计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(11):29−35 [doi: 10.15888/j.cnki.csa.009332]

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

基于 musl libc 库的 RVV 优化①

张　飞[2], 于佳耕[1], 邢明杰[1], 武延军[1]

[1](中国科学院 软件研究所, 北京 100190)
[2](中科南京软件技术研究院 智能软件研究中心, 南京 211100)
通信作者: 于佳耕, E-mail: jiageng08@iscas.ac.cn

摘　要: musl libc 是一个轻量级的标准 C 库, 其代码库小巧, 提供了全面的 POSIX 接口支持, 具有高度可移植性并支持多种架构和操作系统, 被广泛用于嵌入式系统、网络服务器、容器等领域. RISC-V 指令集作为一种开源的指令集, 目前发布了相对稳定的 SIMD 指令集, RISC-V 生态软件环境也迎来了新的优化热潮, 但是对于 musl libc 库 RVV 扩展优化还是一片空白. 本文立足于 musl libc 基础库和 RISC-V RVV 扩展指令集的协同研究点, 提出了兼容基础指令集和向量扩展指令集的实现方案, 利用向量扩展指令集优化了常见的 C 库函数 strlen 和 memset, 并在 gem5 模拟器上进行了对比分析, 实验结果表明, 相较于 C 语言实现, 在性能方面, 利用 RVV 优化的 strlen 函数平均提升 83%–703%, memset 函数平均提升 85%–334%.

关键词: musl libc; RISC-V; 基础指令集; RVV 扩展优化

引用格式: 张飞,于佳耕,邢明杰,武延军.基于 musl libc 库的 RVV 优化.计算机系统应用,2023,32(11):29–35. http://www.c-s-a.org.cn/1003-3254/9332.html

RVV Optimization Based on musl libc Library

ZHANG Fei[2], YU Jia-Geng[1], XING Ming-Jie[1], WU Yan-Jun[1]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](Intelligent Software Research Center, Nanjing Institute of Software Technology, Nanjing 211100, China)

Abstract: As a lightweight standard C library, musl libc features a small code base providing comprehensive POSIX interface support, and high portability and support for various architectures and operating systems. It is widely employed in embedded systems, Web servers, containers, and other fields. RISC-V instruction set is an open source instruction set that has released the relatively stable SIMD instruction set at present. Meanwhile, the RISC-V ecological software environment has ushered in a new optimization boom, but the RVV extension optimization of the musl libc library is still a research gap. Based on the collaborative research of the musl libc basic library and RISC-V RVV extended instruction set, this study proposes an implementation scheme compatible with the basic instruction set and vector extended instruction set. The common C library functions strlen and memset are optimized by the vector extended instruction set, and comparative analysis is carried out on gem5 simulator. The experimental results show that compared with the implementation of C language, the performance of strlen function optimized by RVV is improved by 83%–703% on average, and that of memset function is improved by 85%–334% on average.

Key words: musl libc; RISC-V; basic instruction set; RVV extension optimization

① 本文由“RISC-V 技术及生态”专题特约编辑邢明杰高级工程师、宋威副研究员、张科正高级工程师以及易秋萍副教授推荐.
收稿时间: 2023-05-26; 修改时间: 2023-06-27; 采用时间: 2023-07-21; csa 在线出版时间: 2023-09-21
CNKI 网络首发时间: 2023-09-23

1 引言

SIMD (single instruction multiple data) 技术[1,2] 是一种基于向量运算的并行计算技术, 可以同时对多个数据进行相同的操作. 它通过将多个数据打包成一个向量, 然后对整个向量进行计算, 从而实现了单个指令周期内并行处理多个数据的能力. 在计算机科学领域, SIMD 技术最初用于数字信号处理、图像处理和视频编解码等领域, 以提高运算速度和效率. 现在, 随着计算机硬件的发展, SIMD 技术也被广泛应用于其他领域, 如科学计算、机器学习和数据处理等. 除了向量寄存器, 现代 CPU 还配备了许多 SIMD 指令, 以便处理不同的数据类型和操作. 例如, ARM 架构的 NEON 指令集、Intel 架构的 MMX/SSE 指令集、RISC-V 架构的向量指令集等.

RISC-V 指令集[3] 是一种新兴的开源指令集, 旨在提供一个可扩展、可定制、高效的架构. 它的发展始于 2010 年, 起源于加州大学伯克利分校 (UC Berkeley) 的一个研究项目. 该指令集最初是为了提供一个适合教学和研究的指令集而设计的, 但现在已经成为一个面向商业应用的实用指令集. RISC-V 指令集包含基础指令集和扩展指令集两部分, 包括基础指令集 I、乘法和除法指令集 M、原子指令集 A、单精度浮点指令集 F、双精度浮点指令集 D、位操作指令集 B、向量操作指令集 V 等.

musl libc[4] 是一个轻量级的 C 标准库实现, 旨在提供高性能、低复杂度、可移植性和安全性. 与其他主流的 C 标准库实现 (如 glibc 和 uclibc) 相比, musl libc 具有更小的代码库和更快的启动时间, 同时也更容易维护和制定. 由于其小巧、高效和安全的特性, musl libc 被广泛应用于嵌入式系统、操作系统和各种应用中. musl libc 支持多种体系结构, 包括 X86、ARM、MIPS、PowerPC、RISC-V 等. 目前 RISC-V 的向量扩展指令集仍处于冻结状态, 虽然有了编译工具链和模拟器的向量支持, 但对于 musl libc 库的向量扩展优化还不够成熟, 软件生态亟待发展.

字符串与内存操作函数在许多应用程序中都会被频繁调用, 往往会涉及大量的数据处理任务, 且对于性能要求较高. 在 glibc、musl libc 中 X86、ARM 等架构均使用汇编语言更加高效地实现了这些操作, 而对于 RISC-V 架构仍使用的是 C 语言实现的函数. strlen 与 memset 函数是字符串与内存操作函数中常被向量化优化的函数之一, 因此, 本文选取这两个函数进行向量扩展优化, 并利用内置宏解决硬件不同导致的指令集支持具有差异的兼容性问题, 采用了基础指令集和向量指令集的两种函数汇编实现, 最大程度地提升字符串函数和内存操作函数的执行效率和性能, 并在 gem5 模拟器上进行了性能测试.

2 RISC-V 向量扩展指令集

RISC-V 是一个新兴的指令集架构, 吸引了广泛的关注和采用[5,6]. RISC-V 指令集由基础指令集和扩展指令集组成[7]. 基础指令集中定义了 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 1 所示.

表 1 通用寄存器使用规范

Register	ABI name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5–x7	t0–t2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10–x11	a0–a1	Function arguments/return values
x12–x17	a2–a7	Function arguments
x18–x27	s2–s11	Saved register
x28–x31	t3–t6	Temporaries

在扩展指令集中, RISC-V 的向量指令集 (RVV) 为其提供了高效的 SIMD 支持[8,9]. RVV 是一个独立的寄存器文件, 支持 8 位、16 位、32 位、64 位等数据类型的向量运算. 向量扩展增加了 32 个向量寄存器 v0–v31, 每一个向量寄存器有 VLEN 的位宽, 以及 7 个非特权状态控制寄存器, 如表 2 所示. RISC-V 向量指令集具有 3 个特性: 向量长度不可知、寄存器分组、向量指令支持掩码操作.

表 2 向量状态控制寄存器

Address	Privilege	Name	Description
0x008	URW	vstart	Vector start position
0x009	URW	vxsat	Fixed-point saturate flag
0x00A	URW	vxrm	Fixed-point rounding mode
0x00F	URW	vcsr	Vector control and status register
0xC20	URO	vl	Vector length
0xC21	URO	vtype	Vector data type register
0xC22	URO	vlenb	Vector register length in bytes

2.1 向量长度不可知

RISC-V 的向量寄存器位宽 VLEN 是由具体的硬件实现来定义的, VLEN 的范围在 2^5–2^{16} 之间.

由于 VLEN 的值在编译阶段无法获得, 因此用户可以通过 csrr 指令读取控制状态寄存器 vlenb 中的值, 从而获得向量寄存器的位宽. 在 SIMD 编程中, 往往需要考虑尾部处理, RISC-V 中可使用配置指令 vsetvli 或者 vsetvl 动态配置运行时向量的宽度, 可通过设置 vl 修改感兴趣的向量长度, 这给自动向量化以及向量化编程带来了极大的便利.

2.2 寄存器分组

支持寄存器分组是 RISC-V 向量扩展指令集 (RVV) 的一个重要特性, 它为向量操作提供了更灵活的数据宽度选择. 在传统的向量指令集中, 向量长度和数据宽度通常是固定的, 而在 RVV 中, 向量长度和数据宽度都可以由软件在运行时进行配置. 这为编程者提供了更大的灵活性和自由度, 使得向量计算可以更好地适应不同的场景和应用需求. RISC-V 的向量寄存器组支持不同数量的寄存器, 包括 1 个、2 个、4 个和 8 个, 这取决于 LMUL 参数的设置[10]. 寄存器组的作用是扩展寄存器的位宽, 但相应地会减少可用的寄存器数量. 举个例子, 当 LMUL 设置为 8 时, 只有 v0、v8、v16 和 v24 这 4 个寄存器可用.

如果指令使用了不存在的向量寄存器 (如 v1), 将会引发非法指令异常. 图 1 和图 2 是分别设置 LMUL 为 4 和 8 时寄存器分布情况.

(a) v0 寄存器组 (b) v8 寄存器组 (c) v16 寄存器组 (d) v24 寄存器组

图 1 VLEN=128 b, LMUL=8 时寄存器分布

2.3 掩码操作

RISC-V 向量指令掩码操作是一种在 RISC-V 架构中使用掩码向量来控制向量操作的技术. 可以实现对向量数据的选择性处理, 提高数据处理的灵活性和效率. 掩码操作的基本原理是使用一个掩码向量来指示对应位置的操作是否应该执行. 掩码向量的每个元素表示对应位置的操作是否有效, 当掩码向量中的元素为非零值时, 对应位置的操作将被执行; 当掩码向量中的元素为零值时, 对应位置的操作将被跳过. 这种灵活的条件选择机制使得向量操作可以根据实际需求进行动态配置和选择性处理.

(a) v0 寄存器组 (b) v8 寄存器组 (c) v16 寄存器组 (d) v24 寄存器组

(e) v4 寄存器组 (f) v12 寄存器组 (g) v20 寄存器组 (h) v28 寄存器组

图 2 VLEN=128 b, LMUL=4 时寄存器分布

3 基于 musl libc 的 RVV 优化

字符串与内存处理函数在 ARM、X86 等架构中均有汇编实现, 是基础 C 库常优化的函数, 但对于 RISC-V, 此方面的优化还很欠缺, 因此本文选取常见的函数 strlen、memset 进行优化, 详述了优化原理.

3.1 RVV 兼容方案

由于 RISC-V 指令集的可扩展性, 这导致了向量指令优化的函数无法运行在不支持向量扩展的硬件上. 目前, 在 Linux 系统中常用 hwcap (hardware capabilities) 机制检测和标识硬件的特性和功能. hwcap 机制通过定义了一组位标志来表示不同的硬件功能. 这些位标志被编码为一个或多个特定的寄存器或内存位置, 并且由操作系统内核在系统启动时进行设置. 由于目前 RISC-V 内核 hwcap 机制支持不够完善, 且运行时动态读取硬件配置会带来额外的开销, 因此, 本文使用 gcc 预定义宏__riscv_vector 在编译阶段检查是否支持向量指令集扩展, 以决定是否使用向量化代码路径来提高性能, 同时实现仅包含基础指令集和 RVV 优化的两个版本汇编. 具体兼容实现如下所示.

代码清单 1. RVV 兼容实现

```
#ifdef __riscv_vector
  //RVV 优化汇编实现
#else
  //基础指令汇编实现
#endif
```

3.2 strlen 函数优化

3.2.1 strlen 基础指令集实现

strlen 函数是 C 标准库中的一个字符串处理函数, 用于计算一个字符串的长度 (即字符的个数), 不包括字符串末尾的空字符 (“\0”). 它的函数原型为 size_t strlen(const char* str), 参数 str 是一个指向以空字符结尾的字符串的指针[11]. 函数会从给定的字符串的开头开始遍历, 直到遇到空字符为止, 然后返回计算得到的字符串长度作为无符号整数 (size_t 类型).

实现一个 strlen 的函数功能, 最简单的就是逐字节进行判断是否为终止符“\0”, 这种做法实现逻辑相对简单且容易理解, 但当字符串较长时会增加循环次数导致性能较低, 逐字节判断无法充分利用现代处理器的并行能力, 无法同时处理多个字符. 因此, 本文同时处理 8 个字节, 利用魔法数 0xfefefefefefefeff 和 0x8080808080808080 更加快速高效地找到终止符并返回字符串的长度, 算法流程图如图 3 所示.

利用基础指令实现的 strlen 函数算法如算法 1.

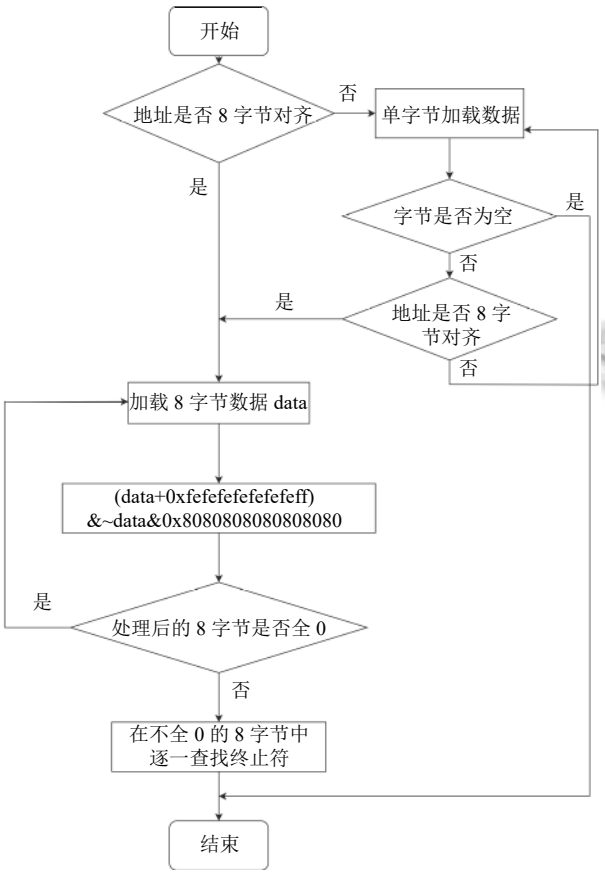

图 3 strlen 基础指令实现流程图

算法 1. 基础指令实现 strlen 函数

(1) 首先判断字符串首地址是否为 8 字节对齐;
(2) 若首地址不对齐, 则先逐一取单个字节数据进行判断是否为终止符, 若是终止符则直接返回长度; 否则直至取到地址 8 字节对齐处, 然后跳转步骤 (3);
(3) 若首地址对齐, 直接取长度为 8 字节的数据 data, 然后利用魔法数对 data 进行处理 (data+0xfefefefefefefeff)&~data&0x8080808080808080, 若处理后的 8 字节仍为空, 则继续循环查找下一个 8 字节数据; 若不为空, 则 8 字节中存在终止符, 在这个 8 字节中逐一查找终止符, 得到字符串长度返回.

3.2.2 strlen RVV 指令集实现

利用魔法数虽能够一次处理 8 个字节的数据, 但随着 RISC-V RVV 指令集的发布, 这对字符串类函数优化有了新的方向. RVV 扩展提供了丰富的向量操作指令, 包括向量加载和存储指令、向量算数指令、向量掩码操作等, 这些指令能够更加有效地利用数据并行性, 减少指令的数量和执行时间. 结合 RVV 扩展指令的丰富性, strlen 向量指令集实现算法流程图如图 4 所示.

利用 RVV 指令实现的 strlen 函数算法如算法 2.

算法 2. RVV 指令实现 strlen 函数

(1) 首先读取 vlenb 寄存器得到向量寄存器位宽, 判断字符串首地址是否按向量寄存器位宽对齐;
(2) 若首地址不对齐, 则先逐一取单个字节数据进行判断是否为终止符, 若是终止符则直接返回长度; 否则直至取到向量寄存器位宽对齐处, 跳转至步骤 (3);
(3) 若首地址对齐, 设置 vl 寄存器为最大向量长度 (参数 SEW=8、LMUL=8), 加载 LMUL×VLEN 长数据并和 0 比较, 相同则置 1、不同置 0, 查找比较后的结果是否存在 1, 有则表明取出的数据包含终止符, 计算字符串长度返回, 没有则继续循环.

3.3 memset 函数优化

3.3.1 memset 基础指令集实现

memset 函数是 C 标准库中的一个常用的内存操作函数. 它的作用是将内存区域的每个字节都赋值为相同的值, 常用于初始化内存、清零内存或填充内存区域. memset 函数的函数原型为 void *memset(void *s, int c, size_t n), 其中 s 是要设置值的内存起始地址, c 是要设置的值, n 是要设置的字节数.

由于 memset 函数设置的字节数不固定, 往往函数实现时会根据不同的数据量采用不同位宽的字节操作指令, 比如 musl 中 memset.c 中其根据数据量大小分别采用了单字节存储、4 字节存储、8 字节存储. 本文利用 RISC-V 指令特性以及编译优化技术, 采用基础指令实现更加高效的 memset 函数.

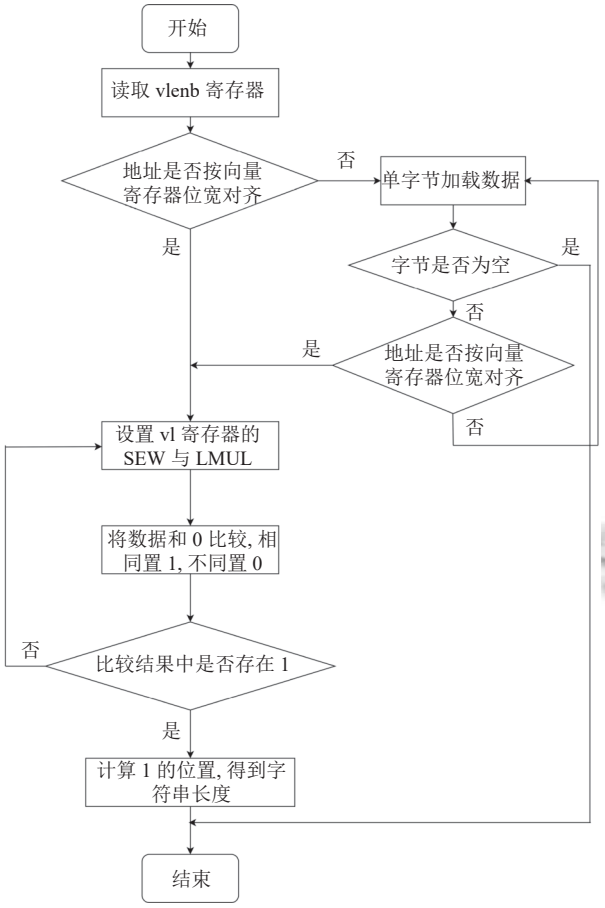

图 4 strlen RVV 指令实现流程图

循环展开: 在 memset 函数中, 循环展开的目的是将字节填充操作展开成多个重复的指令序列, 以提高执行效率. 通过将循环体内的代码重复多次, 减少了循环的迭代次数, 降低了循环控制开销, 同时利用处理器的指令并行和高速缓存, 改善了内存访问模式, 减少对内存的访问延迟. 因此, 本文针对核心循环段展开 32 次, 核心循环代码段如下所示.

代码清单 2. memset 循环展开

```
loop:
    sd a1, 0(t0)
    sd a1, 8(t0)
    sd a1, 2*8(t0)
    sd a1, 3*8(t0)
    … //循环展开
    sd a1, 29*8(t0)
    sd a1, 30*8(t0)
    sd a1, 31*8(t0)
    sd t0, t0, 32*8
    bltu t0, a3, loop
```

地址跳转: 由于循环展开了 32 次, 则要求数据量至少是 256 字节, 因此本文针对数据量小于 256 字节时, 计算其需要展开的次数以及与 loop 循环段的地址偏移, 直接跳转到 loop 内执行, 这样数据量小于 256 字节时仍能够循环展开, 部分代码如下所示.

代码清单 3. memset 地址跳转

```
/* 判断能否做 32 次循环展开 */
andi a4, a4, 31*8
beqz a4, loop
/*计算与 loop 段的偏移*/
neg a4, a4
addi a4, a4, 32*8
sub t0, t0, a4
/* 加载 loop 循环段地址, 并跳转至 loop 内*/
la a5, loop
srli a4, a4, 1
add a5, a5, a4
jr a5
```

尾部处理: 当剩余字节无法循环展开存储, 往往采用的方式是逐字节存储, 本文利用双指针思想, 从剩余字节的头和尾部进行存储, 这种做法虽然会产生重复的存储, 但指令并行与减少跳转次数会带来更大的收益, 部分代码如下所示.

代码清单 4. memset 尾部处理

```
sb a1, 0(t0) //头部存储
sb a1, –1(a3) //尾部存储
li a4, 2 //数据量判断
bgeu a4, a2, 6f
…
sb a1, 5(t0)
sb a1, 6(t0)
sb a1, –6(a3)
sb a1, –7(a3)
li a4, 14
bgeu a4, a2, 6f
sb a1, 7(t0)
```

3.3.2 memset RVV 指令集实现

RVV 指令集特性以及传输指令使得 memset 向量化编程实现更为简洁化. 其中 vmv 指令能够高效地在向量寄存器之间以及向量与标量之间传输和复制数据, 同时配合 vsetvli 指令控制复制的个数, 就不需要像 memset 基础指令集实现那样考虑数据量不同的处理和尾部处理. 核心循环代码如下.

代码清单 5. memset RVV 指令集实现

```
loop:
```

```
vsetvli t1, a2, e8, m8, ta, ma
vmv.v.x v0, a1
sub a2, a2, t1
vse8.v v0, (t0)
add t0, t0, t1
bnez a2, loop
```

4 测试结果及分析

由于目前 RISC-V 硬件支持 RVV 不够完善, gem5 上游中科院软件所 PLCT 实验室提交的 RVV 补丁正在 review, 还未完全合入, 因此本文使用了下游中科院软件所 PLCT 实验室 gem5 的代码仓[12], 并在 gem5 模拟器上测试, 具有一定的参考价值. 测试集采用了 ARM 官方提供的测试用例[13], 由于该测试集中包含了对 ARM 接口函数的调用, 本文将其注释以便在模拟器上运行.

4.1 strlen 函数性能测试

性能测试结果如表 3–表 5 所示, 其中 small aligned 测试了首地址对齐且数据量较小时的性能、small unaligned 测试了首地址不对齐且数据量较小时的性能、medium 测试了数据量中等时的性能, 值越大代表着性能越好.

表 3 small aligned strlen 性能对比

数据量 (B)	musl C语言实现 (B/ns)	基础指令汇编实现 (B/ns)	RVV汇编实现 (B/ns)
1	0.04	0.04	0.03
2	0.08	0.08	0.07
4	0.13	0.14	0.14
8	0.21	0.24	0.27
16	0.38	0.43	0.55
32	0.62	0.73	1.10
64	0.92	1.10	2.20
平均	0.34	0.39	0.62

表 4 small unaligned strlen 性能对比

数据量 (B)	musl C语言实现 (B/ns)	基础指令汇编实现 (B/ns)	RVV汇编实现 (B/ns)
1	0.05	0.05	0.03
2	0.09	0.09	0.07
4	0.14	0.14	0.13
8	0.17	0.17	0.26
16	0.31	0.32	0.52
32	0.53	0.57	1.05
64	0.81	0.91	2.09
平均	0.30	0.32	0.59

从测试结果来看, 基础指令实现的 strlen 与 musl C 语言实现性能相当, 这是因为两者的算法相同导致, 这对于只支持基础指令集的 RISC-V 硬件也可以获得与 C 实现相当的性能; RVV 实现的 strlen 相比于 C 实现, small aligned 测试性能平均提升 83%, small unaligned 测试性能平均提升 98%, medium 测试性能平均提升 703%.

表 5 medium strlen 性能对比

数据量	musl C语言实现 (B/ns)	基础指令汇编实现 (B/ns)	RVV汇编实现 (B/ns)
128 B	1.23	1.51	4.46
256 B	1.45	1.82	9.28
512 B	1.60	2.03	12.46
1 KB	1.68	2.15	15.04
2 KB	1.73	2.22	16.78
4 KB	1.75	2.25	17.80
平均	1.57	2.00	12.64

4.2 memset 函数性能测试

性能测试结果如表 6–表 8 所示, 其中 random 测试了在对应数据量附近时的性能、medium 测试了数据量中等时的性能、large 测试了数据量较大时的性能, 值越大代表着性能越好.

表 6 random memset 性能对比

数据量 (KB)	musl C语言实现 (B/ns)	基础指令汇编实现 (B/ns)	RVV汇编实现 (B/ns)
32	0.67	0.81	1.25
64	0.67	0.80	1.24
128	0.68	0.81	1.25
256	0.67	0.80	1.24
512	0.67	0.80	1.24
1 024	0.67	0.80	1.23
平均	0.67	0.80	1.24

表 7 medium memset 性能对比

数据量 (B)	musl C语言实现 (B/ns)	基础指令汇编实现 (B/ns)	RVV汇编实现 (B/ns)
8	0.39	0.36	0.53
16	0.49	0.54	1.06
32	0.75	1.05	2.12
64	1.32	1.96	3.97
128	2.26	3.05	7.06
256	3.53	6.72	11.57
512	4.89	9.29	17.59
平均	1.95	3.28	6.27

从测试结果来看, 无论是基础指令还是 RVV 实现的 memset, 其性能均好于 musl 库中 C 语言实现的 memset, 一方面基础指令实现的性能提升得益于巧妙地利用循环展开、地址跳转、尾部处理等编程优化,

而这些对于编译器则是无法生成的, 在 random 测试中, 基础指令集实现相比于 C 实现性能提升了 20%, 在 medium 测试中性能提升了 69%, 在 large 测试中性能提升了 88%; 另一方面 RVV 实现的 memset 性能最优, 得益于 RVV 指令集的丰富性, 能够最大粒度地并行处理数据, 在 random 测试中, RVV 实现相比于 C 实现性能提升了 85%, 在 medium 测试中性能提升了 222%, 在 large 测试中性能提升了 334%.

表 8 large memset 性能对比

数据量 (KB)	musl C语言实现 (B/ns)	基础指令汇编实现 (B/ns)	RVV汇编实现 (B/ns)
1	6.02	11.30	22.94
2	6.87	12.91	28.19
4	7.39	13.90	31.84
8	7.68	14.46	34.04
16	7.84	14.75	35.26
32	7.92	14.90	35.90
64	7.96	14.98	36.23
平均	7.38	13.89	32.06

5 结语

针对目前 musl libc 库 RVV 扩展优化还不完善, 提出了基础指令集与 RVV 扩展实现并存的解决方案, 详细介绍了 strlen 与 memset 函数的基础指令集与 RVV 扩展实现的算法流程, 从实验结果看出, RVV 优化的函数具有很大的性能提升. 由于基础 C 库的函数还有很多, 在未来工作中将结合 RVV 扩展优化其他函数, 丰富 RISC-V 基础软件生态.

参考文献

1 冯竞舸, 贺也平, 陶秋铭. 自动向量化: 近期进展与展望. 通信学报, 2022, 43(3): 180–195. [doi: 10.11959/j.issn.1000-436x.2022051]

2 高伟. 面向 SIMD 的自动向量化优化技术研究 [硕士学位论文]. 郑州: 解放军信息工程大学, 2013.

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

4 musl libc 官网. http://www.musl-libc.org/. (2023-05-01) [2023-05-05].

5 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]

6 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]

7 刘畅, 武延军, 吴敬征, 等. RISC-V 指令集架构研究综述. 软件学报, 2021, 32(12): 3992–4024. [doi: 10.13328/j.cnki.jos.006490]

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

9 RISC-V 向量扩展规范. https://github.com/riscv/riscv-v-spec. (2021-09-18)[2023-05-05].

10 叶锡聪, 庄灿锋, 王宇木, 等. RISC-V 向量指令集的 Compute Library 函数库移植. 单片机与嵌入式系统应用, 2021, 21(1): 8–13.

11 李恺, 翁玉萍. 基于龙芯 2F 的 Glibc 库优化. 电子技术, 2010, 37(10): 27–29.

12 gem5 模拟器. https://github.com/plctlab/plct-gem5. (2023-04-25) [2023-05-05].

13 ARM 官方测试集. https://github.com/ARM-software/optimized-routines/tree/master/string/bench. (2022-02-10)[2023-05-05].

(校对责编: 孙君艳)