计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(6): 1469−1491, 2025

SIMD-to-RVV 动态二进制翻译中的跨架构编程模型适配优化

赖远明[1,2] 李亚龙[1,3] 胡瀚之[1,2] 谢梦瑶[1,2] 王 喆[1,2] 武成岗[1,2]
[1]（处理器芯片全国重点实验室（中国科学院计算技术研究所） 北京 100190）
[2]（中国科学院大学 北京 100049）
[3]（郑州大学河南先进技术研究院 郑州 450002）
（laiyuanming@ict.ac.cn）

Optimizing Cross-Architecture Programming Model Adaptation in SIMD-to-RVV Dynamic Binary Translation

Lai Yuanming[1,2], Li Yalong[1,3], Hu Hanzhi[1,2], Xie Mengyao[1,2], Wang Zhe[1,2], and Wu Chenggang[1,2]
[1]（*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190）
[2]（*University of Chinese Academy of Sciences*, *Beijing* 100049）
[3]（*Henan Institute of Advanced Technology*, *Zhengzhou University*, *Zhengzhou* 450002）

Abstract RISC-V, renowned for its open-source nature and modular design, has achieved remarkable success in embedded systems and is progressively expanding into the high-performance computing (HPC) domain. While RISC-V hardware tailored for HPC, such as the Sophon SG2042 multi-core processors, has demonstrated performance comparable to x86/ARM counterparts, its underdeveloped software ecosystem remains a critical barrier to broader adoption. To address this challenge, we develop RVBT, a process-level dynamic binary translator for RISC-V, designed to bridge the software gap by efficiently porting the mature x86 ecosystem to RISC-V platforms, thereby accelerating RISC-V's integration into HPC applications. Focusing on the pervasive use of SIMD instructions in HPC programs, we tackle the inefficiencies arising from fundamental differences in programming models between x86 SIMD and RISC-V vector (RVV) extensions. Specifically, x86 SIMD hardcodes data types within opcodes, whereas RVV dynamically configures vtype and mask registers, leading to redundant operations during direct translation. To overcome this, we propose three innovative optimizations to achieve: 1) Redundancy elimination via data type locality. By leveraging the locality of data types in adjacent SIMD operations, we statically analyze and remove redundant configurations of vtype (achieving 100% dynamic elimination rates for csrr and vsetvl, and 56.31% for vsetvli) and mask settings (74.66% elimination rate in floating-point benchmarks). 2) Hybrid translation with on-demand synchronization. We decouple scalar and vectorized floating-point operations, translating x86 SIMD scalar double-precision instructions to RISC-V's floating-point extensions and reserving RVV for vectorized operations. Data synchronization between scalar and vector registers is optimized through def-use analysis, achieving a 67.35% dynamic synchronization reduction in floating-point benchmarks. Experimental results on SPEC CPU 2006 demonstrate significant improvements on the optimized RVBT, achieving 47.39% and 40.06% of native execution efficiency for integer and floating-point benchmarks, respectively, representing speedups of 1.21 times and 8.31 times over the unoptimized version. RVBT vastly outperforms QEMU (18.84% and 4.81% for integer and floating-point), with floating-point efficiency surpassing QEMU by 8.33 times, highlighting its potential for deployment in specific HPC scenarios. Crucially, these optimizations are architecture-agnostic: The methodology of exploiting data type

收稿日期：2025-03-01；**修回日期**：2025-04-09
通信作者：王喆（wangzhe12@ict.ac.cn）

locality, hybrid instruction translation, and adaptive synchronization apply equally to ARM SIMD (e.g., NEON) to RVV translation, offering a universal framework for cross-ISA binary compatibility. This work provides a pivotal technical foundation for breaking the software ecosystem deadlock and advancing RISC-V's role in HPC.

Key words binary translation; RISC-V vector extension; x86 SIMD; cross-architecture programming model adaptation; floating-point computation; redundant configuration elimination; hybrid translation

摘 要 RISC-V 因其开源和模块化设计等特性,已在嵌入式领域取得显著成功,并逐步向高性能计算(HPC)领域拓展. 面向 HPC 的 RISC-V 硬件(如 Sophon SG2042 多核处理器)已展现出与 x86/ARM 同类型产品相当的性能水平,但不完善的软件生态是阻碍其发展的最大障碍之一. 开发了面向 RISC-V 的进程级动态二进制翻译(DBT)器 RVBT,用于将成熟的 x86 软件生态移植到 RISC-V 平台,加速 RISC-V 在 HPC 领域的应用进程. 针对 HPC 程序广泛依赖 SIMD 指令的特性,聚焦于解决 SIMD 与 RVV 间显著的编程模型差异导致的翻译性能瓶颈问题,提出了 3 项创新的优化方案. x86 SIMD 将数据类型硬编码于操作码,而 RVV 需动态配置 vtype 和掩码寄存器,这导致直接翻译产生了大量冗余操作,严重拉低了翻译运行的效率. 通过充分利用程序数据类型的局部性,优化方案可删除跨架构适配编程模型导致的冗余设置,混合使用浮点扩展和向量扩展翻译 SIMD 指令并按需同步数据,大幅提升了 SIMD 指令的翻译运行效率. 3 项优化方案具备通用性,也适用于 ARM 平台的 SIMD 到 RVV 的翻译. 实验表明,以 SPEC CPU 2006 作为测试集,优化方案对 csrr, vsetvl, vsetvli 指令的平均动态消除率分别达到了 100%, 100% 和 56.31%,在浮点测试集上,掩码设置操作的平均动态消除率达到了 74.66%,数据的平均动态同步率为 67.35%. 优化后的 RVBT 在整点和浮点测试集上的平均运行效率达到了本地执行的 47.39% 和 40.06%,相比优化前的加速比分别为 1.21 和 8.31,并远超 QEMU 18.84% 和 4.81%,展现出了应用于部分 HPC 场景的潜力.

关键词 二进制翻译;RISC-V 向量扩展;x86 SIMD;跨架构编程模型适配;浮点计算;冗余设置消除;混合翻译

中图法分类号 TP314

DOI: 10.7544/issn1000-1239.202550135 **CSTR:** 32373.14.issn1000-1239.202550135

作为开源指令集架构的典型代表,RISC-V(reduced instruction set computer-V)自 2010 年诞生以来,凭借其模块化设计、可扩展性及免授权费等特性[1],迅速获得了学术界和产业界的广泛关注. 在 2022 年,RISC-V 处理器累计出货量就已经突破了 100 亿颗[2],预计 2025 年将达到 624 亿颗[3]. RISC-V 早期主要应用于嵌入式领域,但近年来其在高性能计算(HPC)领域的潜力逐渐显现,研究人员开始探索和推动它在该领域上的应用[4-5]. 已有多项研究表明 RISC-V 在 HPC 上已具备一定的成熟度:西班牙巴塞罗纳超算中心尝试基于 RISC-V 指令集设计软硬协同设计的超级计算机[6], Monte Cimone[7] 项目建成了第一个完全可运行且支持基础 HPC 软件栈的 RISC-V 集群. 产业界也尝试在 HPC 场景中使用 RISC-V 处理器. 2019 年在阿里云数据中心部署的玄铁 910 处理器验证了 RISC-V 在加速特定应用和云计算场景上的可行性[8]. 2023 年发布的 Sophon SG2042 64 核处理器在计算密集型任务中已展现出与 x86/ARM 竞品相当的性能水平[9]. 基于 RISC-V 的高性能处理器正在高速发展[9-13].

然而,RISC-V 在 HPC 领域的推广面临显著的软件生态瓶颈[2]. 尽管主流 Linux 发行版(Ubuntu, Fedora 等)和国产操作系统(麒麟、欧拉、龙蜥等)已提供了基础支持,但在关键应用层(如 Docker, Kubernetes)和基础软件(如 MongoDB, TiDB)的适配上仍存在明显的滞后[14-16],对编译器和常用办公软件的支持也经历了漫长的时间,亟需快速改善软件生态的解决方案[7-8,17].

上述生态困境源于新架构普遍面临的市场壁垒. ARM 架构在移动端和嵌入式市场占据重要地位,x86 架构则是个人计算和服务器环境的默认标准,这些成熟架构的市场惯性为 RISC-V 的进入设置了显著的障碍[18],导致开发者不愿意冒着巨大的风险为 RISC-V 开发应用软件. 而软件生态的缺失又抑制了用户采购和使用的意愿,这 2 个因素互相制约,甚至产生恶性循环,导致市场份额和软件生态共演化衰退,严重阻碍了 RISC-V 在 HPC 领域的发展.

动态二进制翻译(DBT)技术为破解这一困境提

供了新思路，可快速实现 x86/ARM 丰富软件生态向 RISC-V 平台的移植[19-25]. 为此，我们开发了进程级的二进制翻译器 RVBT，它能将 x86 应用程序翻译到 RISC-V 平台执行. RVBT 在翻译 SPEC CPU 2006 的整点测试集时，平均运行效率能达到本地执行的 39.04%，显著优于 QEMU[26] 的 18.84%. 但在浮点测试集上仅能达到本地执行的 4.82%，与 QEMU 处于相当水平，但与其自身在整点测试集上的性能表现相差了 8.10 倍. 经分析后发现，浮点测试集大量使用 SIMD（single instruction multiple data）指令，而 RVBT 翻译 SIMD 指令生成的代码运行效率很低，是主要的性能瓶颈. RVBT 使用 RISC-V 的 RVV 扩展翻译 SIMD 指令. RVV 和 SIMD 都具备数据级并行的能力，在预期中能实现高效的翻译. 对这一反直觉的现象进行深入分析后，我们观察到 3 个现象：

1）SIMD 扩展和 RVV 扩展在编程模型上具有显著差异，前者将数据位宽和操作的数据个数编码于操作码中，而后者通过向量类型寄存器（vector type register）vtype 寄存器以及掩码寄存器（mask register）动态配置. 频繁设置数据类型的开销非常大，是翻译执行的性能不及预期的一个关键原因.

2）x86 使用 SIMD 的指令和寄存器执行标量浮点运算，而 RISC-V 则使用独立的浮点扩展实现，这比使用 RVV 实现更加高效. RVBT 将 SIMD 指令都翻译成了 RVV 指令，使得 x86 上的标量浮点运算在 RISC-V 平台上是使用 RVV 指令来实现翻译的. 这是翻译执行的性能不及预期的另一个关键原因.

3）代码操作的数据类型具有局部性这一特性未被有效利用. 相临近的 SIMD 指令处理的数据类型往往是相同的，在翻译这些 SIMD 指令时使用的 vtype 和掩码设置是一样的. 利用这一特性可以为 RVBT 设计优化方案，提高翻译代码质量和运行效率.

基于上述 3 点观察，本文聚焦于将 x86 平台上的 SIMD 指令翻译到 RISC-V 平台执行的性能瓶颈问题，提出了 SetVType, SetMask 和 SD2Float 三项创新的优化方案. SetVType 和 SetMask 通过静态分析代码，找到冗余的 vtype 和掩码设置操作，并将其删除，以减少指令膨胀，提高翻译的本地码的质量. 以 SPEC CPU 2006 作为测试集，SetVType 对 csrr, vsetvl 和 vsetvli 指令的动态消除率分别达到了 100%, 100% 和 56.31%，在整点和浮点测试集上，对 vsetvli 指令的平均动态消除率分别达到了 48.88% 和 62.62%. SetMask 主要用于优化浮点运算，在浮点测试集上对掩码设置的平均动态消除率达到了 74.66%. SD2Float 使用混合翻译策略，将用于双精度标量浮点计算和打包浮点计算的 SIMD 指令分别翻译成 RISC-V 标量浮点计算指令和 RVV 指令，并通过分析 x86 汇编代码中 SIMD 寄存器的定值-引用关系，仅按需在浮点寄存器和向量寄存器之间同步数据，减少数据同步操作发生的频率. 在浮点测试集上，将平均静态和动态同步率分别削减到了 55.61% 和 67.35%.

实验表明，上述优化方案能显著提升 RVBT 的性能. SetVType 和 SetMask 减少了大量冗余的 vtype 和掩码设置. SD2Float 在使用更高效的指令进行翻译的同时，优化了大量数据同步操作. 同时实施上述优化后，以 SPEC CPU 2006 作为测试集，在全测试集、整点测试集和浮点测试集上，RVBT 翻译执行的平均效率可分别达到本地执行的 43.05%、47.39% 和 40.06%，相对优化前的平均加速比分别为 3.64、1.21 和 8.31. 本文的优化方案将浮点操作的翻译运行效率提升到了接近翻译整点操作的水平. 作为对比，QEMU 在 SPEC CPU 2006 全测试集、整点测试集和浮点测试集上的平均运行效率分别为本地的 8.64%、18.84% 和 4.81%. 优化后的 RVBT 在翻译整点和浮点操作上的性能表现均显著优于 QEMU.

综上，本文为 RISC-V 软件生态建设提供了高效的二进制兼容解决方案，对推动 RISC-V 开源架构在 HPC 领域的应用和发展具有重要的实用价值，具体贡献有 4 点：

1）提出了 SetVType 和 SetMask 优化方案，用于消除 SIMD 指令翻译成 RVV 指令时冗余的 vtype 和掩码设置操作，以减小因 SIMD 和 RVV 在编程模型上的巨大差异带来的性能开销，显著提升了本地码的质量和翻译执行的效率.

2）提出了 SD2Float 优化方案，使用 RISC-V 平台上效率更高的浮点指令来翻译 SIMD 指令中的标量双精度浮点操作，使用 RVV 翻译其他浮点操作，并通过静态分析实现了浮点寄存器和向量寄存器之间的按需数据同步. 这种混合的翻译方案提升了翻译浮点操作的效率.

3）以 x86-64 为源平台，以 RISC-V-64 为目标平台，实现了 3 项优化，解决了多个技术挑战，并通过实验验证了这些优化能显著提升 SIMD 指令的翻译运行效率，特别是浮点操作的翻译运行效率.

4）提出的 3 项优化在将 SIMD 指令翻译成 RISC-V 的 RVV 指令这一场景中是通用的. 不仅适用于 x86 到 RISC-V 平台的翻译，也适用于 ARM 等平台到 RISC-V 平台的翻译.

1 相关背景与动机

1.1 RVBT 系统简介

RVBT 系统是我们设计实现的一款面向 RISC-V 平台的动态二进制翻译系统，可将 x86-64 Linux 平台上的应用程序翻译到 RISC-V-64 Linux 平台上执行. 所有的动态二进制翻译系统在运行时都是在执行反汇编—翻译—优化—执行这一循环[27]，RVBT 也是如此，其系统架构如图 1 所示. RVBT 首先对 x86 二进制代码进行反汇编得到 IR1，随后对 IR1 进行分析，为后续的代码翻译和优化生成辅助信息. IR1 是 x86 汇编代码的结构化表示，对 IR1 中的代码进行逐条翻译后便得到 IR2. 开发动态二进制翻译系统的工作量非常大，为了增加软件的可维护性和鲁棒性，系统设计和实现通常遵循软件工程上的模块化设计理念，动态二进制翻译器对每条源平台的指令进行独立翻译，不考虑指令的上下文. 这是二进制翻译器普遍的设计选择，RVBT 也遵循这样的设计. IR2 是接近 RISC-V 汇编代码的中间表示，其中也包含我们自定义的指令以及还未进行寄存器分配时的临时寄存器占位符(图 1 代码中的 temp0). 对 IR2 进行优化和汇编后即得到可执行的 RISC-V 本地码.

通过上下文切换可实现动态二进制翻译系统的控制器和 x86 程序的目标平台二进制代码(也称本地码)之间的交替执行，如图 1 所示. 当控制器获得执行权时，x86 程序本地码的执行处于被挂起状态. 此时，二进制翻译系统能对执行过程中新发现的 x86 代码块进行翻译，也能重新调整此前已经翻译好的本地码. 在这一过程中，翻译器可以对 IR1 和 IR2 代码进行分析并实施代码优化. 分析可分为动态分析和静态分析. 其中，静态分析不依赖于 x86 程序执行，而动态分析则依赖动态收集的 x86 程序运行状态信息. 比如，在 IR1 或者 IR2 上判断前一条指令定值的寄存器是否被后一条指令使用，仅需要进行静态分析即可完成. 判断哪条执行路径是热代码，则需要进行动态分析.

代码分析和优化是在动态二进制翻译系统翻译执行 x86 程序的过程中进行的. 因此，系统翻译执行 x86 程序的时间包括动态二进制翻译系统初始化的时间、翻译和优化代码的时间，以及 x86 程序本地码的运行时间. 代码分析和优化能提升翻译出来的本地码的质量，进而提升性能. 假设完成代码分析和优化本身所需要的时长为 $t1$，Δt 为执行优化后的 x86 程序本地码节省的时间. 当 $t1<\Delta t$ 时，意味着代码分析和优化能带来性能提升. 反之，则说明代码分析和优化本身的开销过大，会拖累翻译执行的性能.

为了提升翻译执行的效率，RVBT 做了寄存器分

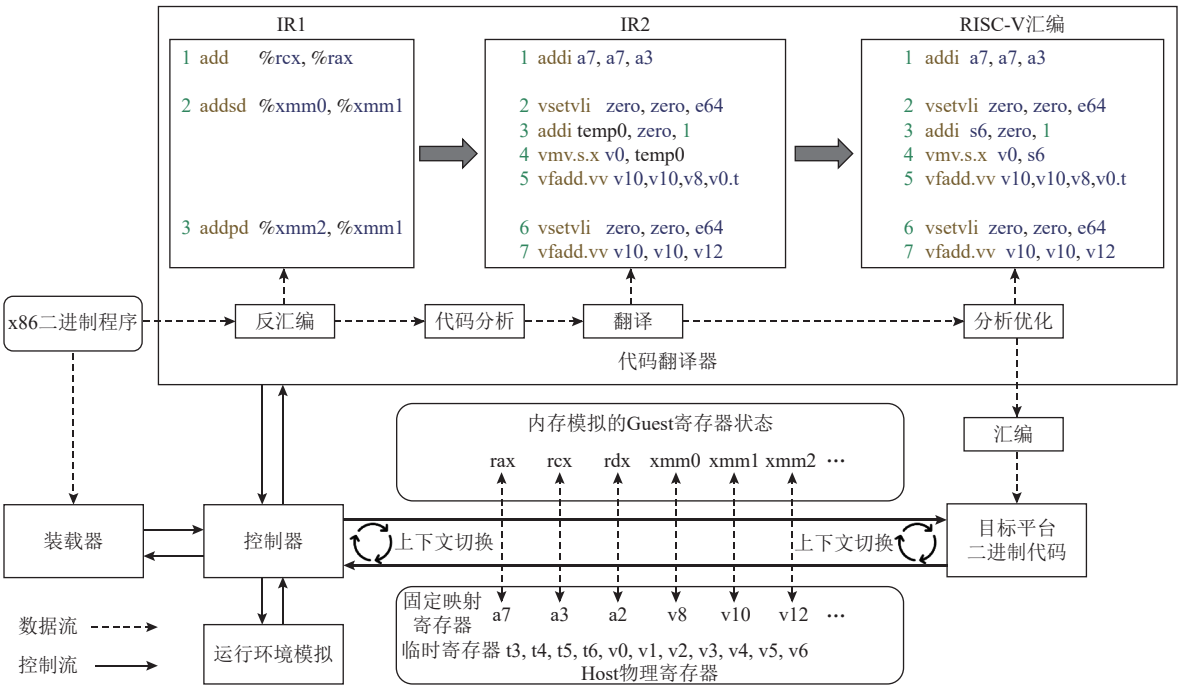


Fig. 1 Architecture of RVBT

图 1 RVBT 架构

配优化，将 x86 的通用寄存器固定映射到 RISC-V 的通用寄存器，将 SIMD 寄存器固定映射到 RVV 的向量寄存器. 比如，将 rax 和 rcx 寄存器分别映射到 a7 和 a3 寄存器上，将 xmm0 和 xmm1 寄存器分别映射到 v8 和 v10. x86 属于 CISC（complex instruction set computer）指令集，指令的语义相对复杂，往往需要使用多条 RISC-V 指令来翻译一条 x86 指令. 用于保存中间计算结果的 RISC-V 寄存器被称为临时寄存器. RVBT 对临时寄存器进行动态分配，必要时需要将其溢出到内存中暂存. 通用寄存器 t3~t6，向量寄存器 v0~v6 被用作临时寄存器. 同时，在 RVV 编程模型中，向量寄存器 v0 还有一个特殊用途，它被用作向量指令的掩码寄存器. 此外，RVBT 还实现了标志位运算优化、基本块链接优化等动态二进制翻译器常用的优化技术以提升性能.

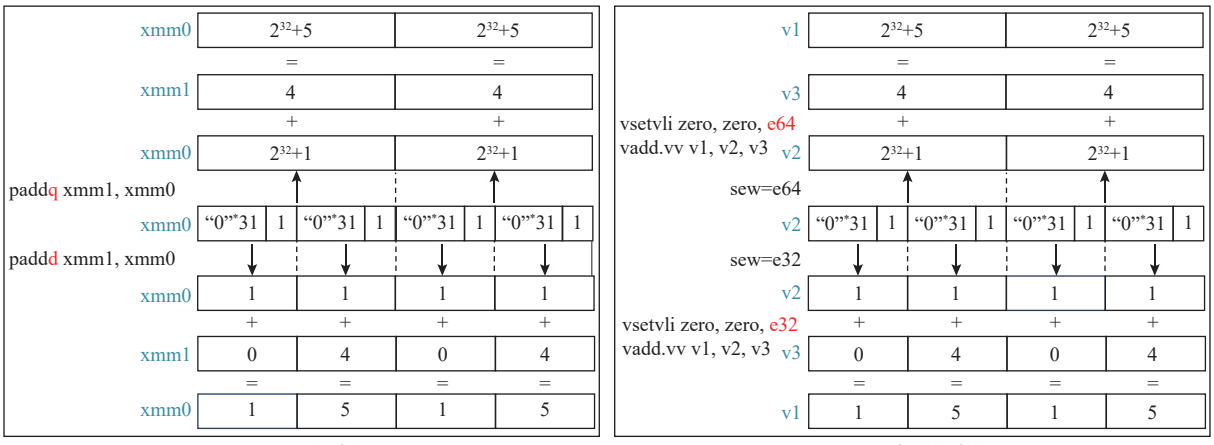
1.2 SIMD 翻译成 RVV 的方法

x86 的 SIMD 扩展和 RISC-V 的 RVV 扩展具有近似的功能，都具备使用 1 条指令处理多个数据元素的能力，可实现数据级并行，以加速程序的运行. 但二者的编程模型具有显著差异. SIMD 和 RVV 扩展的寄存器都可以同时容纳多个数据，比如 4 个 32 位整数或者 2 个 64 位整数，并通过一条指令操作寄存器中的数据. 不同的是，SIMD 指令将其操作的数据元素类型（即位宽）编码到了指令操作码中，而 RVV 指令如何操作寄存器中的数据则通过 vtype 来控制. 在 RVV 0.7.1 版本中，vtype 的第 2 到第 4 个位用于表示向量寄存器中单个数据元素的位宽，也被称为元素位宽 sew（standard element width）. 在本文中，设置 vtype 和设置 sew 表达相同的含义，为表述方便会交替使用. 当 sew 被设置为 e64 时，表示向量寄存器中每个数据元素的位宽是 64 位；当被设置为 e32 时，则表示位宽为 32 位. RVV 提供了 vsetvli，vsetvl 和 csrr 这 3 条指令用于操作 vtype 寄存器. 其中 vsetvli 指令使用立即数指定 sew，而 vsetvl 则使用寄存器指定 sew. csrr 指令的作用是将当前的 sew 状态读取到寄存器中，它与 vsetvl 配对使用可以实现 sew 的保存和恢复功能.

如图 2 所示，对于一次性计算 2 个 64 位整数相加和 4 个 32 位整数相加这 2 个加法功能，x86 分别使用 paddq 和 paddd 指令完成，而 RVV 都使用 vadd. vv 指令. RVV 通过使用 vsetvli 设置分别将 sew 设置为 e64 和 e32，让 vadd. vv 实现了上述 2 种向量加法. 体现 SIMD 和 RVV 上述不同设计理念的一个现象是，SIMD 指令扩展包含的指令数量远多于 RVV 扩展.

根据 SIMD 指令操作的数据类型的不同，我们将其分成 5 类，如表 1 所示. 其中，SS 和 SD 指令用于标量浮点操作，SS 指令只操作 xmm 寄存器的低 32 位，而 SD 指令则只操作 xmm 寄存器的低 64 位. RVV 通过为指令设置掩码来达到只操作向量寄存器中某一个分量元素的目的. 向量寄存器 v0 被 RVV 用作掩码寄存器，它的位宽和其它向量寄存器是一样的. 以图 1 中 IR2 的代码为例，其中第 3 行和第 4 行将 v0 的低 64 位设置为 1，高 64 位设置为 0，在 sew 为 e64 时，这样的设置让第 5 行的 vfadd. vv 仅操作 v10 和 v8 的第一个分量元素，即保存于低 64 位的数据元素. 如果将 v0 的低 64 位设置为 0，高 64 位设置为 1，则 vfadd. vv 就只操作 v10 和 v8 的第 2 个分量元素，即高 64 位. 除了作为掩码寄存器，v0 也能像其它向量寄存器一样用于各种算术运算中.

考虑到 RVV 指令的功能与 SIMD 指令有相似之



(a) x86 SSE指令扩展 (b) RVV 向量指令扩展

Fig. 2 Difference between programming models of SIMD and RVV

图 2 SIMD 和 RVV 编程模型的差异

Table 1 Classification of SIMD Instructions

表 1 SIMD 指令的分类

指令类型	操作数类型	操作的元素个数	指令示例
SS	标量单精度浮点	1	addss
SD	标量双精度浮点	1	addsd
PS	打包单精度浮点	4	addps
PD	打包双精度浮点	2	addpd
PI	打包整点指令	多个	paddd

处, RVBT 将 SIMD 指令翻译成 RVV 指令. 翻译过程可概括为 3 步: 1)设置 sew; 2)若是翻译 SD 或 SS 指令, 则设置掩码; 3)选择对应语义的 RVV 指令翻译 SIMD 指令的语义. 如图 1 所示, 翻译器独立地将每条 IR1 指令生成对应的 IR2 指令序列, IR2 中的第 2~5 条指令翻译 IR1 的第 2 条指令, IR2 中的第 6~7 条指令翻译 IR1 的第 3 条指令.

翻译表 1 中的 5 类 SIMD 指令都需要设置 sew, 翻译 SS, SD 指令和部分 PD 及 PS 指令时还需要设置掩码. 显然, 当 SIMD 指令操作的元素位宽相同时, sew 的设置就是相同的, 比如翻译 SD, PD 类型和操作 64 位整型的 PI 指令时, 使用的 sew 都是 e64. 当指令只操作向量寄存器的同一个分量元素时, 比如都只操作低 32 位, 那么翻译时设置的掩码是相同的. 如图 1 所示, IR2 指令序列中第 2 条和第 6 条指令都将 sew 设置为 e64, 结合指令上下文分析可知, 第 6 条指令是对 sew 的冗余设置, 可以删去. 设置掩码的操作也存在类似的情况. 设置 sew 和掩码开销都很大. 执行 csrr, vsetvl 和 vsetvli 指令分别需要 16, 12 和 2 个时钟周期. RISC-V 的 I 型指令和 U 型指令能分别编码最多 12 位和 20 位的立即数. 受此影响, 设置不同的掩码值, 可能需要不同的指令序列, 时间开销也不同, 为 30~40 个时钟周期. 远远大于单独执行 1 条向量加法指令 vfadd. vv 所需的 0.58 个周期.

为了提升翻译执行的效率, 本文提出了 SetVType 和 SetMask 优化, 充分利用数据类型具有局部性这一程序特征, 分别消除冗余的 sew 和掩码设置操作, 精简代码, 提升本地码的质量.

1.3 标量浮点操作的翻译方法

x86 平台可使用浮点栈和 SSE 指令实现标量浮点操作. 编译器在生成面向现代 CPU 的代码时, 普遍选择使用 SSE 指令(即表 1 中的 SS 和 SD 指令)以获得更好的性能. 比如使用 addsd 指令实现 2 个双精度浮点数相加. 这样一来, x86 平台上标量浮点和打包浮点操作就都使用 SIMD 扩展的指令和寄存器.

在 RISC-V 平台上, 标量浮点操作的实现与 x86 平台有明显区别. RISC-V 有专门的浮点扩展, 并专门设计了 32 个独立的浮点寄存器. 经测试后发现, 在 RISC-V 平台上使用浮点扩展指令实现标量浮点计算比使用 RVV 指令更加高效. 因此, RISC-V 平台上的 GCC 使用浮点扩展指令来完成标量浮点计算.

RVBT 将 SIMD 指令都翻译成了 RVV 指令, 使得 x86 上通过 SSE 指令实现的标量浮点运算被翻译成了 RISC-V 的 RVV 指令. 通过实测发现, 如果将 x86 上的标量浮点操作翻译成 RISC-V 的浮点指令, 能获得更高的翻译运行效率, 如表 2 所示. 但如果将 SIMD 中的打包浮点操作也翻译成 RISC-V 的浮点指令, 则翻译运行效率会下降.

Table 2 Number of Clock Cycles Required for Different Translation Schemes of Scalar Floating-Point

表 2 标量浮点的不同翻译方案所需的时钟周期数

指令	翻向量	翻浮点	翻浮点&同步
mulsd xmm1, xmm2	44.17	0.58	47.74
sqrtsd xmm1, xmm2	61.29	19.08	64.73
subsd xmm1, xmm2	44.15	0.58	46.82
addsd xmm1, xmm2	44.2	0.58	46.78
divsd xmm1, xmm2	62.15	19.09	64.75
maxsd xmm1, xmm2	45.15	0.57	46.73
minsd xmm1, xmm2	45.15	0.59	46.71

本文提出 SD2Float 优化, 将 SD 类型的标量浮点操作翻译成 RISC-V 的浮点指令, 与此同时, 其他 SIMD 指令依然翻译成 RVV 指令. 这种混合的翻译方式能充分利用 RVV 数据级并行的特性和浮点扩展高效的标量浮点计算能力. RISC-V 的浮点寄存器和向量寄存器是 2 组独立的寄存器, 而在 x86 平台上, 标量浮点和打包浮点运算都使用 xmm 寄存器. 这就意味着需要在 RVV 的向量寄存器和浮点扩展的浮点寄存器之间同步数据. 如表 2 所示, 加上数据同步操作之后, 就失去了把 SD 类型的指令翻译成 RISC-V 的浮点指令的性能优势. 我们观察到, 数据类型具有局部性, 在一个代码块中, 程序往往操作相同的数据类型. 充分利用这一特性就不需要为每一条指令都发射同步指令.

为了实现按需同步, SD2Float 优化在 IR1 上通过静态分析确定 SIMD 指令之间的定值-引用关系, 在翻译过程中仅在需要进行数据同步时才发射数据同步指令.

1.4 研究动机

在 x86 平台上，SIMD 指令被真实世界中的程序广泛使用，特别是在面向 HPC 场景的应用软件中，比如深度学习算法[28-29]、大模型推理算法[30-33]、多媒体应用等. OpenCV[34] 和 FFmpeg[35] 等库使用 SIMD 指令来优化其核心算法，以获得更好的性能表现. 据统计，多媒体应用中 25% 的指令是 SIMD 指令[36]. Gedit, Google-V8, VisualStudio 等 9 款 x86 典型应用中，SIMD 指令的平均占比为 3.1%[22]. Ubuntu Linux 16.04 APT 仓库中，含有二进制程序的包里，99% 的包都使用了 SIMD 扩展指令. SSE 是使用最广泛的 SIMD 扩展指令，在 HPC 等场景中大量存在. 大部分较为常用的 SSE 指令操作的都是浮点类型的数据[37]. 在动态二进制翻译中，浮点操作的翻译普遍性能较低，这限制了该技术的应用场景，是一个核心痛点问题[38]. 提升 SIMD 指令的翻译运行效率，能大幅提升翻译浮点操作的性能，扩展翻译器的适用场景，对提升二进制翻译器的实用性非常有价值.

动态二进制翻译技术的演进可追溯至 20 世纪 80 年代. 经过几十年的发展，已经有很多翻译器被研发出来，但以 RISC-V 为目标架构的并不多见. 以 QEMU、Box86/64[39] 和 DBT-FEMU[40] 为代表的几款翻译器能将 x86 应用翻译到 RISC-V 上运行，但没有针对 SIMD 指令进行优化. DBT-FEMU 聚焦在优化整点指令的翻译上，QEMU 和 Box86/64 使用标量指令模拟向量语义，来完成对 SIMD 指令的翻译. 过往对 SIMD 指令的翻译优化研究主要聚焦在 x86 和 ARM 这 2 个平台上. 研究方向主要是如何高效利用目标平台上位宽更长的 SIMD 寄存器资源、如何更好地进行向量化等.

x86 上的 SIMD 和 RISC-V 平台上的 RVV 因编程模型的巨大差异导致翻译效率不高这一问题还没有被关注和研究. SIMD 是在 HPC 场景中被广泛使用的性能加速基础设施. RISC-V 作为高性能计算领域的新入局者面临软件生态不完善的困境. 研发一款能够高效翻译 SIMD 指令至 RISC-V 平台的动态二进制翻译器，对加速 RISC-V 软件生态的建设与发展具有显著实用价值.

2 设计优化方案面临的挑战与解决思路

静态消除冗余的 sew 设置操作的挑战在于消除 vsetvl 指令对静态数据流分析的阻碍，持续跟踪程序对 sew 的设置. RVV 的 vsetvli 和 vsetvl 指令都能修改 sew 的状态. 区别在于 vsetvli 对 sew 的设置编码在立即数中，能直接从代码上读出它会将 sew 设置为何值. 而 vsetvl 指令的操作数是寄存器，完整的汇编指令格式是 vsetvl rd, rs1, rs2. 该指令执行后，其第 2 个源操作数 rs2 中保存的值将被设置为新的 sew. 显然，通过静态分析该指令无法直接读出寄存器 rs2 的值. 因此，vsetvl 指令会阻碍静态分析追踪代码对 sew 的设置过程.

一个直接的方案是进行逆向的数据流分析，回溯该寄存器的定值过程，但这样做开销较大. 本文通过分析 vsetvl 指令的使用场景，以先验知识确定其寄存器操作数 rs2 的数值来源于执行路径上前一条 vsevli 指令，进而高效地将其删除或替换.

静态消除冗余的掩码设置操作的挑战在于定位掩码操作和捕获掩码状态的变化. 静态确定每次掩码设置将掩码寄存器 v0 修改为何值，是通过静态分析消除冗余的掩码设置的前提. 设置掩码是将立即数传送到向量寄存器 v0 的过程. 向量寄存器的位宽远超单条 RISC-V 指令能编码的最大立即数的位宽. 因此设置掩码通常需要使用多条 RISC-V 指令来完成. 设置不同的掩码值，用到的指令序列是不一样的. 同样的掩码值，也可以通过不同的指令序列来设置. 另外，掩码寄存器 v0 和其他向量寄存器一样，还被用于各类算术运算. 这意味着要通过静态分析来确定对掩码的设置，需要在众多对 v0 寄存器进行定值的代码片段中识别出哪些是掩码设置、哪些是普通的算术运算. 这种方案既难以保证精确识别，容易导致正确性问题，还需要进行大量的定值引用分析，开销很大. 如果进行非常保守的分析，则仅能消除少量的掩码设置，对性能提升的帮助很小.

SetMask 优化将掩码设置分为伪指令占位、冗余设置删除和掩码设置指令序列发射 3 个步骤来解决上述问题. 我们在 IR2 上设计了一条伪指令 set_mask，该指令包含 2 个立即数操作数，分别表示掩码的高 64 位和低 64 位. 翻译 IR1 时，首先使用该伪指令在 IR2 的指令序列中标记何处需要进行掩码设置，省去了在优化阶段识别掩码设置指令序列的步骤，同时确定掩码值也不再需要对 v0 进行定值引用分析. 然后将冗余的 set_mask 伪指令删除. 最后则是将剩余的 set_mask 伪指令发射成对应的 RISC-V 指令序列.

SIMD 指令的混合翻译方案面临的挑战在于降低浮点寄存器和向量寄存器之间的数据同步开销. x86 上标量浮点和打包浮点运算都使用 xmm 寄存器，而 RISC-V 的浮点寄存器和向量寄存器是 2 套独立的寄存器. 为了提升性能，RVBT 将 x86 源平台的寄存

器和 RISC-V 目标平台上的寄存器进行了映射. 在混合翻译方案中，整个 xmm 将被映射到 RVV 的向量寄存器，同时，它的低 64 位将被映射到浮点寄存器. 也就是说，xmm 寄存器的低 64 位被同时映射到了 2 个 RISC-V 寄存器上. 当翻译成浮点和翻译成向量的 SIMD 指令交替出现时，就可能要在浮点和向量寄存器之间进行数据同步才能保证正确性.

以图 1 中 IR1 的代码为例，第 2 条指令执行后，xmm1 的低 64 位被定值了，第 3 条指令同时使用了 xmm1 寄存器的高 64 位和低 64 位. 在混合翻译方案中，第 2 条指令被翻译成 RISC-V 的浮点指令. 假设 xmm1 的低 64 位映射到 RISC-V 的浮点寄存器 fa1. 当翻译的本地码执行后，计算结果就保存在了浮点寄存器 fa1 中. 但 xmm1 的低 64 位还被映射到了向量寄存器 v10 的低 64 位中，而浮点指令不会更新 v10. 此时，浮点寄存器 fa1 和向量寄存器 v10 的高 64 位组合起来表达的才是 x86 程序的正确执行状态. 后续第 3 条指令被翻译成 RVV 向量指令，指令执行时将使用向量寄存器 v10 的高 64 位和低 64 位. 这将引发错误，因为上一条指令的执行结果没有被更新到 v10 的低 64 位中. 需要在执行第 3 条指令的翻译前，先把 fa1 的值同步到 v10 的低 64 位，才能维护 x86 指令的定值-引用关系. 如果混合翻译里每条指令执行后都进行浮点和向量寄存器之间的数据同步，就无法获得性能上的提升.

SD2Float 优化利用程序中数据类型的局部性特征，通过定值引用分析，只进行必要的数据同步，在充分发挥出 RISC-V 浮点扩展和 RVV 向量扩展各自性能优势的同时，降低数据同步带来的性能开销，提升翻译执行 x86 浮点操作的效率.

以较低的性能开销实现静态代码分析和优化是本文 3 项优化方案面临的共同挑战. 影响动态二进制翻译系统运行效率的一大因素是代码翻译和优化上的开销[40-41]. 因此，设计优化机制时必须非常谨慎，实施优化操作本身带来的开销必须得到控制，否则优化就难以带来性能提升. 一方面，本文通过控制分析范围减小静态分析的开销，以翻译块(translation block，TB)为 SetVType 和 SetMask 优化的分析单元和以函数内的执行路径为 SD2Float 优化的分析单元. 这 3 项优化都利用了程序数据类型具有局部性这一特征，持续扩大分析范围带来的性能增益是递减的，同时会增大静态分析的开销. 另一方面，本文通过翻译器运行机制上的先验知识来减少对静态数据流分析的依赖，比如根据 vsetvl 指令的使用场景，设计低开销的指令消除和转换算法，以及用自定义的伪指令标记掩码设置操作，从而避免进行代码模式分析和匹配.

3 优化方案的设计与实现

3.1 SetVType 优化方案的设计与实现

SetVType 优化在 IR2 上以 TB 为单元实施优化. 通过对 IR2 进行静态代码分析，完成 3 个关键操作：1)消除 csrr 和 vsetvl 指令；2)如果一条 vsetvli 指令直接支配的所有指令都是不受 sew 影响的指令，则删除该 vsetvli 指令；3)删除冗余设置 sew 的 vsetvli 指令. 指令 *A* 直接支配(immediately dominant)指令 *B*，是指指令 *A* 支配指令 *B*，且 *A* 与 *B* 之间的控制流上不存在其他的 *A* 指令. 将静态代码分析控制在单个 TB 中，能降低分析的复杂度，尽可能减小静态分析带来的性能开销. IR1 上 TB 的控制流结构简单，具有单入单出的特点. 将 IR1 的一个 TB 翻译成 IR2 后，控制流中可能存在分支跳转，比如 IR2 使用分支跳转指令来翻译 IR1 中的条件设置(conditional set)指令. 但在 IR2 中，TB 的控制流依然是相对简单的，相当比例的 TB 依然是没有分支的.

为了尽可能多地消除 vsetvli 指令，SetVType 优化首先对阻碍分析的 csrr 和 vsetvl 指令进行删除或替换. RVBT 使用 csrr 和 vsetvl 指令对的目的只有 1 个，先用 csrr 指令保存 sew 的状态，然后使用 vsetvli 指令修改 sew，最后使用 vsetvl 指令恢复先前保存的 sew. 出于模块化设计，RVBT 将设置掩码以及从内存中加载数据等公共操作封装成 API 函数，供所有编写翻译函数的开发人员使用. 在软件工程和软件设计角度，这些 API 函数不应该感知调用点的上下文. 当它要进行向量操作时，需要完成如下操作序列：使用 csrr 指令保存调用者的 sew → 使用 vsetvli 指令设置新 sew → 完成向量操作 → 恢复调用者的 sew. 这样就能让 API 的使用者不必关心调用 API 后向量执行环境是否发生了改变. 在 RVBT 发射的本地码中，csrr 和 vsetvl 指令总是成对出现的.

RVBT 在翻译每一条 SIMD 指令时，都使用 vsetvli 指令为其设置对应的 sew. 这就意味着每一对 csrr-vsetvl 都必然和一条 vsetvli 指令关联. 这条 vsetvli 指令可能出现在 csrr-vsetvl 指令对之前，也可能在它之后，这取决于翻译函数的开发人员是如何排布指令顺序的. 我们设计了算法 1 用于消除 csrr 和 vsetvl 指令. 当一个 TB 中存在不被 vsetvli 指令支配的 csrr-vsetvl 指令对时，我们直接将其删除. 对于该 TB 来说，

这是一次无效的 sew 保存和恢复操作. 如果一条 vsetvli 直接支配了一个 csrr-vsetvl 指令对, 则删除 csrr 指令, 并使用该 vsetvli 指令替换 vsetvl 指令. vsetvli 直接支配一个 csrr-vsetvl 指令对, 意味着该 vsetvli 指令和该 csrr-vsetvl 指令对之间不存在其他 vsetvli 指令. 替换 vsetvl 的目的有 2 个. 一是为后续消除 vsetvli 指令做准备. 二是执行 vsetvli 的开销要远小于 vsetvl. 执行 vsetvli 指令需要 2 个时钟周期, 而 vsetvl 则需要 12 个时钟周期, 相差 5 倍. 即使后续优化无法删除用于替换 vsetvl 的 vsetvli 指令, 也能获得性能上的提升.

算法 1. csrr-vsetvl 指令对消除算法.

输入: 待优化的 TB;

输出: 无.

① procedure *convert_vsetvl* (*TB*&*tb*);
② foreach *csrr-vsetvl* pair in *tb* do
③ *vli*←get the immediate dominant vsetvli of the pair;
④ delete *csrr*;
⑤ if *vli* is not null then
⑥ replace *vsetvl* with *vli*;
⑦ else
⑧ delete *vsetvl*;
⑨ end if
⑩ end for
⑪ end procedure

RVV 中每条指令的执行都依赖于 sew. 但有的指令在不同的 sew 下执行结果是一样的, 表 3 列出了 3 类执行结果不受 sew 影响的指令. 这意味着如果被一条 vsetvli 直接支配的所有指令都是不受 sew 影响的, 该 vsetvli 指令可以被删除. 我们使用算法 2 来实现这一操作.

算法 2. 无效的 sew 设置删除算法.

输入: 待优化的 TB;

输出: 无.

① procedure *del_useless_vli* (*TB*&*tb*);
② foreach vsetvli instruction *vli* in *tb* do
③ *L*←get *inst* list *L* immediately dominanted by *vli*;
④ if all *inst* in *L* are sew unaffected isntrctions then
⑤ delete *vli*;
⑥ end if
⑦ end for
⑧ end procedure

Table 3 Instruction Categories Not Unaffected by sew

表 3 不受 sew 影响的指令类

类型	指令功能	指令示例
Ⅰ	仅操作标量	add a0, a1, a2
Ⅱ	向量寄存器间数据移动	vmv.vv v2, v3, v4
Ⅲ	向量位操作	vxor.vv v2, v3, v4

通过上述 2 项操作后, IR2 上与 sew 设置相关的指令就只剩下 vsetvli 了. 此时, 通过分析该指令的操作数就能获得每一处对 sew 的设置将其调整到的状态. SetVType 使用前向数据流分析来确定哪些 vsetvli 指令可以被删除, 数据流方程见式(1)~(4), 在 SetMask 优化中, 也将使用这套数据流方程进行静态分析. 在 SetVType 优化中, 使用这组数据流方程分析 TB 内的所有 vsetvli 指令, 在 SetMask 优化中则分析 set_mask 伪指令. 处理每条 vsetvli 指令前, 要判断在该指令之前, 对 sew 的既往设置是否依然可用. 若既往的 sew 设置是不可用的, 则说明该 vsetvli 指令必须被保留. 若既往的设置依然可用, 则需要比较该 vsetvli 指令设置的 sew 与既往设置的 sew 是否一致. 如果是一致的, 则说明该 vsetvli 指令的设置是冗余的, 可以被删除, 否则应该被保留.

$$State_{\text{in}}(entry) = \phi, \tag{1}$$

$$State_{\text{in}}(S) = \bigcap_{S' \in pred(S)} State_{\text{out}}(S'), \tag{2}$$

$$State_{\text{out}}(S) = Gen(S) \cup (State_{\text{in}}(S) - Kill(S)), \tag{3}$$

$$State_{\text{in}}(S) \stackrel{?}{=} State_{\text{out}}(S). \tag{4}$$

删除冗余 vsetvli 指令的实现如算法 3 所示. RVBT 的翻译器在生成 IR2 时, 如果代码中的控制流含有分支, 则一定会生成 label 来标记分支跳转指令的目标. 若在遍历 TB 的指令时遇到了 label, 说明下一条指令是跳转指令的跳转目标. 此时, 根据式(2)获取各个前驱中直接支配当前 label 的 vsetvli 指令对 sew 的设置情况. 当 label 处没有可用的 sew 设置时, 通过算法 3 的行⑮标记无可用的 sew 状态. 每当分析到 vsetvli 指令时, 都根据式(4)判断该指令能否被消除, 对应算法 3 的行⑤~⑦伪代码.

算法 3. 冗余 sew 设置消除算法.

输入: 待优化的 TB;

输出: 无.

① procedure *del_repetitive_vli* (*TB*&*tb*);
② *pre_sew* ← −1;
③ foreach *inst* in *tb* do

④ if *inst* is vsetvli (*sew*) then
⑤ if *sew* == *pre_sew* then
⑥ delete *inst*;
⑦ else
⑧ *pre_sew* ← *sew*;
⑨ end if
⑩ end if
⑪ if *inst* is label then
⑫ if *inst*'s immediate pred sews are the same then
⑬ *pre_sew* ← pred *sew*;
⑭ else
⑮ *pre_sew* ← −1;
⑯ end if
⑰ end if
⑱ end for
⑲ end procedure

经过 SetVType 的 3 步优化操作, IR2 上不再含有开销很大的 csrr 和 vsetvl 指令, 同时大量的 vsetvli 指令也被消除.

3.2 SetMask 优化方案的设计与实现

SetMask 优化将翻译过程中的掩码设置分成伪指令占位、冗余设置删除和掩码设置指令序列发射 3 个阶段实现. 在基线版 RVBT 的设计中, 翻译函数在将 IR1 翻译成 IR2 的过程中, 设置掩码的操作会被立即发射成 IR2 的指令序列. 优化的第 1 阶段是 SetMask 优化提供了用于占位的伪指令 set_mask. 在翻译 IR1 时, 如果进行掩码值已知的掩码设置, 则通过发射一条 set_mask (hi, lo) 伪指令进行占位, 其中 hi 和 lo 是 2 个立即数, 分别表示掩码的高 64 位和低 64 位. 如果一个 TB 中发射了 set_mask 伪指令, 那么该 TB 在动态分配临时寄存器时就不会考虑 v0, 避免给后续的静态分析带来干扰, 同时也保证删除冗余的掩码设置后, 程序不会因为 v0 寄存器被其他操作再次定值而运行出错.

在翻译 IR1 的过程中有 2 类掩码设置. 一类是静态可确定掩码值的设置, 比如翻译 addsd 指令时需要将掩码高 64 位设置为 0, 低 64 位设置为 1. 另一类是静态不确定掩码值的设置, 比如在翻译 x86 指令 vblendvpd 时就会遇到这类情况. 该指令的语义是根据第 1 个操作数各个数据元素的最高位是否为 1, 来决定是从第 2 个还是从第 3 个操作数中选择对应的数据元素存入目的操作数中. 指令 vblendvpd 的翻译如图 3 所示, 第 4 行代码是设置掩码寄存器 v0, 但掩码的值是通过将 v8 中的元素右移 63 位得到, 静态是未知的. 这种情况不能发射 set_mask 伪指令进行占位, 必须即刻发射设置对应掩码的 RISC-V 指令. 在第 2 阶段的静态分析中, 我们将对这种情况进行保守处理, 认为掩码设置的数据流无法被继续跟踪下去.

```
x86指令:
1  vblendvpd %xmm0, %xmm1, %xmm2, %xmm3

#  翻译x86指令得到的RISC-V本地码:
2  vsetvli x0, x0, e64
3  li s6, 63
4  vsrl.vx v0, v8, s6
5  vermge.vvm v14, v12, v10, v0
```

Fig. 3 Mask setting with unknown mask value in static
图 3 掩码值静态未知的掩码设置

优化的第 2 阶段是借助静态数据流分析来删除 IR2 上冗余的掩码设置. 为实现轻量级的静态分析, 降低分析开销, 我们以 TB 为单元进行分析并删除冗余的掩码设置. 数据流分析过程中使用的数据流方程与 SetVType 优化类似, 见式(1)~(4). 不同之处在于分析的指令由 vsetvli 变为 set_mask 伪指令. 算法 4 实现了第 2 阶段的数据流分析和冗余设置删除过程, 其中行④~⑦伪代码用于处理掩码值未知的掩码设置.

算法 4. 冗余掩码设置消除算法.

输入: 待优化的 TB;

输出: 无.

① procedure *del_repetitive_mask* (*TB*&*tb*);
② *pre_mask* ← (−1, −1);
③ foreach *inst* in *tb* do
④ if register v0 is defined by *inst* then
⑤ *pre_mask* ← (−1, −1);
⑥ continue;
⑦ end if
⑧ if *inst* is a set_mask (*mask*) pseudo-inst then
⑨ if *mask* == *pre_mask* then
⑩ delete *inst*;
⑪ else
⑫ *pre_mask* ← *mask*;
⑬ *emit* (*mask*);
⑭ end if
⑮ end if
⑯ if *inst* is a label then
⑰ if *inst*'s immediate pred masks are the same then

⑱ *pre_mask* ← *inst*'s pred *mask*;
⑲ else
⑳ *pre_mask* ← (−1, −1);
㉑ end if
㉒ end if
㉓ end for
㉔ end procedure

优化的第 3 阶段实现在 *emit* (*mask*) 函数中，该函数在算法 4 的行⑬被调用，用于为无法删除的掩码设置伪指令 set_mask 发射 RISC-V 指令序列. 受 RISC-V 指令能直接编码的立即数的最大位宽限制，设置掩码通常需要使用一组指令来完成. 这个设置可以概括为一个通用的指令序列模版，即加载立即数设置掩码值的一部分比特位，进行位移操作，继续加载立即数，并与此前设置好的比特位结合，反复上述操作数次，直到掩码值的所有比特位都被设置好. 我们对 5 个常用的掩码值的设置进行了定制优化，以能获得更好的性能，如图 4 所示.

mask	*lo*=1 *hi*=0	*lo*=0 *hi*=1	*lo*=0x0101010101010101 *hi*=0	*lo*=1 *hi*=1	*lo*=0x100000000 *hi*=0
定制指令序列	csrr t4, vtype vsetvli x0, x0, 0b01100 li t3, 1 vmv.s.x v0, t3 vsetvl zero, zero, t4	csrr t4, vtype vsetvli x0, x0, 0b01100 vmv.v.i v1, 1 vslide1up.vx v0, v1, zero vsetvl x0, x0, t4	csrr t4, vtype vsetvli x0, x0, 0b00000 vmv.v.i v1, 1 vslidedown.vi v0, v1, 8 vsetvl x0, x0, t4	csrr t4, vtype vsetvli x0, x0, 0b01100 vmv.v.i v0, 1 vsetvl x0, x0, t4	csrr t4, vtype vsetvli x0, x0, 0b01100 li t3, 1 slli t3, t3, 0x20 vmv.s.x v0, t3 vsetvl x0, x0, t4
周期数	31.2	34	31	30	31.6

通用模版指令序列	csrr t5, vtype
	sext.w t4, zero
	addiw t3, zero, 1
	vsetvli x0, x0, 0b01100
	vmv.s.x v1, t4
	vslide1up.vx v0, v1, t3
	vsetvl x0, x0, t5
		csrr t5, vtype
		addiw t4, zero, 1
		sext.w t3, zero
		vsetvli x0, x0, 0b01100
		vmv.s.x v1, t4
		vslide1up.vx v0, v1, t3
		vsetvl x0, x0, t5
			csrr t5, vtype
			sext.w t4, zero
			lui t3, 0x1010
			addiw t3, t3, 257
			slli t3, t3, 0x10
			addi t3, t3, 257
			slli t3, t3, 0x10
			addi t3, t3, 257
			vsetvli x0, x0, 0b01100
			vmv.s.x v1, t4
			vslide1up.vx v0, v1, t3
			vsetvl x0, x0, t5
				csrr t5, vtype
				addiw t4, zero, 1
				addiw t3, zero, 1
				vsetvli x0, x0, 0b01100
				vmv.s.x v1, t4
				vslide1up.vx v0, v1, t3
				vsetvl x0, x0, t5
					csrr t5, vtype
					sext.w t4, zero
					addiw t3, zero, 1
					slli t3, t3, 0x20
					vsetvli x0, x0, 0b01100
					vmv.s.x v1, t4
					vslide1up.vx v0, v1, t3
					vsetvl x0, x0, t5

周期数	35	35	39.7	35.6	36

Fig. 4 Performance comparison between custom instruction sequence and general template instruction sequence for mask settings

图 4 掩码设置的定制指令序列和通用模版指令序列性能对比

SetMask 优化能删除大量冗余的掩码设置，提升本地码质量，从而提升翻译执行的效率.

3.3 SD2Float 优化方案的设计与实现

面向现代 CPU 的 x86 代码中，标量浮点和打包浮点计算都是使用 SIMD 指令实现的. SD2Float 优化旨在将 x86 的双精度标量浮点操作(即表 1 中 SD 类型的指令)翻译为 RISC-V 的浮点操作，将其他浮点操作翻译成 RVV 向量操作. 这种混合的翻译方法能充分利用 RISC-V 浮点扩展和 RVV 扩展各自的性能优势. 在消除不必要的数据同步后能提升翻译执行的效率.

SD2Float 优化提供的混合翻译机制由分析阶段和翻译阶段组成. 在分析阶段，通过在 IR1 上分析浮点操作相关的 SIMD 指令对所有 xmm 寄存器的定值-引用关系，为这些指令生成数据同步的信息. 在随后的翻译阶段，翻译器的翻译函数根据分析阶段提供的信息按需发射数据同步的指令，在浮点寄存器和向量寄存器之间进行数据同步.

我们将浮点操作相关的 SIMD 指令分为 2 类：一类会被翻译为 RISC-V 浮点指令，称为 sfp 指令；另一类会被翻译为 RVV 向量指令，称为 svec 指令. 在图 5 所示的 IR1 代码中，addsd 指令是 sfp 指令，addpd 则是 svec 指令. 触发 RISC-V 浮点和向量寄存器之间发生数据同步的条件是，一类指令对 xmm 寄存器低 64 位的定值被另一类指令引用了. 数据同步的方向取决于定值操作是由哪一类指令完成的. 若 sfp 指令的定值被 svec 指令引用，则数据从浮点寄存器同步到向量寄存器的低 64 位，即 sync_fp_to_vec 操作. 反之，则从向量寄存器的低 64 位同步到浮点寄存器，即 sync_vec_to_fp 操作.

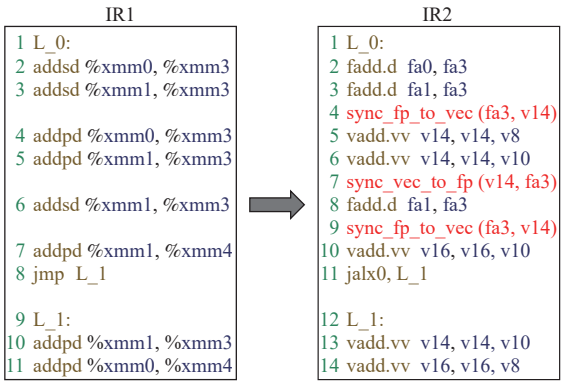


Fig. 5 Example for sfp, svec mixed sequence translation

图 5 sfp、svec 混合序列翻译示例

根据指令类型和语义，把对 xmm 的定值行为分为 3 类：高 64 位向量定值(vec_def_hi)、低 64 位向量定值(vec_def_lo)和低 64 位浮点定值(fp_def_lo). 对 xmm 的引用行为也可以分为 3 类：高 64 位向量引用(vec_use_hi)、低 64 位向量引用(vec_use_lo)和低 64 位浮点引用(fp_use_lo). 执行 sfp 指令可能发生 fp_def_lo 和 fp_use_lo 操作，而执行 svec 指令则可能发生 vec_def_hi、vec_def_lo、vec_use_hi 和 vec_use_lo 操作. 当 vec_def_lo 操作产生的定值被 fp_use_lo 操作引用时，将触发 sync_vec_to_fp 数据同步，当 fp_def_lo 操作产生的定值被 vec_use_lo 操作引用时，将触发 sync_fp_to_vec 数据同步.

以图 5 所示的 IR1 代码为例，第 2 行代码中 sfp 指令(addsd)定值了 xmm3 低 64 位，即发生了 fp_def_lo. 第 4 行代码中的 svec 指令(addpd)引用了 xmm3，即发生了 vec_use_lo 和 vec_use_hi. 但是翻译第 2 行代码时不需要发射数据同步指令，因为第 2 行 sfp 指令的定值被其后的第 3 行代码注销了. 而第 3 行代码对 xmm3 的 fp_def_lo 操作定值了 xmm3 的低 64 位，且在第 4 行被 svec 指令的 vec_use_lo 操作引用了. 所以，翻译第 3 行代码时需要发射 RISC-V 浮点寄存器向向量寄存器同步数据的指令，即 IR2 上的第 4 行伪代码 sync_fp_to_vec(fa3, v14). 同理，在翻译 IR1 的第 5 行代码时，则需要进行另一个方向的数据同步.

为了实现上述按需同步数据的方案，我们设计了算法 5，在 IR1 上对 16 个 xmm 寄存器发起后向数据流分析，该分析和活跃变量分析的过程类似. 分析过程中，算法 5 为 IR1 每一条 SIMD 指令记录下将来翻译它时是否需要发射数据同步指令以及数据同步的方向，对应行⑱~㉓. 算法将动态维护一张名为 xmm_recent_use 的表，该表被现实为 TB 类的成员变量，包含 16 行，每一行对应一个 xmm 寄存器. 它记录了算法分析到当前的程序点上时，在其后续控制流上各个 xmm 寄存器是否存在 vec_use_hi、vec_use_lo 和 fp_use_lo 操作. 算法每分析一条指令都对 xmm_recent_use 进行更新，更新时遵循式(5)所示的数据流转换方程，对应行㉔和行㉖.

算法 5. 按需同步数据的分析算法.

说明：TB. xmm_recent_use 的所有域在创建时全被初始化为 1，TB 的 is_analyzing 和 is_analyzed 域被初始化为 false，inst. sync_to_fp 和 inst. sync_to_vec 都被初始化为 0.

输入：待分析的 TB；

输出：*tb. xmm_recent_use*.

① procedure *sync_analysis* (*TB&tb*);
② if *tb. is_analyzing* then
③ return *tb. xmm_recent_use*
④ end if
⑤ if *tb. is_analyzed* then
⑥ return *tb. xmm_recent_use*;
⑦ end if
⑧ *tb. is_analyzing* ← true;
⑨ all *succ_use fileds* ← 0;
⑩ foreach *succ* in *tb*'s succ tbs do
⑪ *succ_use* ∪= *sync_analysis*(*succ*); /*recursive call*/
⑫ end for
⑬ *tb. xmm_recent_use* ← *succ_use*;
⑭ *inst* ← *tb. tail*;
⑮ while *inst* in *tb* do
⑯ foreach *opnd* in *inst*'s all xmm oprands do
⑰ if *opnd* is dest oprand then
⑱ if need sync to fp then
⑲ inst. *sync_to_fp* ← 1;
⑳ end if
㉑ if need sync to vec then
㉒ inst. *sync_to_vec* ← 1;
㉓ end if
㉔ kill *tb. xmm_recent_use*[*opnd*];
㉕ end if
㉖ update *tb. xmm_recent_use*[*opnd*];
㉗ end for
㉘ *inst* ← *inst. pre*;
㉙ end while
㉚ *tb. is_analyzing* ← false, *tb. is_analyzed* ← true;
㉛ return *tb. xmm_recent_use*;
㉜ end procedure

SD2Float 按需同步数据的分析将单次分析限定在函数内的可达路径范围内，跨 TB 的数据流分析遵循式(6)和式(7)所描述的数据流方程，实现如算法 5 行⑩~⑬伪代码所示. 这样的设计选择是为了让分析更加轻量化，同时，尽可能避免为了维护正确性进行保守处理而引入不必要的数据同步操作. 按需的数据同步策略需要考虑性能和正确性 2 个因素. 在正确性方面，缺少了必要的数据同步会导致程序运行出错. 性能方面，数据同步次数越少，静态分析复杂度越低，则越有利于提升性能. 在到达分析范围的边界时，如果无法确定一个定值是否在后续的控制流

中被引用，就需要保守处理，默认它会被引用. 以图 5 为例，如果以 TB 为分析单位，那么翻译 IR1 的第 7 行代码时，需要将数据从 xmm4 映射的 RVV 向量寄存器 v16 同步到其映射的浮点寄存器 fa4 上. 因为仅分析 TB 内的代码，无法确定第 7 行发生的 vec_def_lo 的定值是否会在后续的控制流中被 fp_use_lo 操作引用，只能保守地认为需要数据同步. 这样会导致过多不必要的数据同步而拖累性能.

$$IN[i] = use[i] \cup (OUT[i] - def[i]), \tag{5}$$

$$OUT[B] = \bigcup_{s \in succ(B)} IN[s], \tag{6}$$

$$IN[B] = use_B \cup \left(OUT[B] - def_B\right). \tag{7}$$

如果从起始 TB 出发，将能发现的代码都纳入一次分析中，则分析的开销过大，特别是程序的启动会变得更加缓慢. 数据同步能够被避免，是因为数据类型具有局部性. 代码中出现连续的 sfp 指令或 svec 指令时，数据同步将显著减少. 从这点上来看，分析范围的持续扩大，并不能稳步提升数据同步操作的消除数量，进而难以持续带来性能上的增益.

将单次分析限定在函数内的一条可达路径范围内，意味着我们需要处理成环的路径、函数调用和函数返回 3 种情况. 相当于处理 3 种代码边界. 当发现成环的路径时，说明对代码的探索来到了这一次逆向数据流分析的起点，此时我们保守地认为，在该程序点上，所有 xmm 寄存器都同时被 vec_use_hi、vec_use_lo 和 fp_use_lo 三个操作引用. 处理方式如算法 5 的行③所示，因为 TB 类在初始化时将其 xmm_recent_use 的所有域都初始化为 1，所以直接将 xmm_recent_use 返回即可. 在处理函数调用和函数返回方面，我们借助 System-V 的调用约定进行保守处理，即认为 call 指令对 xmm0-xmm7 有 vec_use_hi, vec_use_lo 的引用以及 ret 指令对 xmm0-xmm1 有 vec_use_hi, vec_use_lo 的引用.

4 实验评估

本文提出和实现的 3 项性能优化方案旨在提升动态二进制翻译器 RVBT 在翻译执行包含大量 SIMD 指令的 x86 应用时的运行效率，让其成为一个面向 RISC-V 平台的高性能动态二进制翻译器.

我们将在 RISC-V 平台上运行开启所有优化后的 RVBT，并翻译执行测试集的 x86-64 版本可执行文件来统计运行时间. 通过与 RISC-V 原生版测试集的运行时间对比，获得 RVBT 相对本地执行的运行效率. 这能一定程度上反映优化后的 RVBT 是否有潜力成为一个高性能的动态二进制翻译器. 同时，我们还将之与 QEMU、未加优化的基线版本 RVBT 进行横向性能对比，以评估本文提出的优化方法带来的性能提升.

我们还将评估各项优化独立开启对 RVBT 性能提升带来的贡献. 对于 SetVType 优化，我们统计了 csrr、vsetvl 以及 vsetvli 指令的静态和动态消除率，以评估其消除冗余 sew 设置的能力. 对于 SetMask 优化，我们统计了掩码设置操作的静态和动态消除率. 式(8)(9)定义了某个操作的静态和动态消除率是如何计算的. 对于 SD2Float 优化，我们统计了静态和动态同步率，计算式见式(10)(11)，以反映混合翻译中数据同步发生的频率，评估分析算法的能力. 其中，*op* 是掩码设置和 sew 设置等操作的简写.

$$op\text{的静态消除率} = \frac{\text{优化前发射}op\text{的次数} - \text{优化后发射}op\text{的次数}}{\text{优化前发射}op\text{的次数}}, \tag{8}$$

$$op\text{的动态消除率} = \frac{\text{优化前执行}op\text{的次数} - \text{优化后执行}op\text{的次数}}{\text{优化前执行}op\text{的次数}}, \tag{9}$$

$$\text{静态同步率} = \frac{\text{翻译时发射同步操作的次数}}{\text{翻译sfp类型指令的次数}}, \tag{10}$$

$$\text{动态同步率} = \frac{\text{动态执行同步操作的次数}}{\text{动态执行sfp类型指令本地码的次数}}. \tag{11}$$

4.1 实验设置与测试集介绍

我们使用 SPEC CPU 2006 基准测试集评估 RVBT 的正确性和性能. 先在 x86-64 服务器上使用 GCC 7.5.5 编译测试集以得到测试集在 x86-64 源平台上动态链接的可执行文件. 为了让编译器在指令选择过程中进行自动向量化，尽可能多地使用 SIMD 指令，编译时使用-O3 编译优化选项. 在 64 位的 RISC-V 实验平台上，使用系统自带的 GCC 13.1.1 编译器，同样采用-O3 编译优化选项编译测试集，得到动态链接的目标平台原生可执行文件. 通过分别运行基线版本的 RVBT、优化后的 RVBT 和 QEMU 来翻译执行源平台测试用例，以及直接运行目标平台原生测试用例，对比它们的执行时间，以评估 RVBT 的性能表现，以及各项优化方案对 RVBT 性能提升的贡献.

正确性是二进制翻译器的基础验证指标. SPEC CPU 2006 测试集使用了 C、C++和 Fortran 三种编程

语言, 代码量非常大, 包含了丰富的语言特性和语法现象. 其测试用例是真实世界的大型复杂应用程序, 比如编译器(gcc)、语音识别程序(sphinx3)等, 涵盖了多种典型应用场景, 非常具有代表性. SPEC CPU 2006 测试集提供了运行测试用例的脚本, 以及各个测试用例正确运行后的标准输出结果. 用户使用脚本运行测试用例后, 脚本会自动地将测试用例该次的运行结果与标准输出进行对比, 以验证测试用例的运行是否正确. 若测试用例运行正确, 脚本将给出其运行时间. 因此, SPEC CPU 2006 测试集可用于评估 RVBT 系统的正确性和性能. 若 RVBT 能正确翻译执行 SPEC CPU 2006 测试集, 则很大程度上说明本文的翻译方法和优化方法是正确的.

QEMU 是最具代表性的开源二进制翻译器之一. 在翻译 SIMD 指令上, 它选择了和 RVBT 不同的技术路线, 即使用目标平台的标量指令模拟源平台的 SIMD 指令. 而 RVBT 则使用 RISC-V 平台上同样具有单指令多数据流功能的向量扩展来翻译 SIMD 指令. QEMU 的一大设计目标是支持多平台, 但更高的翻译运行效率也同样是其核心追求, 它在 TCG-IR 上进行了各种优化以生成更高质量的本地码, 进而提升性能. 本文所提翻译器与 QEMU 进行横向对比的目的在于评估 RVBT 的基线性能水平, 并以 RVBT 的基线性能作为基准, 评估本文提出的优化方案. 这样能更加客观地评估优化方案带来的性能提升效果. 同时, 优化后的 RVBT 与 QEMU 的性能对比也在一定程度上展示了 2 种翻译 SIMD 指令的技术路线在性能上的差异, 体现了 RISC-V 向量扩展在提升 SIMD 指令翻译效率上的潜力.

在测试时, 我们使用 QEMU 最新的稳定版本 9.0.0, 并基于 Capstone 5.0.3 库对测试集进行反汇编和分析. 在运行 SPEC CPU 2006 的所有测试用例时均采用 ref 输入集. 该输入集是被 SPEC 认证的用于发布正式测试结果的输入集. 相比于 test 和 train 输入集, ref 输入集的数据规模最大, 能覆盖测试用例中的极端分支, 适合压力测试, 在获得更稳定且客观的性能数据的同时, 也能充分地测试 RVBT 的正确性.

本文的 RISC-V 实验平台为 Milk-V Pioneer 主机. 该主机搭载的 Sophon SG2042 CPU 是一款基于平头哥 C920 的高性能 RISC-V 处理器, 它支持 RISC-V v0.7.1 版本的向量扩展(即 RVV 0.7.1), 拥有 128 GB 内存, 其详细硬件配置如表 4 所示. 主机上运行了 RISC-V 64 位版本的 Fedora 38 发行版操作系统, 使用 Linux 6.1.31 内核. 实验的系统软硬件栈如图 6 所示.

Table 4 Hardware Configurations of Experiment Platform

表 4 实验平台硬件配置

硬件	配置
CPU	Sophon SG2042 核心数：64 核 主频：2 GHz L1d Cache：64 KB L1i Cache：64 KB L2 Cache：1MB/Cluster L3 Cache：64 MB System Cache 向量扩展：RVV 0.7.1
内存	128 GB DDR4 RAM

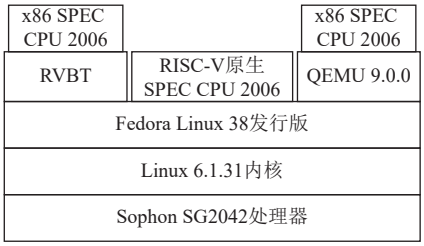


Fig. 6 Software and hardware stack of experiment platform

图 6 实验平台软硬件栈

4.2 正确性与性能评估

经实验验证, RVBT 在本文的实验平台上正确翻译执行了 x86-64 平台上 SPEC CPU 2006 测试集. 考虑到该测试集包含了真实世界的大型复杂应用程序, 对它的支持验证了 RVBT 系统的正确性.

RVBT 在开启本文提出的所有优化后, 性能得到了显著提升, 在 SPEC CPU 2006 测试集上, 运行效率平均可达到本地性能的 43.05%, 如图 7 所示. 未开启优化的 RVBT 基线版本运行效率平均只达到本地性能的 11.81%, 而 QEMU 的运行效率更低, 平均仅有本地性能的 8.64%. 优化后的 RVBT 相对其基线版本的平均加速比为 3.64, 相对 QEMU 的平均加速比为 4.98.

本文的优化方案在 SPEC CPU 2006 浮点测试集上获得了更为显著的性能提升. 如图 7 所示, 在浮点测试集上, RVBT 的运行效率平均可达到本地性能的 40.06%, 翻译执行 433.milc, 437.leslie3d, 450.soplex, 459.GemsFDTD 和 470.lbm 等 5 个测试用例的运行效率都超过了本地性能的 50%. 而未施加本文优化的 RVBT 基线版本的平均运行效率只有本地性能的 4.82%. QEMU 的平均运行效率和 RVBT 基线版本相当, 仅为本地性能的 4.81%. 在 SPEC CPU 2006 浮点测试集上, 开启所有优化后的 RVBT 相对其基线性能的最大加速比、最小加速比和平均加速比分别为 24.17, 2.61, 8.31, 相对 QEMU 的最大加速比、最小加速比和平均加速比分别为 17.74, 4.69, 8.33.

在整点测试集上, 本文提出的优化方案也能获

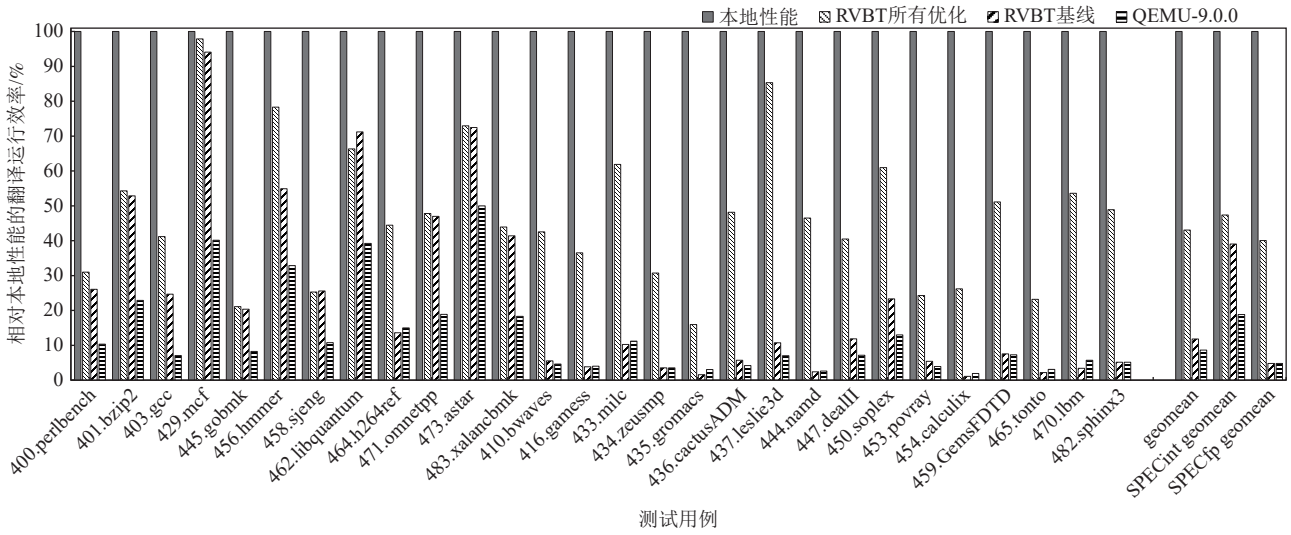


Fig. 7 Performance of full optimized RVBT, baseline RVBT and QEMU

图 7 全优化的 RVBT 与基线 RVBT 及 QEMU 的性能表现

得可观的性能提升. 如图 7 所示, 在整点测试集上, RVBT 的运行效率平均可达到本地性能的 47.39%. RVBT 基线版本的平均运行效率为本地性能的 39.04%, 而 QEMU 的平均运行效率仅为本地性能的 18.84%. 在整点测试集上, 开启所有优化后的 RVBT 相对 RVBT 基线性能的平均加速比为 1.21, 相对 QEMU 的平均加速比为 2.52. 开启优化的 RVBT 和 QEMU 翻译运行 429.mcf 都获得了最高的运行的效率, RVBT 可以达到本地性能的 97.90%, 而 QEMU 只能达到本地性能的 40.14%. RVBT 仅在翻译 400.perlbench、445.gobmk 和 458.sjeng 这 3 个测试用例时效率低于本地性能 30%, 翻译其他的 9 个测试用例的效率均高于本地性能 30%. 而 QEMU 仅翻译 429.mcf、456.hmmer、462.libquantum 和 473.astar 这 4 个测试用例的效率高于本地性能的 30%.

翻译浮点操作的运行效率低下是跨指令集的动态二进制翻译器普遍面临的挑战之一, 这严重阻碍了该技术用于翻译运行人工智能、科学计算、图形系统和多媒体等含有密集浮点操作的应用程序[38]. 基线版本的 RVBT 翻译浮点测试集的效率比其翻译整点测试集慢了 8.10 倍, 而在开启本文提出的 3 项优化方案后, 只慢 1.18 倍. 作为对比, QEMU 翻译浮点测试集的效率比其翻译整点测试集慢了 3.91 倍. 本文提出的优化方案有效提升了翻译浮点操作的运行效率, 有助于拓宽动态二进制翻译器的应用场景. 经过优化后, RVBT 在性能上全面超越了 QEMU, 相比其优化前的基线版本, 性能也有大幅提升, 充分说明了本文提出的 3 项优化方案的有效性.

各项优化单独开启的 RVBT 以及 QEMU 相对 RVBT 基线的加速比如图 8 所示. 可观察到 2 个现象: 1)在整点测试集上, 3 项优化带来的性能提升都相对有限, 在浮点测试集上, 3 项优化都能带来显著的性能提升; 2)SetVType 优化带来的性能提升最为明显. 这和测试集上的指令构成以及各项优化本身的特点有关.

我们对测试集的 x86-64 源平台测试用例进行反汇编, 并统计了各个测试用例中 SIMD 指令的占比, 如图 9 所示. 在 SPEC CPU 2006 整个测试集中, SIMD 指令的平均占比为 9.68%. 但 SIMD 指令在整点测试集和浮点测试集中的占比差异非常大. 其中, 在整点测试集中的平均占比仅为 3.31%, 而在浮点测试集中的占比高达 21.62%, 二者相差了 6.53 倍. 这解释了为什么本文提出的优化方案在浮点测试集上获得了更显著的性能提升. 另外, 3 项优化方案中, 只有 SetVType 优化对所有的 SIMD 指令都具有优化效果, 所以单独开启时对性能提升最为显著.

4.3 各优化方案的效果分析

4.3.1 SetVType 优化效果分析

单独开启 SetVType 优化后, RVBT 在 SPEC CPU 2006 测试集上的运行效率可达到本地性能的 36.29%, 相对于 RVBT 基线和 QEMU 基线的平均加速比分别为 3.07 和 4.2, 对 csrr、vsetvl 和 vsetvli 指令的动态消除率分别达到了 100%, 100% 和 56.31%.

翻译函数将 SS、SD、PS、PD 和 PI 类型的 SIMD 指令翻译成 RISC-V 向量指令时, 均需要设置 vtype 寄存器. 因此, SetVType 优化对所有类型的 SIMD 指令的翻译均具有优化效果.

在整点测试集上, 虽然 SIMD 指令占比较低, 但

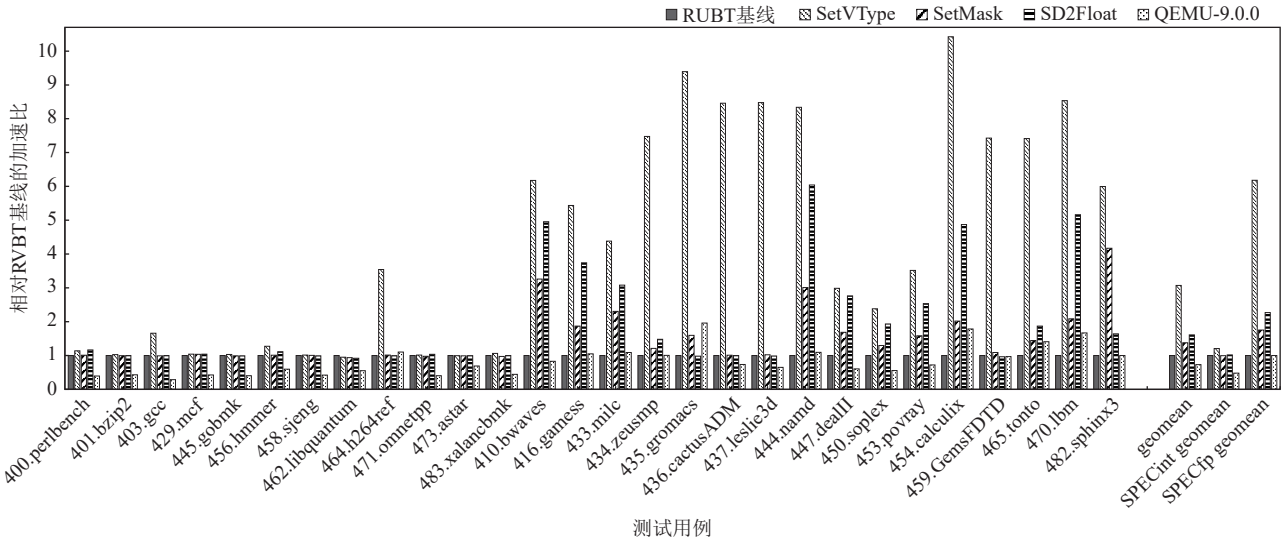


Fig. 8 Performance of RVBT with each optimized individual turn-on and QEMU are compared with RVBT baseline

图 8 各项优化单独开启的 RVBT 及 QEMU 相对于 RVBT 基线的性能表现

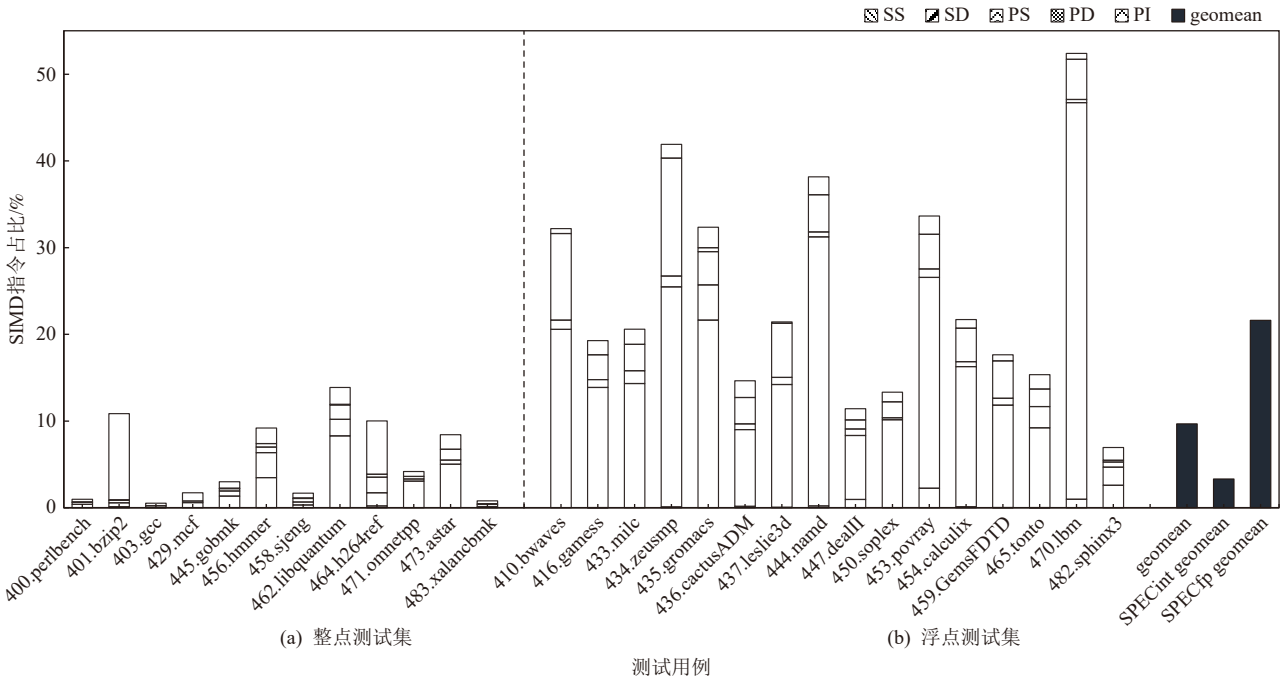


Fig. 9 Static proportion of SIMD instructions in SPEC CPU 2006 test program

图 9 SPEC CPU 2006 测试程序中 SIMD 指令的静态占比

开启优化后依然能获得一定的性能提升, 让 RVBT 的运行效率达到本地性能的 47.21%, 相对 RVBT 基线和 QEMU 基线的平均加速比分别为 1.21 和 2.51.

在浮点测试集上, SetVType 优化可以大幅提升 RVBT 的性能, 让其运行效率达到本地性能的 29.79%, 相对 RVBT 基线的最小加速比、最大加速比和平均加速比分别为 2.38, 10.42 和 6.18. 翻译执行 437.leslie3d 的运行效率相比 RVBT 基线提高了 8.48 倍, 达到了本地性能的 91.16%. 与 QEMU 相比, 获得的最小加速比、最大加速比和平均加速比分别为 4.01、12.98、6.19.

我们统计了优化开启后 vsetvli、vsetvl 和 csrr 指令的静态消除率和动态消除率. 经过 SetVType 优化后, vsetvl 和 csrr 指令在测试集的所有测试用例中均被完全消除, 静态消除率和动态消除率都是 100%. 经实验测定, 执行 vsetvl 和 csrr 指令需要 28 个时钟周期, 而执行一条向量加法指令只需要 0.58 个时钟周期. 所以, 消除冗余的 vsetvl 和 csrr 指令非常有必要.

在整点测试集上, vsetvli 的平均静态消除率和平均动态消除率分别为 44.20% 和 48.88%, 在浮点测试集上分别为 51.72% 和 62.62%, 如图 10 所示. 动态指

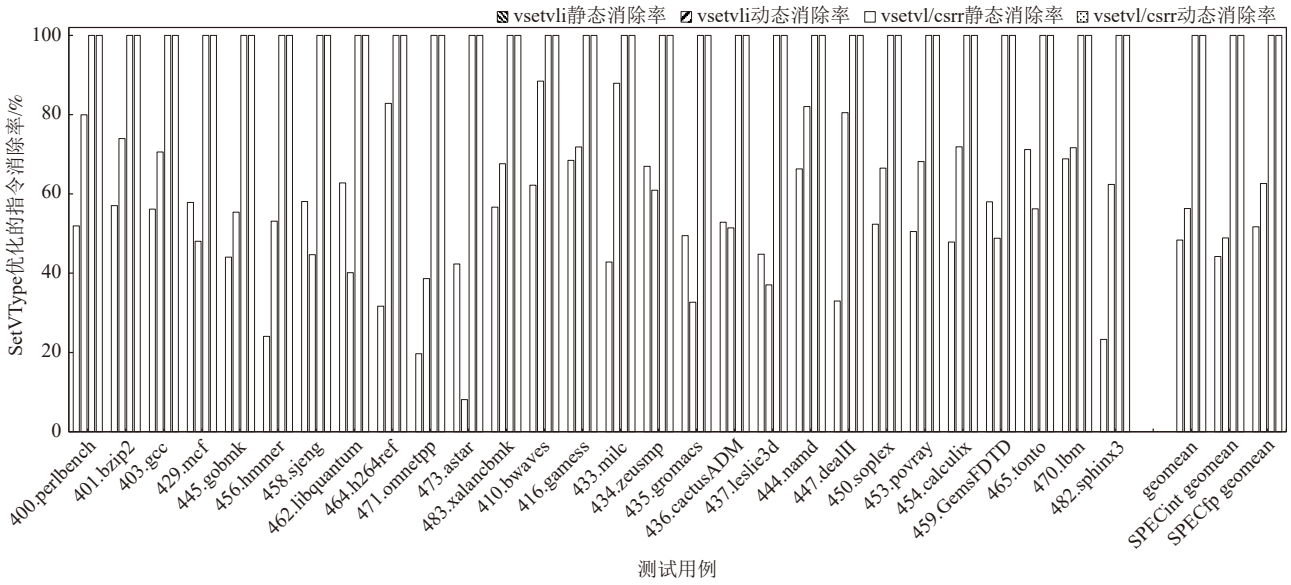


Fig. 10 Dynamic and static instruction eliminate rates of SetVType optimization

图 10 SetVType 优化的动静态指令消除率

令消除率与性能提升的幅度是正相关的. 这一定程度上解释了为什么 SetVType 优化在浮点测试集上能获得比整点测试集更大的性能提升.

4.3.2 SetMask 优化效果分析

单独开启 SetMask 优化后, RVBT 在 SPEC CPU 2006 浮点测试集上获得了性能提升, 相对 RVBT 基线平均加速比为 1.76. 该优化能消除冗余的掩码寄存器设置指令序列. 对掩码寄存器的设置主要是在翻译 SS 和 SD 类型指令时引入的. 浮点基准测试集中包含大量的这 2 种指令, 平均占比达到了 15.33%, 在 470.lbm 上甚至达到了 46.73%. 实施 SetMask 优化后, 掩码设置操作的静态和动态的平均消除率分别为 85.73% 和 74.66%, 如图 11 所示. 翻译执行浮点测试集时, 在掩码设置操作被大量消除后, 性能获得了明显的提升.

在整点测试集上, SetMask 是否开启对性能几乎没有影响. 在整点基准测试集的指令中, SS 和 SD 类型的指令占比非常小, 平均占比仅为 1.07%. 尽管 SetMask 优化在整点测试集上对掩码设置的静态和动态的平均消除率达到了 55.87% 和 18.33%, 但受限于可优化的指令总体量太小, 没有带来性能提升. 得益于该优化很小的分析开销, RVBT 开启该优化后, 在整点测

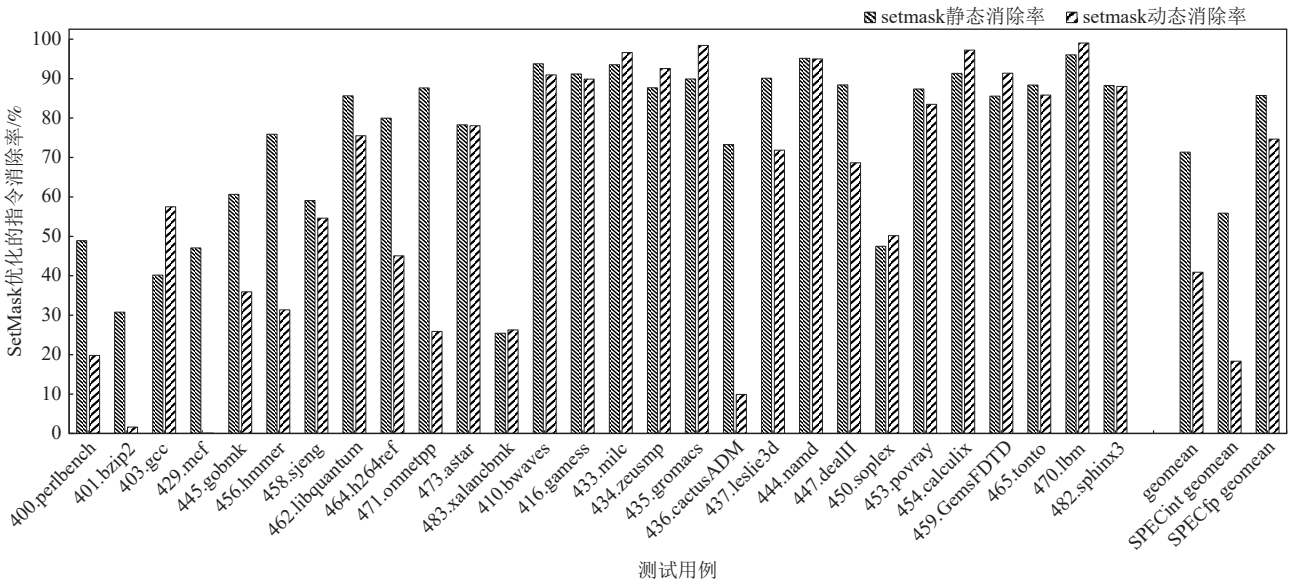


Fig. 11 Dynamic and static instruction eliminate rates of SetMask optimization

图 11 SetMask 优化的动静态指令消除率

试集上也几乎没有观察到性能下降的现象.

4.3.3 SD2Float 优化效果分析

单独开启 SD2Float 优化后, RVBT 可以在 SPEC CPU 2006 浮点测试集上获得较大的性能提升, 相对 RVBT 基线的平均加速比为 2.27. SD2Float 优化针对的是 SD 类型的 SIMD 指令. 该类型的指令在整点测试集上的平均占比仅为 0.57%, 体量太小, 使得 SD2Float 优化在整点测试集上平均仅能获得 2% 的性能提升. 在整个测试集上对 RVBT 基线的平均加速比为 1.61.

SD2Float 优化能提升 RVBT 运行效率的关键在于降低了寄存器之间的数据同步频率. 我们使用静态和动态同步率来反映 SD2Float 优化中的数据同步频率, 它们的定义如式(10)和式(11)所示. 在浮点测试集中, SD2Float 优化平均的静态同步率和动态同步率分别为 55.61% 和 67.35%, 如图 12 所示. 动态同步率最低的基准测试用例是 454.calculix, 仅为 13.66%, 说明本文的按需数据同步算法有效降低了数据同步频率. 在整点测试集, 静态和动态的同步率都要高出许多. 但整点测试集中浮点操作相关的指令占比非常少, 高同步率下引入的同步指令执行次数相比于总的指令执行次数而言, 绝对数量也就非常小, 所以对性能的影响很小.

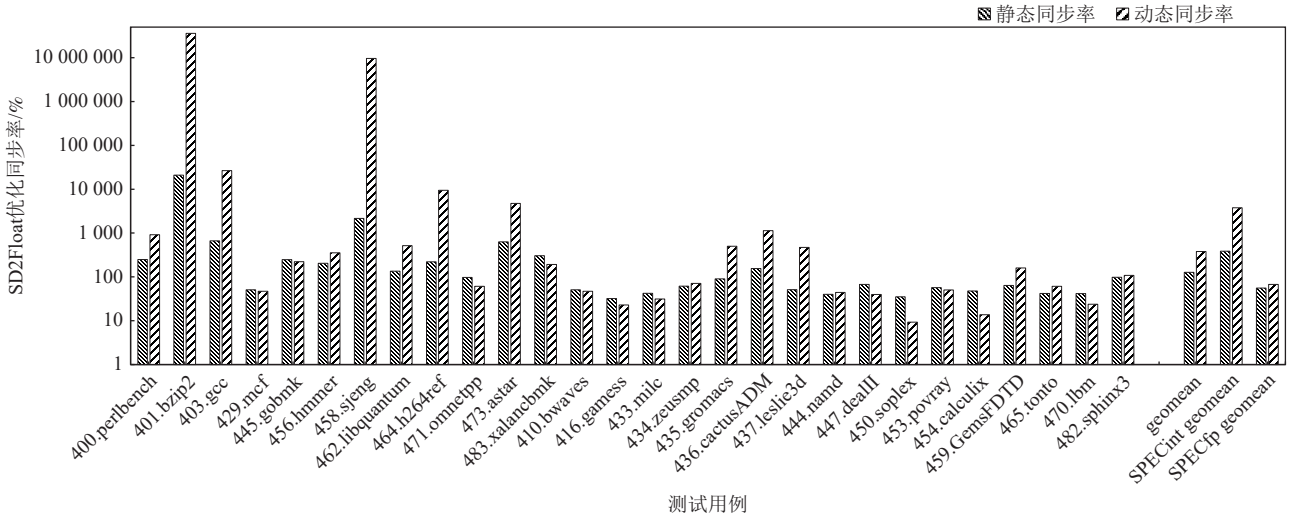


Fig. 12 Dynamic and static synchronization rates of SD2Float optimization

图 12 SD2Float 优化的动静态同步率

在整点测试集中, 有个别测试用例的动态同步率非常高. 产生高动态同步率的原因是同步指令被发射在一个循环内, 而引发同步的 SD 指令在循环外. 如图 13 所示, 图 13(a)是一段 C 语言代码, 图 13(b)是其对应的 x86-64 汇编代码, 其中图 13(a)的第 8 和第 9 行代码对应图 13(b)的第 9 行汇编代码, 图 13(a)中的第 11 行代码对应图 13(b)的第 12 行汇编代码. 图 13(b)被 RVBT 翻译成了图 13(c)所示的 RISC-V 汇编代码. 在分析图 13(b)的第 9 行代码时, SD2Float 优化的数据同步算法发现 addpd 指令定义了 xmm1 寄存器的低 64 位, 同时第 14 行的 addsd 指令使用了该寄存器的低 64 位, 即第 9 行代码对 xmm1 的定义

```
1  double array[2] = {1.1, 2.2};
2
3  void foo()
4  {
5      ...
6      for(int i = 0; i < 10; i++)
7      {
8          array[0] +=1;
9          array[1] +=1;
10     }
11     array[0] += 2.0;
12     ...
13 }
```

(a) C语言代码

```
1  foo:
2  ...
3  movaps array(%rip), %xmm1 # %xmm1 = array
4  movapd .LC0(%rip), %xmm0 # %xmm0 = {1, 1}
5  movl   $100,%edx
6  xorl   %eax, %eax
7  .L1:                          # loop start
8      addl    $1, %eax
9      addpd   %xmm0, %xmm1  # %xmm1 += {1, 1}
10     cmpl    %edx, %eax
11     jne .L1                   # loop end
12     movaps %xmm1, array(%rip)
13 .L2:
14     addsd   .LC2(%rip), %xmm1  # %xmm1[0] +=2.0
15     movsd   %xmm1, array(%rip)
16 ...
```

(b) x86-64汇编代码

```
1  foo:
2  ...
3  vlbu.v v8, gp, disp1        # %xmm1 = array
4  vlbu.v v6, gp, disp2        # %xmm0 = {1, 1}
5  addi a2, zero, 100
6  addi a7, zero, 0
7  .L1:                        # loop start
8      addi a7, a7, 1
9      vfadd.vv v8, v8, v6     # %xmm1 += {1, 1}
10     数据同步: v8 → fa1
11     sub temp, a7, a2
12     bne temp, .L1           # loop end
13     vsb.v v8, gp, disp1
14 .L2:
15 fadd.d fa1 fa1 ft8          # %xmm1[0] +=2.0
16 vsb.v v8, gp, disp1, v0
17 ...
```

(c) RISC-V汇编代码

Fig. 13 Example for high synchronization rate code segment

图 13 高同步率代码片段示例

被第 14 行代码引用了. 由于 addpd 和 addsd 指令分别被翻译成了 RISC-V 的向量指令和浮点指令. 所以，在将 addpd 指令翻译成 vfaddd 指令后，还必须将映射 xmm1 的向量寄存器 v8 中的值同步到浮点寄存器 fa1 中，如图 13(c)的第 9 和第 10 两行代码所示. 当图 13(c)的代码执行时，同步指令位于循环中，将执行 100 次，而引起同步的指令 addsd 在循环外，只执行 1 次，动态同步率为 10 000%. 在上述例子中，若将同步指令放到循环结构之外，就能大幅降低数据同步的开销. 这有赖于循环结构的分析识别和优化，不在本文的探讨范围之内.

5 讨　　论

本文针对 SIMD 指令跨架构翻译为向量指令面临的编程模型适配问题，基于程序局部性特征提出了 3 项优化方案，并将它们实现在 x86 到 RISC-V 平台的动态二进制翻译器中. 这 3 项优化方案是架构无关的，具备跨架构的适应性. 不同架构处理器的 SIMD 扩展具有较大的差异. 在跨架构的二进制翻译中，若源平台的 SIMD 扩展是将其指令操作的数据元素类型和个数硬编码到指令操作码中，且目标平台的向量扩展是动态配置其指令操作的数据元素类型和个数，则适用本文的 SetVType 和 SetMask 优化. 若标量浮点操作和 SIMD 操作在源平台共用一套寄存器，而在目标平台使用 2 套独立的寄存器，则适用本文的 SD2Float 优化. 将 ARM 平台的 NEON 指令翻译为 RISC-V 的向量指令是适用本文 3 项优化方案的一个典型例子.

QEMU 使用目标平台的标量指令模拟源平台的 SIMD 指令. 已经有相关研究通过在 QEMU 中添加向量 TCG-IR，以实现对目标平台上单指令多数据流硬件资源的利用，进而提升翻译性能[42-44]. 本文提出的 3 项优化方案和这些研究的方案是正交的. 若翻译的源平台和目标平台符合本文优化方案的适用条件，则可将本文的优化方案应用到 QEMU 中. 其他选择将源平台 SIMD 指令翻译为目标平台向量指令的二进制翻译器，也可使用本文提出的优化方案.

6 相关工作

二进制翻译技术能将一种体系结构的二进制代码转换成另一种体系结构的二进制代码，实现在二进制层面进行跨平台软件迁移[19-20]，可以用于遗产代码迁移和实现不同架构之间的软件通用[21,23-24]. 早在上世纪 90 年代，微软公司就已经推出了商业用的二进制翻译器 FX!32[45]. 随着新硬件架构的不断推出，二进制翻译技术被广泛用于将成熟架构上的软件移植到新架构上运行，以迅速弥补新架构上的软件生态空缺.

经过多年的发展，已经有大量的二进制翻译器被开发出来[22]. 但支持以 RISC-V 作为目标平台的动态二进制翻译器还比较少见，以 QEMU, Box86/64[39] 和 DBT-FEMU[40] 为代表的翻译器能将 x86 代码翻译到 RISC-V 平台，其中 QEMU 和 Box86/64 是开源的. 这 3 款翻译器的侧重点不同，QEMU 追求多平台，首先将源平台的二进制程序翻译成其中间表示 TCG-IR，在对中间表示进行优化后再将其转换成目标平台的代码. 通过增加一层平台无关的 TCG-IR, QEMU 在多源平台和多目标平台支持上具有优势，但因为使用内存模拟目标平台的寄存器以及使用标量指令模拟 SIMD 指令的语义等原因，QEMU 翻译质量较低[40] 且性能较差[46]. Box86/64 支持以 ARM 和 RISC-V 作为目标平台，对 x86 程序进行动态二进制翻译. 它主要依赖函数库本地化技术来提升性能，通过直接使用 RISC-V 原生的 glibc 等动态库，减少翻译执行的代码量[22]，同时通过 JIT 引擎提升性能[2]. DBT-FEMU 主要利用了 RISC-V 的 B 扩展和 P 扩展对整点操作的翻译进行优化，在 SPEC CPU 2006 测试集的整点测试集上，平均翻译效率能达到本地性能的 33.54%. 和上述支持 RISC-V 平台的翻译器相比，RVBT 在性能上表现得非常有竞争力.

动态二进制翻译系统普遍面临性能瓶颈，特别是在跨指令集架构进行翻译时容易发生性能衰减，且源架构和目标架构的差异越大，性能下降越明显[27,47]. 而性能是跨指令集的动态二进制翻译系统取得成功的关键因素[48]，因此有大量关于提升翻译性能的研究. 源架构与目标架构之间的差异会导致翻译出来的本地码出现严重的指令膨胀[2,27]. 因此，一个重要的优化方向是消除冗余操作，比如标志位优化[2,25,49-51] 遵循的就是这一思想，通过程序分析等技术消除冗余的标志位设置，缓解代码膨胀问题. RVBT 在将 SIMD 指令翻译成 RVV 指令时，也观察到了指令膨胀的现象. 本文的一个重点优化方向是利用程序中数据类型的局部性特征消除冗余的 sew 和掩码设置，缓解因 SIMD 与 RVV 在编程模型上的显著差异导致的指令膨胀问题. 优化方案的指导思想都是消除冗余操作，但解决的问题不一样. 同时，RVBT 中也集成了上述部分标志位优化技术.

不同平台上的 SIMD 扩展在设计和实现上呈现持续分化的趋势[52], 普遍存在差异, 这为跨平台的 SIMD 指令翻译带来了挑战[53]. 过往对 SIMD 指令的翻译优化研究主要聚焦在 x86 和 ARM 这 2 个平台上, 集中在x86-x86[54-55]、ARM-x86[52,56-57] 以及ARM-ARM[42,48,58-61] 的翻译中, 早期也有研究探索 x86 到 Itanium(IA64)上的 SIMD 翻译优化[62]. 主要研究如何充分利用目标平台上的 SIMD 计算资源来提升性能, 比如动态向量化等. 而 x86 上的 SIMD 和 RISC-V 平台上的 RVV 因编程模型的巨大差异导致翻译效率不高这一问题还没有被关注和研究.

SIMD 技术在不断发展, 新推出的 SIMD 硬件具有更强大的计算能力. 但面向旧 SIMD 硬件的遗产代码往往无法利用新硬件来提升性能. 同架构上 SIMD 指令的翻译优化主要是为了让遗产代码能充分利用新平台上更加强大的 SIMD 硬件资源, 比如利用新平台上位宽更长的寄存器, 优化技术包括循环信息重建、动态向量化等[54-55,58].

跨架构 SIMD 翻译优化的核心思想是充分利用目标平台上的 SIMD 资源, 弥合跨平台 SIMD 的语义鸿沟, 比如在目标平台上寻找语义与源平台接近的指令进行翻译[59-60,63], 探索更好的寄存器映射[62] 和分配方案, 以及动态向量化[64-66] 等优化思路. Li 等人[62] 在将 x86 程序翻译到 Itanium 平台时研究了在目标平台寄存器支持的数据类型比源平台寄存器支持的更少时, 如何将 1 个源平台寄存器映射到多个目标平台寄存器. 文献 [48, 52, 54−55, 61] 研究了在目标平台寄存器位宽比源平台的更长时, 如何通过更好的寄存器分配和指令合并等方法获得更好的性能. Wu 等人[56] 还研究了将 ARM 的通用寄存器映射到 x86 的 xmm 寄存器上, 以提升翻译性能. 还有一类研究先将源平台代码转成 LLVM IR, 通过 LLVM 的编译优化来选择目标平台的 SIMD 指令, 达到使用目标平台 SIMD 进行加速的目的[46,57,59,67]. 针对 QEMU 使用标量指令模拟 SIMD 指令语义, 文献 [42−44] 设计了向量 TCG-IR 来表示 SIMD 指令, 并改进翻译 SIMD 的 helper 函数, 文献 [68] 则将 TCG-IR 转成 LLVM IR.

本文充分利用目标平台上同样具备数据级并行功能的 RISC-V RVV 扩展来翻译 x86 的 SIMD 指令, 强调对目标平台上硬件资源的充分利用. 同时, 通过冗余操作消除和混合翻译, 弥合 2 个平台在编程模型上的语义鸿沟, 实现性能提升.

二进制翻译在处理浮点操作时普遍性能较低, 提升性能的方法一般是使用目标平台的浮点计算单元替代软件模拟的翻译[38]. 文献 [47] 通过动态的浮点寄存器分配来提升性能, 文献 [69] 则针对 x86 的浮点栈提出了扩展虚拟栈(extending virtual stack)处理方案, 让源平台的浮点寄存器可以直接映射到目标平台的浮点寄存器中. 本文使用了混合的翻译方法, 同时使用 RISC-V 的 RVV 扩展和浮点扩展来翻译 SIMD 中的浮点运算操作, 即使用 RISC-V 的浮点扩展翻译双精度标量浮点运算, 使用 RVV 扩展翻译打包浮点运算等其他浮点操作, 并通过静态分析实现按需的数据同步以提升翻译运行的效率.

7 结 论

RISC-V 在 HPC 领域的崛起面临软件生态滞后的关键挑战, 而动态二进制翻译技术为跨架构移植 x86/ARM 成熟的软件生态提供了高效路径. 本文揭示了 SIMD 指令翻译至 RVV 时遭遇性能瓶颈的根源在于两者在编程模型上存在显著差异. 针对这一问题, 本文提出了 3 项创新的优化方案, 充分利用代码操作的数据类型具有局部性这一特点, 通过消除翻译过程中的冗余操作, 降低混合翻译浮点操作所需的数据同步频率, 系统性地提升了 SIMD 到 RVV 的翻译效率. 特别是将翻译浮点操作的效率提升到了接近翻译整点操作的水平. 这为 RISC-V 在 HPC 场景的软件生态突破提供了一种可能的解决方案. 实验显示, 经过本文优化后, 翻译器获得了显著的性能提升, 翻译运行 SPEC CPU 2006 的整点测试集和浮点测试集时, 平均运行效率分别达到了本地性能的 47.39% 和 40.06%, 远超 QEMU 的 18.84% 和 4.81%. 相对优化前的加速比分别达到了 1.21 和 8.31, 相对 QEMU 的加速比则分别达到了 2.52 和 8.33.

作者贡献声明: 赖远明是论文工作的主要完成人, 提出、设计和实现了本文的 3 项优化方案, 并分析实验结果, 撰写和修改论文; 李亚龙和胡瀚之参与了优化方案的编码实现, 运行了部分实验; 谢梦瑶对优化方案的实现细节提出了改进建议; 王喆对优化方案的设计提出了建议, 并修改论文; 武成岗为论文的撰写提供了建议和指导.

参 考 文 献

[1] Waterman A, Lee Y, Patterson D A, et al. The RISC-V instruction set manual, volume I: user-level ISA, version 2.0[EB]. EECS

Department, University of California, Berkeley, Tech. Rep. UCB/EECS−2014−54, 2014: 4

[2] Li Chunqiang, Liu Zhiwei, Shang Yunhai, et al. A hardware non-invasive mapping method for condition bits in binary translation[J]. Electronics, 2023, 12(14): 3014

[3] RISC-V International. There will be 62.4 billion RISC-V processor cores in operation by 2025[EB/OL]. (2020-04-02)[2024-11-12]. https://riscv.org/ecosystem-news/2020/04/there-will-be-62-4-billion-risc-v-processor-cores-in-operation-by-2025-francois-gauthier-lembarque

[4] Weaver D, McIntosh-Smith S. An empirical comparison of the RISC-V and AArch64 instruction sets[C]//Proc of the SC'23 Workshops of the Int Conf on High Performance Computing, Network, Storage, and Analysis. New York: ACM, 2023: 1557−1565

[5] Xi Wang, Leidel J D, Williams B, et al. Xbgas: A global address space extension on RISC-V for high performance computing[C]//Proc of 2021 IEEE Int Parallel and Distributed Processing Symp (IPDPS). Piscataway, NJ: IEEE, 2021: 454−463

[6] Perez B, Fell A, Davis J D. Coyote: An open source simulation tool to enable RISC-V in HPC[C]//Proc of 2021 Design, Automation & Test in Europe Conf & Exhibition (DATE). Piscataway, NJ: IEEE, 2021: 130−135

[7] Bartolini A, Ficarelli F, Parisi E, et al. Monte Cimone: Paving the road for the first generation of RISC-V high-performance computers[C]// Proc of 2022 IEEE 35th Int System-on-Chip Conf (SOCC). Piscataway, NJ: IEEE, 2022: 1−6

[8] Chen Chen, Xiang Xiaoyan, Liu Chang, et al. Xuantie−910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension: Industrial product[C]//Proc of 2020 ACM/IEEE 47th Annual Int Symp on Computer Architecture (ISCA). Piscataway, NJ: IEEE, 2020: 52−64

[9] Brown N, Jamieson M. Performance characterisation of the 64-core SG2042 RISC-V CPU for HPC[C]//Proc of Int Conf on High Performance Computing. Berlin: Springer, 2025: 354−367

[10] 北京开源芯片研究院. 2024 中关村论坛 10 项重大科技成果——第三代“香山”开源高性能 RISC-V 处理器核对外发布. [EB/OL]. (2024-04-25)[2024-11-12]. https://www.bosc.ac.cn/newsinfo/7105467.html?templateId=564439

[11] Brown N, Jamieson M, Lee J, et al. Is RISC-V ready for HPC prime-time: Evaluating the 64-core Sophon SG2042 RISC-V CPU[C]//Proc of the SC'23 Workshops of the Int Conf on High Performance Computing, Network, Storage, and Analysis. New York: ACM, 2023: 1566−1574

[12] Sophon. Milk-V pioneer board. [EB/OL]. [2024-11-12]. https://sophon-static.sophon.cn/product/introduce/pioneerBoard.html

[13] Xuantie. 如意 BOOK 甲辰版. [EB/OL]. [2024-11-12]. https://www.xrvm.cn/product/xuantie/4321312678808195072

[14] Dakić V, Mršić L, Kunić Z, et al. Evaluating ARM and RISC-V architectures for high-performance computing with docker and kubernetes[J]. Electronics, 2024, 13(17): 3494

[15] Canal R, Carlo S D, Gizopoulos D, et al. Vitamin-V: Expanding open-source RISC-V cloud environments[J]. arXiv preprint, arXiv: 2407.00052, 2024

[16] Canal R, Chenet C, Arelakis A, et al. Vitamin-V: Virtual environment and tool-boxing for trustworthy development of RISC-V based cloud services[C]//Proc of the 26th Euromicro Conf on Digital System Design (DSD). Piscataway, NJ: IEEE, 2023: 302−308

[17] Nick Brown. RISC-V for HPC: Where we are and where we need to go[J]. arXiv preprint, arXiv: 2406.12398, 2024

[18] Saidova J. RISC-V architecture and its role in the near future[J]. Journal of Advanced Scientific Research, 2024, 5(9): 54−67

[19] Chen Long, Wu Chenggang, Xie Haibin, et al. Graph matching method for parsing multi-target branching sentences in binary translation[J]. Journal of Computer Research and Development, 2008, 45(10): 1789−1798 (in Chinese)
(陈龙, 武成岗, 谢海斌, 等. 二进制翻译中解析多目标分支语句的图匹配方法[J]. 计算机研究与发展, 2008, 45(10): 1789−1798)

[20] Yang Hao, Tang Feng, Xie Haibin, et al. Library function processing in binary translation[J]. Journal of Computer Research and Development, 2006, 43(12): 2174−2179 (in Chinese)
(杨浩, 唐锋, 谢海斌, 等. 二进制翻译中的库函数处理[J]. 计算机研究与发展, 2006, 43(12): 2174−2179)

[21] Tang Feng, Wu Chenggang, Zhang Zhaoqing, et al. Binary translation application level exception handling[J]. Journal of Computer Research and Development, 2006, 43(12): 2166−2173 (in Chinese)
(唐锋, 武成岗, 张兆庆, 等. 二进制翻译应用级异常处理[J]. 计算机研究与发展, 2006, 43(12): 2166−2173)

[22] Xie Wenbing, Tian Xue, Qi Fengbin, et al. A review of binary translation technology[J]. Jouinal of Software, 2024, 35(6): 2687−2723 (in Chinese)
(谢汶兵, 田雪, 漆锋滨, 等. 二进制翻译技术综述[J]. 软件学报, 2024, 35(6): 2687−2723)

[23] Li Jianjun, Wu Chenggang, Wei-Chung H. Efficient and effective misaligned data access handling in a dynamic binary translation system[J]. ACM Transactions on Architecture and Code Optimization, 2011, 8(2): 1−29

[24] Li Jianjun, Wu Chenggang, Wei-Chung H. An evaluation of misaligned data access handling mechanisms in dynamic binary translation systems[C]//Proc of 2009 Int Symp on Code Generation and Optimization. Piscataway, NJ: IEEE, 2009: 180−189

[25] Tang Feng, Wu Chenggang, Feng Xiaobing, et al. Marker linear analysis algorithm based on dynamic feedback[J]. Journal of Software, 2007, 18(7): 1603−1611 (in Chinese)
(唐锋, 武成岗, 冯晓兵, 等. 基于动态反馈的标志位线性分析算法[J]. 软件学报, 2007, 18(7): 1603−1611)

[26] Bellard F. QEMU, a fast and portable dynamic translator[C]//Proc of USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2005, 41(46): 41−46

[27] Xie Benyi, Yan Yue, Yan Chenghao, et al. An instruction inflation analyzing framework for dynamic binary translators[J]. ACM Transactions on Architecture and Code Optimization, 2024, 21(2): 1−25

[28] Georganas E, Avancha S, Banerjee K, et al. Anatomy of high-performance deep learning convolutions on SIMD architectures[C]// Proc of SC18: Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2018:

830−841

[29] Zhang Jiyuan, Franchetti F, Low T M. High performance zero-memory overhead direct convolutions[C/OL]//Proc of Int Conf on Machine Learning. 2018[2024-11-12]. https://proceedings.mlr.press/v80/zhang18d.html

[30] Park D, Egger B. Improving throughput-oriented LLM inference with CPU computations[C]//Proc of the 2024 Int Conf on Parallel Architectures and Compilation Techniques. New York: ACM, 2024: 233−245

[31] Shen Haihao, Chang Hanwen, Dong Bo, et al. Efficient LLM inference on CPUs[J]. arXiv preprint, arXiv: 2311.00502, 2023

[32] Shen Haihao, Meng Hengyu, Dong Bo, et al. An efficient sparse inference software accelerator for transformer-based language models on cpus[J]. arXiv preprint, arXiv: 2306.16601, 2023

[33] Jiang Xuanlin, Zhou Yang, Cao Shiyi, et al. Neo: Saving GPU memory crisis with CPU offloading for online LLM inference[J]. arXiv preprint, arXiv: 2411.01142, 2024

[34] Bradski G. The openCV library[J]. Dr. Dobb's Journal: Software Tools for the Professional Programmer, 2000, 25((11):): 120−123

[35] Tomar S. Converting video formats with FFmpeg[J]. Linux Journal, 2006, 2006(146): 10

[36] Shuja J, Gani A, ur Rehman M H, et al. Towards native code offloading based MCC frameworks for multimedia applications: A survey[J]. Journal of Network and Computer Applications, 2016, 75: 335−354

[37] Akshintala A, Jain B, Chia-Che Tsai C C, et al. x86−64 instruction usage among C/C++ applications[C]//Proc of the 12th ACM Int Conf on Systems and Storage. New York: ACM, 2019: 68−79

[38] Cota E G, Carloni L P. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation[C]//Proc of the 15th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments. New York: ACM, 2019: 74−87

[39] PtitSeb. Box64[EB/OL]. [2024-11-12]. https://github.com/ptitSeb/box64

[40] Yu Zihao, Chen Lu, Shun Ninghui, et al. Dynamic binary translation code quality optimization method targeting RISC-V[J]. Journal of Computer Research and Development, 2023, 60(10): 2322−2334 (in Chinese)
(余子濠, 陈璐, 孙凝晖, 等. 以 RISC-V 为目标的动态二进制翻译代码质量优化方法[J]. 计算机研究与发展, 2023, 60(10): 2322−2334)

[41] Yang Zhaoxin, Chen Xuehai, Wang Liangpu, et al. MFHBT: Hybrid binary translation system with multi-stage feedback powered by LLVM[C]//Proc of Int Symp on Advanced Parallel Processing Technologies. Berlin: Springer, 2023: 310−325

[42] Michel L, Fournel N, Pétrot F. Speeding-up SIMD instructions dynamic binary translation in embedded processor simulation[C]//Proc of 2011 Design, Automation & Test in Europe. Piscataway, NJ: IEEE, 2011: 1−4

[43] Fu Shengyu, Wu J J, Hsu W C. Improving SIMD code generation in QEMU[C]//Proc of 2015 Design, Automation & Test in Europe Conf & Exhibition (DATE). Piscataway, NJ: IEEE, 2015: 1233−1236

[44] Fu Shengyu, Hong Dingyong, Wu J J, et al. SIMD code translation in an enhanced HQEMU[C]//Proc of 2015 IEEE 21st Int Conf on Parallel and Distributed Systems (ICPADS). Piscataway, NJ: IEEE, 2015: 507−514

[45] Chernoff A, Hookway R. DIGITAL FX!32 running 32-bit x86 applications on Alpha NT[C]//Proc of Large-Scale System Administration of Windows NT Workshop. Berkeley, CA: USENIX Associatio, 1997: 37−42

[46] Fu Shengyu, Hong Dingyong, Liu Yuping, et al. Efficient and retargetable SIMD translation in a dynamic binary translator[J]. Software: Practice and Experience, 2018, 48(6): 1312−1330

[47] D'Antras A, Gorgovan C, Garside J, et al. Low overhead dynamic binary translation on ARM[C]//Proc of the 38th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2017: 333−346

[48] Hong Dingyong, Fu Shengyu, Liu Yuping, et al. Exploiting longer SIMD lanes in dynamic binary translation[C]//Proc of 2016 IEEE 22nd Int Conf on Parallel and Distributed Systems (ICPADS). Piscataway, NJ: IEEE, 2016: 853−860

[49] Ma Xiangning, Wu Chenggang, Tang Feng, et al. Flag bit optimization technology in binary translation[J]. Journal of Computer Research and Development, 2005, 42(2): 329−337 (in Chinese)
(马湘宁, 武成岗, 唐锋, 等. 二进制翻译中的标志位优化技术[J]. 计算机研究与发展, 2005, 42(2): 329−337)

[50] Wang Wenwen, Wu Chenggang, Bai Tongxin, et al. Patterned translation method of flag bits in binary translation[J]. Journal of Computer Research and Development, 2014, 51(10): 2336−2347 (in Chinese)
(王文文, 武成岗, 白童心, 等. 二进制翻译中标志位的模式化翻译方法[J]. 计算机研究与发展, 2014, 51(10): 2336−2347)

[51] Zeng Hongqing, Xie Min, Dong Yong, et al. Efficient condition code emulation for dynamic binary translation systems[C]//Proc of the 3rd Int Symp on Computer Engineering and Intelligent Communications (ISCEIC 2022). Bellingham, WA: SPIE, 2023, 12462: 421−432

[52] Liu Yuping, Hong Dingyong, Wu J J, et al. Exploiting SIMD asymmetry in ARM-to-x86 dynamic binary translation[J]. ACM Transactions on Architecture and Code Optimization, 2019, 16(1): 1−24

[53] Shuja J, Gani A, Ko K, et al. SIMDOM: A framework for SIMD instruction translation and offloading in heterogeneous mobile architectures[J]. Transactions on Emerging Telecommunications Technologies, 2018, 29(4): e3174

[54] Hallou N, Rohou E, Clauss P, et al. Dynamic re-vectorization of binary code[C]//Proc of 2015 Int Conf on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS). Piscataway, NJ: IEEE, 2015: 228−237

[55] Hallou N, Rohou E, Clauss P. Runtime vectorization transformations of binary code[J]. International Journal of Parallel Programming, 2017, 45: 1536−1565

[56] Wu Jin, Dong Jian, Fang Ruili, et al. Effective exploitation of SIMD resources in cross-ISA virtualization[C]//Proc of the 17th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments. New York: ACM, 2021: 84−97

[57] Liu Yuping, Hong Dingyong, Wu J J, et al. Exploiting asymmetric SIMD register configurations in ARM-to-x86 dynamic binary

translation[C]//Proc of the 26th Int Conf on Parallel Architectures and Compilation Techniques (PACT). Piscataway, NJ: IEEE, 2017: 343−355

[58] Lin C M, Fu Shengyu, Hong Dingyong, et al. Exploiting vector processing in dynamic binary translation[C]//Proc of the 48th Int Conf on Parallel Processing. New York: ACM, 2019: 1−10

[59] Fu Shengyu, Hong Dingyong, Liu Yuping, et al. Dynamic translation of structured loads/stores and register mapping for architectures with SIMD extensions[C]//Proc of the 18th ACM SIGPLAN/SIGBED Conf on Languages, Compilers, and Tools for Embedded Systems. New York: ACM, 2017: 31−40

[60] Fu Shengyu, Hong Dingyong, Liu Yuping, et al. Optimizing data permutations in structured loads/stores translation and SIMD register mapping for a cross-ISA dynamic binary translator[J]. Journal of Systems Architecture, 2019, 98: 173−190

[61] Hong Dingyong, Liu Yuping, Fu Shengyu, et al. Improving SIMD parallelism via dynamic binary translation[J]. ACM Transactions on Embedded Computing Systems, 2018, 17(3): 1−27

[62] Li Jianhui, Zhang Qi, Xu Shu, et al. Optimizing dynamic binary translation for SIMD instructions[C]//Proc of Int Symp on Code Generation and Optimization (CGO'06). Piscataway, NJ: IEEE, 2006: 12−280

[63] Clark N, Hormati A, Yehia S, et al. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping[C]//Proc of the 13th Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2007: 216−227

[64] Zhou Ruoyu, Wort G, Erdős M, et al. The janus triad: Exploiting parallelism through dynamic binary modification[C]//Proc of the 15th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments. New York: ACM, 2019: 88−100

[65] Nakamura T, Miki S, Oikawa S. Automatic vectorization by runtime binary translation[C]//Proc of the 2nd Int Conf on networking and computing. Piscataway, NJ: IEEE, 2011: 87−94

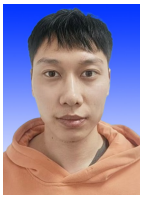
[66] Jordan M G, Knorst T, Vicenzi J, et al. Boosting SIMD benefits through a run-time and energy efficient DLP detection[C]//Proc of 2019 Design, Automation & Test in Europe Conf & Exhibition (DATE). Piscataway, NJ: IEEE, 2019: 722−727

[67] Shen B Y, Chen J Y, Hsu W C, et al. LLBT: An LLVM-based static binary translator[C]//Proc of the 2012 Int Conf on Compilers, Architectures and Synthesis for Embedded Systems. New York ACM, 2012: 51−60

[68] Hong Dingyong, Hsu C C, Yew P C, et al. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores[C]//Proc of the 10th Int Symp on Code Generation and Optimization. New York: ACM, 2012: 104−113

[69] Xie Haibin, Wu Chenggang, Cui Huimin, et al. x86 floating point stack processing in binary translation[J]. Journal of Computer Research and Development, 2007, 44(11): 1946−1954 (in Chinese)
(谢海斌, 武成岗, 崔慧敏, 等. 二进制翻译中的 x86 浮点栈处理[J]. 计算机研究与发展, 2007, 44(11): 1946−1954)

Lai Yuanming, born in 1991. PhD candidate. His main research interests include binary translation, computer systems security, and code obfuscation.
赖远明, 1991 年生. 博士研究生. 主要研究方向为二进制翻译、计算机系统安全、代码混淆.



Li Yalong, born in 2000. Master candidate. His main research interests include binary translation optimization and RISC-V architecture. (li2542369686@163.com)
李亚龙, 2000 年生. 硕士研究生. 主要研究方向为二进制翻译优化、RISC-V 体系结构.

Hu Hanzhi, born in 1999. Master candidate. His main research interests include binary translator optimization and compiler optimization. (huhanzhi22s@ict.ac.cn)
胡瀚之, 1999 年生. 硕士研究生. 主要研究方向为二进制翻译优化、编译优化.

Xie Mengyao, born in 1992. PhD, associate professor, master supervisor. Her main research interests include computer system architecture and system security.
谢梦瑶, 1992 年生. 博士, 副研究员, 硕士生导师. 主要研究方向为计算机系统结构、系统安全.

Wang Zhe, born in 1990. PhD, associate professor, master supervisor. Member of CCF. His main research interests include computer systems security, operating systems, and computer architecture.
王　喆, 1990 年生. 博士, 副研究员, 硕士生导师. CCF 会员. 主要研究方向为计算机系统安全、操作系统、计算机体系结构.

Wu Chenggang, born in 1969. PhD, professor, PhD supervisor. Distinguished member of CCF. His main research interests include binary translation, compiler optimization, and computer systems security. (wucg@ict.ac.cn)
武成岗, 1969 年生. 博士, 研究员, 博士生导师. CCF 杰出会员. 主要研究方向为二进制翻译, 编译优化、计算机系统安全.