RISCV_HI20 and rel.inst is LUI
3.     if rel.nextInst is one of (LW ro LD or SW or SD)
4.       if rel.inst.offset is in range of Uint16 and aligned by 4 b
5.         Call removeInst(rel)
6.   else if  rel.type is R_RISCV_LO12_I or R_RISCV_LO12_S
7.     if rel.inst is LW
8.       if rel.inst.offset is in range of Uint16 and aligned by 4 b
9.         Call repleaseInstByLWGP(rel)
10.          rel.type = R_RISCV_LWGP
11.    else if rel.inst is SW
12.      if rel.inst.offset is in range of Uint16 and aligned by 4 b
13.        Call repleaseInstBySWGP(rel)
14.        rel.type = R_RISCV_SWGP
15.    else if rel.inst is LD
16.      if rel.inst.offset is in range of Uint17 and aligned by 8 b
17.        Call repleaseInstByLDGP(rel)
18.        rel.type = R_RISCV_LDGP
19.    else if rel.inst is SD
20.      if rel.inst.offset is in range of Uint17 and aligned by 8 b
21.        Call repleaseInstBySDGP(rel)
22.        rel.type = R_RISCV_SDGP
23. end
```

链接器优化结束后, 意味着各个段的地址已经被最终确定. 编译器会分别为不同的重定位标记计算地址, 并按照相应的指令格式将偏移量写入指令. 同样因为 LSGP 四条指令的格式各不相同, 所以需要分别处理.

6 LSGP 优化效率分析

为了分析 LSGP 指令对于程序的优化效果, 我们尝试使用上文中修改的 LLD 和 Clang 对 RISC-V 测试 (riscv-test) 代码的部分程序进行编译链接. 并对比分析使用 LSGP 指令前后反汇编代码数目.

6.1 LSGP 缩减代码体积的比例

由于 LSGP 指令针对于全局变量进行访问, 我们从 RISC-V test 测试集合中选取了两个使用全局变量较为频繁的测试程序, 分别是用来测试整数加法的 Dhrystone 测试以及测试递归调用的 towers 测试, 此外还编译了 Linux 常用软件 bash 和 vim 进行测试. 测试过程中均以 riscv32imac 作为基准, 结果如图 7 所示.

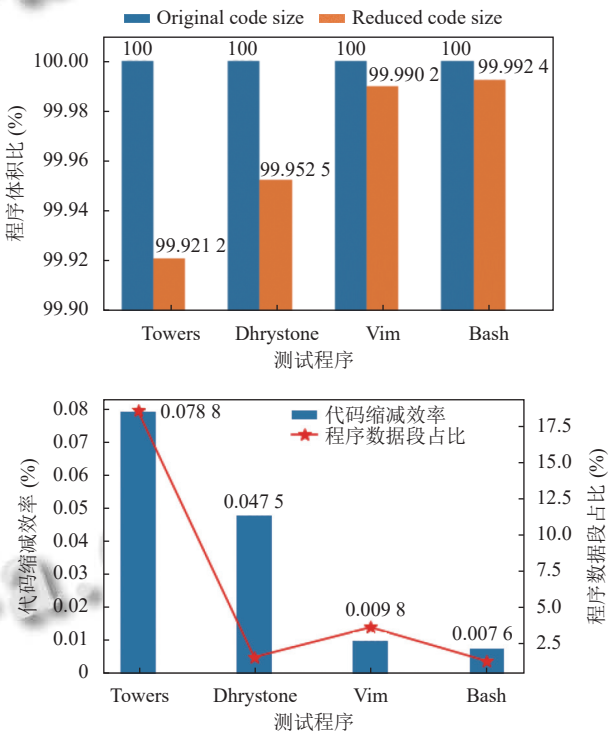

图 7 LSGP 缩减程序体积与数据段大小的关系

在 Dhrystone 测试程序中, 使用 LSGP 指令前后反汇编得出的指令条数分别为 18 447 和 18 413 条. .data 和.sdata 共计 4 384 字节. 有 22 个全局变量被访问, 共计 184 字节的全局数据通过 LDGP/SDGP 被访问. 优化前二进制大小 286 216 字节, 优化后二进制大小 286 080 字节. 二进制体积减少约 0.047%. 对于 towers 测试程序, 由于相对使用全局变量较少且程序整体代码量较少. .data 与.sdata 总计 1 880 字节, 共计 36 字节全局数据通过 LSGP 被访问, 优化前二进制大小为 10 152 字

节, 优化后为 10 144 字节. 二进制程序体积减少了 0.07%. 我们还尝试编译了目前常用的 GNU 软件 vim 和 bash 作为日常软件的代表. 与测试集合中刻意的测试代码不同, bash 和 vim 程序体积减小的幅度小于 Dhrystone 和 towers. Bash 中.data 与.sdata 总计 35 548 字节中的 148 字节数据被通过 LSGP 指令访问. Vim中则共有 782 字节的数据被 LSGP 访问. 相较于使用 LSGP 加载数据之前, bash 和 vim 二进制程序体积的缩减效率分别是 0.007 6% 和 0.009 8%. 总体而言, 程序体积的缩减效率与程序数据段占比呈正相关.

表 3 中展示的优化效率看似较为低下, 其主要由于 Zce 扩展的优化空间所导致. Zce 指令集的目的是在 C 压缩指令扩展的基础上进一步缩减程序体积. C 扩展指令已经将 RISC-V 程序体积大幅度缩小. 尽管如此, 相较于 ARM Cortex M4 架构下的二进制程序, 仍然有不到 10% 的体积差距[5]. 于是 Zce 子扩展则致力于进一步缩小这不到 10% 的差距. 这也就导致了 Zce 扩展的优化空间普遍较小, 从而优化效率相较于 C 指令集较低. 同时考虑到 Zce 扩展中其他单条指令的优化效率也都在 0.02%–0.24% 之间, 所以从这个角度来分析 LSGP 作为单条指令的优化效率也算合格.

表 3 LSGP 缩减程序体积的效率

项目	Dhrystone	Towers	Bash	Vim
数据段体积 (.data+.sdata)	4 384	1 880	35 548	126 712
原程序体积	286 216	10 152	2 818 904	3 528 864
数据段占比 (%)	1.53	18.52	1.26	3.59
优化后程序体积	286 080	10 144	2 818 690	3 528 518
体积缩减效率 (%)	0.047 5	0.078 8	0.007 6	0.009 8
通过LSGP访问的数据占比 (%)	4.19	1.91	0.42	0.62

对于代码体积的优化问题, 本研究中主要针对于对全局变量的优化, 因此正如表 3 和图 7 所体现的, 一个程序中全局变量的数量或占比决定了 LSGP 优化的效率. 而程序中全局变量使用的数量一定程度上取决于程序的规模[6] 和功能. 因此, 对于底层软件, 例如操作系统, 单片机程序等也会使用到较多全局变量的程序来说, LSGP 缩减代码体积的效果同样是乐观的.

7 结论和展望

综上所述, 与 LW 等常规字加载指令相比, LSGP 指令能够针对 LW 指令的部分使用场景进行优化, 通过约定基址寄存器的方式将寄存器的位宽分配给偏移量使用, 从而扩大指令的寻址范围. 本文在 LLD 链接器上实现这部分的优化并进行了评估. 对于 RISC-V 的部分标准测试程序来说, LSGP 达到了较高的优化效率. 同时在日常通用软件中, LSGP 对于程序体积的缩减也起到了一定的作用.

虽然前文描述了 LWGP 确实在一定程度上优化了代码体积. 但是优化效率相较于标准测试程序中的理想条件仍有一定差距. 可以通过改进以下问题进一步减小这个差距.

(1) LSGP 指针的编码不合理.

(2) 部分 LSGP 的寻址能力被浪费.

(3) 局限于优化.sdata 段而忽略了其他可以被优化的数据段.

psABI 只考虑到 LW 指令的 4K 寻址能力. 因此将 GP 指针的值设置为.sdata+2K (0X800) 的位置来确保尽可能大覆盖到.sdata 节的数据. 但是对于 LSGP 指令达到 64 KB 的寻址能力来说, GP 指令仍位于.sdata+2K (0X800) 位置的话就意味着 LSGP 指令的寻址范围并不是从.sdata 段开始. 所以最简单的办法本应是改变 GP 指针的位置, 但由于 LSGP 指令无法完全代替 LW 指令, 无法改变 GP 指针的位置. 基于此, 当前最好的解决办法就是尝试更改 LSGP 指令的格式, offset 偏移量从带符号数改为无符号数, 从 GP±32 KB 变成 GP±64 KB, 这样 LW 和 LWGP 搭配使用, 可以通过 GP 指针访问更大范围的全局变量.

又因为 LSGP 大部分的寻址范围覆盖到了除.sdata 段以外的地址. 因此每个数据段之间的相对位置就变得相对重要. 如果相关的数据段排列在一起, 可以更大程度上避免 LSGP 寻址能力被浪费. 同时, .sdata 段是小数据段, 其存储了数据长度小于某一阈值 (通常小于 8 字节) 的变量, 其余的全局变量会被存储到.data 段. 这就导致程序中的.sdata 段普遍较小, 甚至一部分程序根本不存在.sdata 段. 目前链接器的实现 (以 LLD 为例) 仅基于.sdata 段设置 GP 指针. 如果.sdata 段不存在, 则 GP 指针就会被 LLD 忽略, 不只 LSGP, 甚至对于 LW 的优化也会被无效化. 如果链接器在.sdata 段不存在的情况下将 GP 指向.data 段, 程序体积可以被进一步缩减.

8 结束语

本文通过介绍和分析 LW 指令的作用以及存在的

问题, 阐述了 LSGP 指令的优势和特点. 将之实现到 LLD 链接器上并粗略评估了 LSGP 指令优化效率. 相较于现有的字加载指令, LSGP 通过扩大偏移量立即数的位宽增大寻址范围的方式避免使用 LUI 指令加载高位地址, 从而缩减代码条数和程序体积的方式, 针对于使用 GP 寄存器作为基址的情况进行优化. 证明了 LSGP 指令存在一定的优化价值. 同时在整个过程中作者也发现了目前 LSGP 作为实验性指令存在的一些问题. 针对于这些问题提出了相应的解决方案. 我们已经将这些问题和建议反馈到 RISC-V 社区.

参考文献

1 Bakthavatsalam G, Mehata KM. A case for hybrid instruction encoding for reducing code size in embedded system-on-chips based on RISC processor cores. Journal of Computer Science, 2014, 10(3): 411–422. [doi: 10.3844/jcssp.2014.411.422]

2 王诲喆, 唐丹, 余子濠, 等. 开源芯片、RISC-V 与敏捷开发. 大数据, 2019, 7(4): 50–66. [doi: 10.11959/j.issn.2096-0271.2019032]

3 Halambi A, Shrivastava A, Biswas P, *et al*. An efficient compiler technique for code size reduction using reduced bit-width ISAs. Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition. Paris: IEEE, 2002. 402–408. [doi: 10.1109/DATE.2002.998305]

4 Perotti M, Schiavone PD, Tagliavini G, *et al*. HW/SW approaches for RISC-V code size reduction. Workshop on Computer Architecture Research with RISC-V. Zurich: ETH Library, 2020. 1–8.

5 Ackerman J. Initial evaluation of multiple RISC ISAs using the Embench™ benchmark suite. 2019 RISC-V Summit. California: RISC-V Foundation, 2019.

6 Gesellensetter L, Glesner S. Interprocedural speculative optimization of memory accesses to global variables. 14th International Euro-Par Conference on Euro-Par 2008 Parallel Processing. Las Palmas de Gran Canaria: Springer, 2008. 350–359.

(校对责编: 孙君艳)