计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2021,30(11):27−32 [doi: 10.15888/j.cnki.csa.008349]

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

基于 LLVM 的 RISC-V 向量扩展栈帧布局优化①

陆旭凡[1,2], 胡海根[2], 邢明杰[1]

[1](中国科学院 软件研究所, 北京 100190)
[2](浙江工业大学 计算机科学与技术学院, 杭州 310012)
通讯作者: 邢明杰, E-mail: mingjie@iscas.ac.cn

摘　要: 为了能够生成正确、优化的机器指令代码, 需要在编译器后端代码的生成阶段, 设计和使用合适的程序栈帧布局. 由于 RISC-V 向量扩展架构具有可伸缩性、其向量寄存器的长度在编译时不可知, 传统的栈帧布局无法适用. 之前 LLVM 中针对向量扩展实现的栈帧布局虽然能够生成正确的机器指令, 但存在访存指令较多, 栈帧空间较大, 以及预留寄存器较多等问题. 我们对原有实现所存在的问题进行分析, 在此基础上提出了新的布局方式以及向量对象地址计算方式, 并通过巴塞罗那超算中心开发的测试集进行验证. 实验表明新的栈帧布局能够有效减少访存指令数和栈空间大小.

关键词: LLVM; RISC-V; 向量扩展; 栈帧布局

引用格式: 陆旭凡,胡海根,邢明杰.基于 LLVM 的 RISC-V 向量扩展栈帧布局优化.计算机系统应用,2021,30(11):27–32. http://www.c-s-a.org.cn/1003-3254/8349.html

Optimization of RISC-V Vector Extension Stack Frame Layout Based on LLVM

LU Xu-Fan[1,2], HU Hai-Gen[2], XING Ming-Jie[1]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](College of Computer Science and Technology, Zhejiang University of Technology, Hangzhou 310012, China)

Abstract: For correct and optimized machine instructions, it is necessary to design and use a suitable program stack frame layout during the code generation stage of the compiler back-end. Due to the scalability of the RISC-V vector extension architecture and the unknown length of its vector register at compile time, the traditional stack frame layout cannot be applied. Although the previous stack frame layout implemented for vector extension in LLVM can generate correct machine instructions, it has problems such as many load/store instructions and reserved registers as well as large stack frame sizes. We analyze the problems existing in the previous implementation and propose a new layout and vector object calculation method on this basis. Then we verify it through the test set developed by the Barcelona Supercomputing Center. Experiments show that the new stack frame layout can greatly reduce the number of load/store instructions and stack space.

Key words: LLVM; RISC-V; vector extension; stack frame layout

1　引言

LLVM[1,2] 是一种开源的编译基础设施, 采用模块化的方式进行设计, 具有清晰的架构层次和详细的文档资料, 并且提供了丰富的工具集, 因此基于 LLVM 来开发支持特定体系结构的编译器, 可以缩短开发时间, 并生成高效的代码.

① 基金项目: 中国科学院战略性先导科技专项 (C 类)(XDC05040200)
Foundation item: CAS Strategic Priority Program (Category C)(XDC05040200)
本文由“RISC-V 技术与生态”专题特约编辑武延军研究员、李玲研究员以及邢明杰高级工程师推荐.
收稿时间: 2021-04-28; 修改时间: 2021-05-21, 2021-06-08; 采用时间: 2021-06-11; csa 在线出版时间: 2021-10-22

RISC-V[3] 是一种新兴的开源 RISC 指令集架构. 它的标准由非营利性的 RISC-V 基金会维护. 由于具有开放、标准的扩展方式, 使得几乎所有的计算机系统, 从小型的微控制器到超级计算机都可以使用它. 例如, 为了满足高性能计算、机器学习、密码学领域的计算需求而提出的向量扩展. RISC-V 向量扩展[4,5] 具有可伸缩性, 向量寄存器的长度由具体的硬件实现来定义, 可以在运行时动态设置向量寄存器组, 以及要处理的向量长度. 这些特性对编译器的实现提出了许多挑战. 其中之一就是程序的栈帧布局.

本文介绍了 LLVM 原有的栈帧布局方式, 并对该栈帧布局的缺点进行了分析, 随后提出了新的优化方案, 并基于巴塞罗那超算中心开发的测试集进行验证. 通过对比分析优化前后生成的机器指令, 可以看到优化后的栈帧布局能够有效减少访存指令数量以及栈空间大小. 除此之外, 新方案还可以减少预留寄存器的数量, 有助于缓解寄存器压力.

2 RISC-V 向量扩展对栈帧布局的挑战

2.1 RISC-V 向量扩展寄存器

RISC-V 向量扩展架构具有 32 个向量数据寄存器, 分别为 V0–V31. 向量寄存器的位长 *VLEN* 由具体的硬件实现来定义. 多个连续的向量寄存器可以组合一起使用, 从而能够处理更长的数据. 控制状态寄存器 vlenb 用来保存向量寄存器的字节长度 (等于 *VLEN*/8), 控制状态寄存器 vtype 用来保存向量寄存器的元素宽度 *SEW*, 向量寄存器组乘数 *LMUL* 等信息. *LMUL* 的值可以为 1、2、4、8. 图 1 展示了不同的 *LMUL* 值所对应的向量寄存器组织结构.

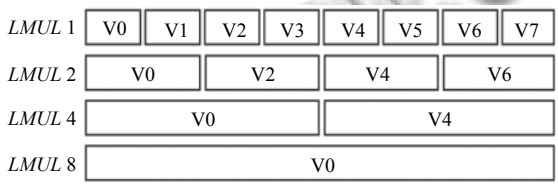

图 1 不同 *LMUL* 下的向量寄存器组织结构

用户可以通过指令 csrr 来读取控制状态寄存器 vlenb 中的内容, 从而获得向量寄存器的长度信息; 可以通过指令 vsetvli 或者 vsetvl 在运行时动态设置 *LMUL* 的值, 从而使用不同大小的向量寄存器组.

2.2 长度未知的向量类型

由此可见, 对于应用程序中实际处理的向量对象, 以及编译器中的虚拟向量寄存器, 其长度有两个因素来决定: 一个是具体处理器实现时所定义的单个向量寄存器长度 *VLEN*, 该值必须大于或等于 128, 并且为 2 的幂次方; 另一个是向量寄存器组中的寄存器个数, 即 *LMUL* 值. 为了简化编译器实现, 方便用户编程, 向量扩展 intrinsic 编程接口针对不同的元素类型和 *LMUL* 值分别提供了相应的向量数据类型. 例如元素类型为 int8, *LMUL* 分别为 1、2、4、8 的向量类型: vint8m1_t、vint8m2_t、vint8m4_t、vint8m8_t. 这些类型的 *LMUL* 值在编译时是已知的. 但是由于 *VLEN* 的值在编译时期无法静态获得, 因此向量对象的实际大小在编译时期也是未知的. LLVM 编译器在内部实现中定义了一种可伸缩向量类型, 用来表示这种长度未知的向量.

2.3 可伸缩向量对栈帧布局造成的困难

对于不使用向量类型变量的函数, 编译器一般可以通过计算每一个栈对象的大小来确定它相对于栈帧寻址指针的偏移量, 以及函数使用的栈空间大小, 从而完成栈帧布局并且在编译时期确定栈对象的地址. 而对于使用向量类型变量的函数, 由于向量类型的大小在编译时期未知, 因此无法在编译时期确定栈对象与栈帧寻址指针的偏移量. 这就导致静态分配栈对象, 编译时期确定栈对象地址的方式无法适用.

3 LLVM 原有向量扩展栈帧布局

图 2 展示了 LLVM 原有的 RISC-V 向量扩展栈帧布局. 图中有 3 种类型的指针[6], 分别用于指向栈帧中的不同位置, 帧指针指向栈帧的顶部, 栈指针指向栈帧的底部, 而基指针只在函数有可变大小的变量 (例如变长数组) 并且需要栈地址对齐的情况下存在, 指向栈帧中最后一个固定大小的对象.

在这种布局方式下, 对于每一个向量类型对象, 在栈帧中会首先分配一个对象, 用于存储栈中向量类型对象的地址. 由于是用来存储地址, 因此它的大小是固定的. 接着通过控制状态寄存器读写指令获取 vlenb 的值, 在栈上动态分配 *VLEN*×*LMUL* 大小的空间. 最后将向量对象的地址存储到之前分配的固定大小对象中.

如果要读取或者写入一个栈向量对象, 编译器首先要获得向量对象的地址. 在这种栈帧布局下, 由于在分配过程中已经对每一个向量对象的地址进行保存, 因此要获得向量对象的地址, 可以直接通过 load 指令

将固定大小对象中的内容读取到通用寄存器中. 然后, 就可以通过寄存器间接寻址方式对栈向量对象进行访问.

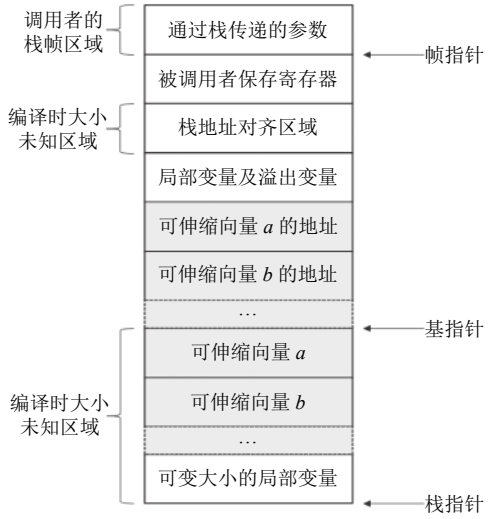

图 2 LLVM 原有的向量扩展栈帧布局

3.1 存在的问题

3.1.1 需要较多的访存指令

原有栈帧布局下, 在访问栈向量对象时需要先读取该向量对象的地址, 因此需要大量的 load/store 指令来完成对栈向量对象的分配与读写. 在现代计算机存储体系结构下, 内存读写操作通常要比算术运算操作花费更多的时钟周期[7], 且能耗更高[8], 因此较多的访存指令将会导致程序性能下降.

3.1.2 需要较大的栈空间

原有栈帧布局下, 在分配每一个栈向量对象时, 需要额外分配一个保存其地址的空间, 在 RV64 架构下, 保存地址的空间大小是 8 个字节. 因此每分配一个栈向量对象, 都需要多分配 8 个字节, 且可能由于对齐的要求, 每个栈向量对象会分摊到更多的字节.

3.1.3 需要较多的预留寄存器

在 LLVM 代码生成阶段, 如果是在寄存器分配过程之后创建新的虚拟寄存器, 则需要在栈帧的溢出变量区域中首先分配一个紧急溢出槽. 在之后的寄存器清扫过程中, 如果该虚拟寄存器的活跃区间内有可用的物理寄存器, 则使用该物理寄存器来替换, 如果没有可用的物理寄存器, 则需要溢出一个活跃的物理寄存器到紧急溢出槽. 在对紧急溢出槽进行寻址时, 如果再引入新的虚拟寄存器, 则会陷入递归而导致编译器崩溃. 因此, 用于寻址的指针 (栈指针、帧指针或基指针) 与栈帧中对象之间不能存在编译时期大小未知的区域.

原有栈帧布局下, 由于栈指针与分配固定大小对象 (包括局部变量和向量地址) 区域之间存在编译时期大小未知的可伸缩向量, 因此栈指针不能用于栈帧对象寻址. 在不需要栈地址对齐的时候, 可以用帧指针来寻址. 在需要对齐的时候, LLVM 会使用基指针来指向最后一个固定大小的对象, 通过基指针来寻址. 因此, 原有栈帧布局最少需要预留 2 个寄存器来保存栈指针和帧指针, 最多要预留 3 个寄存器来保存栈指针、帧指针和基指针. 尽管 RISC-V 有 32 个通用寄存器, 但对于寄存器压力较大的情况, 例如编译器内联优化时引入大量函数体代码的时候, 预留过多的寄存器容易导致寄存器溢出, 从而影响程序性能.

4 新的栈帧布局方案

为了减少读写栈向量对象使用的访存指令数, 并且减小函数使用的栈空间大小, 就需要摒弃保存每一个栈向量对象地址的栈帧布局方式. 由于 RISC-V 向量扩展架构支持在运行时获得向量寄存器的长度信息, 因此可以通过编译时插入算术指令来动态计算栈向量对象的地址. 按照这个思路, 我们提出了新的栈帧布局方案.

同时, 为了减少预留寄存器的个数, 新方案会根据是否存在可变大小局部变量, 是否需要栈地址对齐, 这些不同的场景对栈帧布局进行调整. 表 1 展示了不同场景下栈对象的寻址方式, 以及两种方案所需要预留的寄存器情况. 可以看到, 在没有可变大小局部变量的情况下, 新方案能够使用栈指针来寻址, 需要预留的寄存器数量要比原有方案少.

表 1 不同场景下的栈对象寻址方式及需要保存的指针

是否存在可变大小的局部变量	是否需要栈地址对齐	新方案使用哪种指针进行栈对象寻址	新方案需要预留寄存器保存的指针	原有方案需要预留寄存器保存的指针
否	否	栈指针	栈指针	栈指针、帧指针
否	是	栈指针	栈指针、帧指针	栈指针、帧指针、基指针
是	否	帧指针	栈指针、帧指针	栈指针、帧指针
是	是	基指针	栈指针、帧指针、基指针	栈指针、帧指针、基指针

4.1 不存在可变大小局部变量且不需要栈地址对齐

图 3 展示了不存在可变大小局部变量且不需要栈地址对齐时的栈帧布局. 栈向量对象区域位于被调用者保存寄存器区域与局部变量及溢出变量区域之间. 在这种场景下, 可以使用栈指针来寻址. 在栈帧布局的时候, 通过计算栈向量类型对象的个数和 *LMUL* 值, 以及读取控制状态寄存器 vlenb 的值, 可以在运行时动态计算出栈向量对象区域的长度大小值.

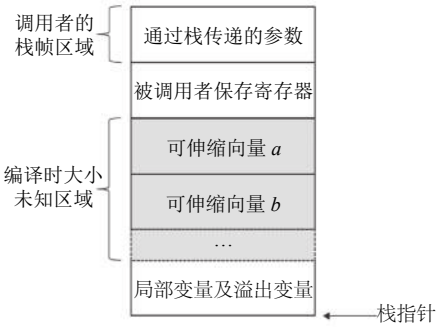

图 3 不存在可变大小局部变量且不需要栈地址对齐时的栈帧布局

LLVM 内部使用二维抽象数据类型来表示栈对象相对于寻址指针的偏移量. 其中第一个维度表示固定大小的偏移量, 第二个维度表示可伸缩大小的偏移量. 在 RISC-V 向量扩展中, 可伸缩偏移量表示的实际值是该偏移量值与控制状态寄存器 vlenb 的值的乘积. 因此, 在计算栈向量对象地址时, 其固定偏移量为局部变量及溢出变量区域的大小, 可伸缩偏移量则取决于该对象在栈向量对象区域中的位置.

4.2 不存在可变大小局部变量且需要栈地址对齐

图 4 展示了不存在可变大小局部变量且需要栈地址对齐时的栈帧布局. 由于需要栈地址对齐, 在被调用者保存寄存器区域和栈向量对象区域之间多出一块编译时未知的对齐区域, 因此, 需要预留寄存器来保存帧指针, 以便利用帧指针在函数退出时恢复栈帧. 在这种场景下, 可以使用栈指针来寻址. 栈向量对象相对于栈指针的固定偏移量为局部变量和溢出变量区域的大小, 可伸缩偏移量取决于栈向量对象在栈向量对象区域中的位置.

4.3 存在可变大小局部变量且不需要栈地址对齐

图 5 展示了存在可变大小局部变量且不需要栈地址对齐时的栈帧布局. 与之前不同的地方是, 我们调换了局部变量及溢出变量区域和栈向量对象区域的分配顺序, 将编译时大小未知区域连在一起, 从而可以避免引入基指针. 在这种场景下, 可以使用帧指针来寻址. 栈向量对象的固定偏移量为局部变量及溢出变量区域与被调用者保存寄存器区域的大小之和, 可伸缩偏移量则取决于该对象在栈向量对象区域中的位置.

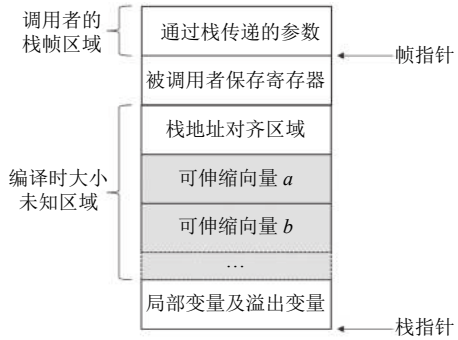

图 4 不存在可变大小局部变量且需要栈地址对齐时的栈帧布局

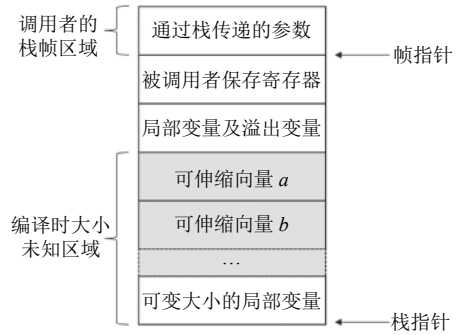

图 5 存在可变大小局部变量且不需要栈地址对齐时的栈帧布局

4.4 存在可变大小局部变量且需要栈地址对齐

图 6 展示了存在可变大小局部变量且需要栈地址对齐时的栈帧布局. 在这种场景下, 由于局部变量与栈指针和帧指针之间都存在编译时大小未知区域, 为了避免在寻址紧急溢出槽时递归地引入虚拟寄存器而导致编译器崩溃, 需要使用基指针来寻址. 此时栈向量对象相对于基指针的固定偏移量为局部变量及溢出变量区域的大小, 可伸缩偏移量则取决于该栈向量对象在分配栈向量对象区域中的位置.

4.5 通过指令动态计算栈向量对象地址

将 LLVM 中表示偏移量的抽象二维数据类型转化到实际的偏移量时, 需要通过多条指令来实现. 图 7 展示了计算固定偏移量为 32, 可伸缩偏移量为 8 时生

成的 3 条指令. 首先通过 csrr 指令读取控制状态寄存器 vlenb 的值, 随后通过 slli 指令得到该值与可伸缩偏移量 8 的乘积, 最后通过 addi 指令与固定偏移量相加得到该栈向量对象相对于栈指针的实际偏移量.

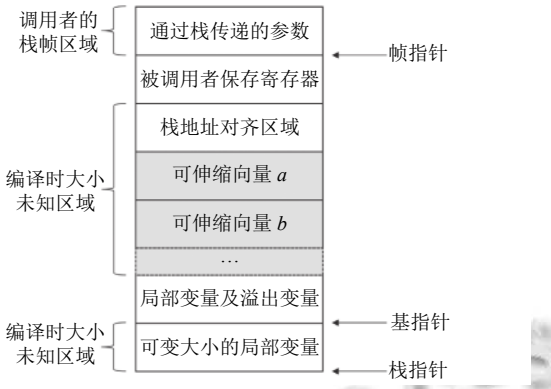

图 6 存在可变大小局部变量且需要栈地址对齐时的栈帧布局

```
csrr    a0,   vlenb
slli    a0,   a0,   3
addi    a0,   a0,   32
add     a0,   a0,   sp
vs8r.v  v8,   a0
```

图 7 计算二维数据 (32, 8) 时的指令

5 测试验证

我们对新方案基于 LLVM 11.0 版本进行了代码实现, 并测试通过了 LLVM 自带的回归测试集.

由于 RISC-V 向量扩展指令集标准尚处于草案阶段, 目前还没有适于做性能测试的硬件平台. 为了评估新方案对生成代码的优化效果, 我们使用了巴塞罗那超算中心为 RISC-V 向量扩展所开发的测试集[9,10], 该测试集主要测试的内容是向量的数学计算 (例如向量按元素乘加, 向量按元素取余弦值), 可以充分利用向量指令及寄存器. 我们对该测试集分别在-O0 优化级别 (即不做优化) 和-O2 优化级别下编译测试文件, 然后对生成代码的访存指令数量和栈空间大小进行静态统计.

图 8 展示了新方案下访存指令数量相对于原有方案下的百分比. 在-O0 优化级别下, 所有测试文件的访存指令数量均有减少, 其中使用向量类型变量较多的测试文件 CumNormalInv.cpp, 优化后的访存指令数量仅占原有方案下的 39.5%. 在-O2 优化级别下, 由于测试文件 axpy.c 使用的向量类型变量较少, 经过优化后都已全部放在向量寄存器中, 因此访存指令数没有变化. 其他测试文件的访存指令数量均有减少, 其中测试文件 CumNormalInv.cpp 在优化后的访存指令数量仅占原有方案下的 1.6%.

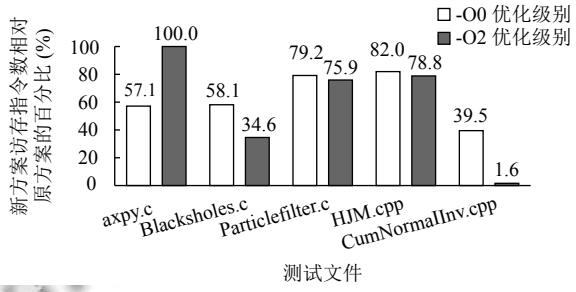

图 8 访存指令数对比

图 9 展示了新方案下栈空间大小相对于原有方案下的百分比. 在-O0 优化级别下, 对于所有的测试文件, 相比原有方案, 优化方案下栈空间的使用量均有所减少, 其中使用向量类型变量较多的文件 CumNormalInv.cpp, 优化后栈空间使用量为原有方案下的 49.2%. 在-O2 优化级别下, 除了 axpy.c 测试文件在栈上没有分配向量对象, 其他文件相比原有方案, 优化方案下栈空间的使用量均有减少, 其中测试文件 CumNormalInv.cpp 的栈空间使用量仅占原有方案下的 13.3%.

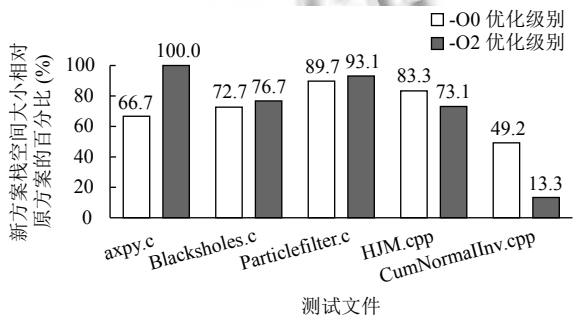

图 9 栈空间大小对比

图 10 展示了新方案下总指令数量相对于原有方案下的百分比. 在-O0 优化级别下, 对于所有测试文件, 新方案的总指令数量均有增加, 其中测试文件 axpy.c 新增指令最多, 占原有方案下的 112.5%. 在-O2 优化级别下, 测试文件 CumNormalInv.cpp 新增指令最多, 占原有方案下的 129.8%. 测试文件 Particlefilter.c 的总指令数量反而会下降, 占原有方案下的 90.0%. 这是由于原有方案下函数使用的栈空间比较大, 导致寻址偏移

量超过了 12 bit 的范围, 需要生成额外的指令来处理, 而优化后减小了栈空间, 因此不需要再生成这部分指令.

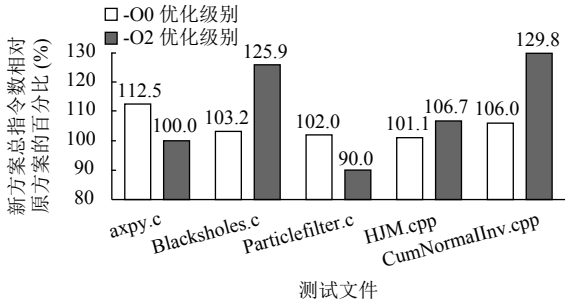

图 10 总指令数对比

表 2 展示了优化方案和原有方案下的预留寄存器数量对比, 由于-O0 优化级别下, 编译器不开启与寄存器使用相关的帧指针消除优化, 因此表 2 仅统计了-O2 优化级别下每个测试文件累计预留寄存器的数量. 可以看到, 优化方案在-O2 优化级别下能够减少预留寄存器的数量.

表 2 累计预留寄存器数量对比

测试文件	原有方案	优化方案
axpy.c	10	10
Blackscholes.c	60	48
Particlefilter.c	180	144
HJM.cpp	110	88
CumNormalInv.cpp	20	16

总的来说, 优化后的栈帧布局由于采用动态计算栈向量对象地址的方式, 所以能够减少保存和读取地址时的 load/store 指令, 由于不需要分配栈对象来保存栈向量对象地址, 所以能够减少栈空间的使用. 除此之外, 由于通过区分不同场景来确保紧急溢出槽始终紧邻用于寻址的指针, 所以相比原有栈帧布局方案, 能够减少预留寄存器的数量. 但是, 采用动态计算栈向量对象地址的方式会插入多条指令, 带来额外的运行时计算开销. 因此, 新方案最终优化效果如何, 还需要在实际的硬件平台上通过运行测试来评测.

6 结语

本文介绍了原有 LLVM 中 RISC-V 向量扩展的栈帧布局, 并针对原有栈帧布局中存在的问题, 提出了优化方案. 通过巴塞罗那超算中心开发的测试集, 验证了优化方案的正确性, 并通过静态统计访存指令数量以及栈空间大小, 证明了优化方案在不同优化级别下能有效减少访存指令数量以及栈空间的使用量. 目前, 优化后的栈帧布局代码实现已经提交到 LLVM 官方仓库[11].

参考文献

1 Lattner C, Adve V. The LLVM compiler framework and infrastructure tutorial. In: Eigenmann R, Li Z, Midkiff SP, eds. Languages and Compilers for High Performance Computing. Berlin: Springer, 2004. 15–16.

2 LLVM 官方网站. https://llvm.org. [2021-06-08].

3 Waterman A, Lee Y, Avizienis R, *et al*. The RISC-V instruction set. 2013 IEEE Hot Chips 25 Symposium (HCS). Stanford: IEEE, 2013. 1.

4 Asanović K. Vector microprocessors[Ph. D. Thesis]. Berkeley: University of California, 1998.

5 RISC-V 向量扩展规范. https://github.com/riscv/riscv-v-spec. [2021-06-08].

6 Aho AV, Lam MS, Sethi R, *et al*. Compilers: Principles, Techniques, and Tools. 2nd ed. Boston: Prentice Hall, 2006.

7 Bryant RE, O'Hallaron DR. Computer Systems: A Programmer's Perspective. 3rd ed. Upper Saddle River: Pearson, 2015.

8 Gauthier L, Ishihara T. Optimal stack frame placement and transfer for energy reduction targeting embedded processors with scratch-pad memories. 2009 IEEE/ACM/IFIP 7th Workshop on Embedded Systems for Real-Time Multimedia. Grenoble: IEEE, 2009. 116–125. [doi: 10.1109/ESTMED.2009.5336819]

9 巴塞罗那超算中心 RVV 测试集. https://github.com/RALC88/riscv-vectorized-benchmark-suite/tree/rvv-1.0. [2021-06-08].

10 Ramírez C, Hernandez CA, Palomar O, *et al*. A RISC-V simulator and benchmark suite for designing and evaluating vector architectures. ACM Transactions on Architecture and Code Optimization, 2020, 17(4): 38.

11 向 LLVM 官方仓库提交的代码实现. https://reviews.llvm.org/rGa9b9c64fd4c8d456f11dcbd409cdd62116eb021f. [2021-06-08].