软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(6):2346−2369 [doi: 10.13328/j.cnki.jos.007617] [CSTR: 32375.14.jos.007617]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向 RISC-V 指令集多样性的兼容性感知多层级构建方法*

屈 晟[1,2], 苏运强[1], 林哲远[3], 燕言言[3], 冯 洋[3], 赵 琛[1]

[1](中国科学院 软件研究所 智能软件研究中心, 北京 100190)

[2](中国科学院大学, 北京 100049)

[3](计算机软件新技术全国重点实验室 (南京大学), 江苏 南京 210023)

通信作者: 赵琛, E-mail: zhaochen@iscas.ac.cn

摘 要: RISC-V 指令集架构凭借其开放性与模块化设计, 推动了芯片架构实现创新与定制化, 但与此同时也引发了严重的软件生态碎片化问题. 传统的跨平台软件构建机制 (如现场编译、IFUNC 与 Multilib) 在 RISC-V 生态中面临兼容性差、维护成本高与优化粒度不足等显著挑战, 亟需新的解决方案. 提出了一种面向 RISC-V 平台的兼容性感知多层级编译方法——RuyiBuild 工具链, 以 LLVM IR 为中间表示, 结合非侵入式编译流程拦截机制, 实现了对现有软件构建系统的透明适配, 使构建生成的操作系统软件包既具备面向不同 RISC-V 扩展指令集的兼容性, 又可以面向最终运行平台进行基于扩展指令集的自适应优化, 有效解决了 RISC-V 指令集多样性背景下, 多指令扩展组合与平台差异下二进制高性能优化与广泛兼容性的双目标统筹实现问题. RuyiBuild 工具链围绕 LLVM IR 的提取与部署、转换与优化设计了一套完整的跨平台软件分发与精细化优化框架, 主要包含 4 个核心机制: 透明化双路径编译与 LLVM IR 提取机制、动态链接库 LLVM IR 合并与链接转换机制、LLVM IR 文件部署与发行版 RPM 自动化集成机制以及客户端与云端 LLVM IR 动态转换与资源自适应调度机制. 在技术实现方面, RuyiBuild 工具链通过封装编译工具与系统命令, 在不修改源码与构建系统的前提下实现 LLVM IR 的全路径提取与分发, 并在部署阶段支持 LLVM IR 与传统二进制的同步发布. 同时, 为支持多种目标设备和微架构的性能优化, RuyiBuild 支持客户端资源感知的延迟转换、云端面向多架构的适配与 LLVM IR 动态转换. 实验结果表明, RuyiBuild 能够面向多种 RISC-V 指令集扩展与微架构组合实现对目标程序的部署, 并在性能、兼容性、构建开销与部署复杂度方面取得较好平衡, 解决了硬件兼容性与硬件性能充分发挥之间的矛盾. 为 RISC-V 生态中的软件构建、部署与适配提供了新的解决思路, 具备良好的学术价值与实践推广潜力.

关键词: RISC-V; LLVM IR; 跨平台编译; 软件生态碎片化; 多架构适配; 构建系统透明性

中图法分类号: TP311

中文引用格式: 屈晟, 苏运强, 林哲远, 燕言言, 冯洋, 赵琛. 面向RISC-V指令集多样性的兼容性感知多层级构建方法. 软件学报, 2026, 37(6): 2346–2369. http://www.jos.org.cn/1000-9825/7617.htm

英文引用格式: Qu S, Su YQ, Lin ZY, Yan YY, Feng Y, Zhao C. Compatibility-aware Multi-level Build Method for RISC-V Instruction Set Diversity. Ruan Jian Xue Bao/Journal of Software, 2026, 37(6): 2346–2369 (in Chinese). http://www.jos.org.cn/1000-9825/7617.htm

Compatibility-aware Multi-level Build Method for RISC-V Instruction Set Diversity

QU Sheng[1,2], SU Yun-Qiang[1], LIN Zhe-Yuan[3], YAN Yan-Yan[3], FENG Yang[3], ZHAO Chen[1]

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

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

* 本文由“RISC-V 与人工智能系统软件前沿进展”专题特约编辑武延军研究员、谢涛教授、侯锐研究员、宋威研究员、邢明杰高级工程师推荐.
收稿时间: 2025-09-08; 修改时间: 2025-10-20, 2025-12-08; 采用时间: 2025-12-17; jos 在线出版时间: 2025-12-26
CNKI 网络首发时间: 2026-04-02

[3](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: The RISC-V instruction set architecture, characterized by its openness and modular design, promotes innovation and customization in processor architecture, while simultaneously introducing severe software ecosystem fragmentation. Traditional cross-platform software build mechanisms, such as on-site compilation, IFUNC, and Multilib, encounter significant challenges in the RISC-V ecosystem, including limited compatibility, high maintenance overhead, and insufficient optimization granularity, which highlights the need for new solutions. To address these issues, this study proposes a compatibility-aware multi-level compilation method for the RISC-V platform—RuyiBuild toolchain. By adopting LLVM IR as the intermediate representation and integrating a non-intrusive compilation interception mechanism, transparent adaptation to existing software build systems is achieved. As a result, the generated operating system software packages simultaneously support compatibility across heterogeneous combinations of RISC-V extension instruction sets and adaptive, extension-aware optimizations tailored to target execution platforms. This approach systematically resolves the dual-objective challenge of achieving both high-performance binary optimization and broad compatibility under diverse instruction set extensions and platform variations inherent in the RISC-V ecosystem. Centered on the extraction, deployment, transformation, and optimization of LLVM IR, RuyiBuild establishes a comprehensive framework for cross-platform software distribution and fine-grained optimization. The framework consists of four core mechanisms: a transparent dual-path compilation and LLVM IR extraction mechanism; a dynamic library LLVM IR aggregation and link-time transformation mechanism; an LLVM IR deployment and automated RPM integration mechanism; a client-cloud collaborative LLVM IR dynamic transformation and resource-adaptive scheduling mechanism. From an implementation perspective, the RuyiBuild toolchain encapsulates compilation tools and system commands, enabling full-path LLVM IR extraction and distribution without modifying source code or existing build systems. During deployment, synchronized distribution of LLVM IR and traditional binaries is supported. Furthermore, to facilitate performance optimization across various target devices and microarchitectures, client-side resource-aware deferred transformation and cloud-side multi-architecture adaptation with dynamic LLVM IR transformation are provided. Experimental results show that RuyiBuild supports deployment across a wide range of RISC-V instruction set extensions and microarchitecture combinations, while achieving a favorable balance among performance, compatibility, build overhead, and deployment complexity. Consequently, this study provides a novel and effective solution for software build, deployment, and adaptation in the RISC-V ecosystem, offering both academic value and practical potential.

Key words: RISC-V; LLVM IR; cross-platform compilation; software ecosystem fragmentation; multi-architecture adaptation; build system transparency

1 引 言

近年来, 随着计算领域多样化应用需求的不断涌现, 传统的指令集架构 (如 x86 架构与 ARM 架构) 逐渐暴露出其封闭性、扩展性不足、授权费用高昂等局限性问题[1,2]. 作为应对这一挑战的全新方案, RISC-V 指令集架构以其开放性、精简性和可扩展性的独特优势, 迅速获得学术界与产业界的广泛关注与认可[3]. RISC-V 允许芯片厂商根据特定性能需求和应用场景需求, 自由定制指令集扩展与微架构, 从而推动了芯片设计的灵活性与创新性[4,5].

同时为了满足标准化、大规模部署以及用户快速使用的需要, 如 openEuler、Debian、Ubuntu 等预编译二进制 Linux 操作系统得到了广泛的使用, 为与纯理论含义上的操作系统内核进行区分, 一般称之为操作系统发行版或直接简称为发行版. 这些操作系统均是以软件包为单位进行组织. 如何方便、快速、标准化地构建软件包, 并且充分发挥硬件的性能就成为一个主要的研究课题. 如果使用贴近特定硬件特性的方式构建软件包, 那么将会带来兼容性的巨大挑战; 反之, 如果使用通用硬件的特性, 那么又会导致无法发挥硬件最强性能, 这一般被称为碎片化. 当前各主流发行版普遍选择优先保障兼容性.

然而, 指令集扩展的高度自由化与微架构的多样化, 也给 RISC-V 生态系统带来更加严重的软件生态碎片化问题[6]. 不同厂商实施差异化的指令集扩展与微架构优化, 使得软件跨平台的兼容性与性能优化问题变得尤为突出. 传统的软件分发模式 (如预编译二进制文件) 在 RISC-V 生态中面临难以逾越的困难, 若需覆盖所有可能的指令集扩展组合与微架构, 软件包数量将呈现指数级增长, 严重提高了软件分发与维护成本; 而源码级的现场编译 (on-device compilation) 则受限于终端设备资源不足与用户体验要求, 难以推广实施[7].

针对上述问题, 目前学术界与产业界给出的解决方案主要基于两种思想: 一种是通过修改动态库的源代码, 从函数或动态库级别提供面向不同指令集优化的版本, 在软件运行时, 可以根据当前硬件的特性选择适合的版本; 另

外一种是结合最终硬件特点, 直接从源码编译.

间接函数支持 (indirect function support, IFUNC) 与多库支持 (Multilib)[8,9]是第 1 种思路的两种具体实现. 每增加一种新的指令集组合或微架构实现, 均需要对软件的源代码进行修改, 这样虽然能提升特定架构上的性能, 但也显著增加了维护的开销且降低了方法的泛用性. 例如, 在 RISC-V 优化 strlen 函数对 glibc 进行的修改中[10]需要显式提供一个常规实现与一个 Zbb 优化实现, 并通过 hwprobe() 检测是否支持 Zbb, 动态选择优化版本以避免性能退化. 较差的可维护性极大地限制了这类方法的广泛使用. 一般来讲, 这类方法仅应用于少数重点优化的库 (如 glibc), 以及只应用于指令集的几个主要版本 (如 x86-64-v2、x86-64-v3、ARMv9、RISC-V RVA23). 处于这些大版本中间状态的大量指令集组合无法得到支持, 一些没有包含在这些主要版本中的额外指令集扩展也无法得到支持, 并且该类方法无法对每个指令集扩展的启用情况进行细粒度控制, 只能在代码编写时对目标平台做出固定选择, 缺乏编译器或运行器的动态适配能力, 难以针对特定微架构进行更加精细的动态优化[11]. 除此之外, 可以想象随着指令集的持续发展, 为了使用新指令集, 需要持续对这些代码进行修改, 软件的维护负担也将越来越重.

类似 Gentoo 的源代码发行版是第 2 种思路的实现. Gentoo 以源代码而非二进制进行分发, 进而在终端用户侧进行源代码向二进制的编译. 该方案可以结合最终硬件平台特性充分发挥编译器的优化潜能, 但存在两类问题: 一是要求用户具备处理各种复杂编译问题的能力, 如编译依赖的缺失、版本兼容问题等; 二是在源码编译时不仅耗费大量硬件资源 (包括 CPU、内存), 还需要从网络上下载编译依赖, 导致总体耗时较长, 例如, 在 openEuler 24.03 LTS SP2 平台上针对 RISC-V64 架构构建 OpenCV 软件包的完整过程耗时长达 15 h[12].

为此本文提出构建操作系统发行版的一种思路, 即将从源代码直接编译生成二进制操作系统的主流方式, 转化为结合编译过程环境依赖差异的两阶段构建流程. 第 1 阶段, 从源代码编译到一种中间表示, 例如 LLVM IR (LLVM intermediate representation), 此过程需要处理复杂的依赖关系, 例如编译器版本、头文件、本地环境变量、同软件包中各文件之间的关系等, 环境的变化可能会造成编译结果的变化, 乃至编译失败, 同时这一阶段所需要时间也是整个编译过程中最长的; 这一阶段可以由专业的操作系统开发团队 (如 openEuler 开发团队) 在云环境中进行. 第 2 阶段, 从中间表示生成最后的可执行程序或共享库等, 既可以生成兼容性最好的可执行程序, 也可以根据具体的硬件指令集和微架构实现进行定向优化, 此过程可以在无需编译依赖的环境进行, 且资源消耗较少, 在最终用户设备上进行. 通过实现以上方案, 既能够摆脱源码编译对于最终用户技能的高要求, 同时也能够生成满足兼容性要求的可执行程序或共享库, 使硬件性能的发挥接近从源码编译的效果.

LLVM 以及 Clang 编译器支持持久化存储的中间表示 (LLVM IR), 随着 LLVM IR 的逐渐标准化和逐渐稳定, 其为解决这个问题带来了希望. 但是大部分现有软件使用的构建系统 (如 Autotools 和 CMake) 均是按照文件为单位, 首先将 C 源码编译成目标机器的目标格式 (如 Linux 系统上的 ELF 格式), 再将这些单独的目标文件链接成为一个可执行程序或动态库. 这与我们提到的两个阶段方法完全不同, 因此需要一种创新性的方案在不需要修改现有软件代码及构建系统的基础上使用新的方式构建和分发软件.

针对上述背景与挑战, 本文提出了一种软件构建与分发工具链 RuyiBuild. RuyiBuild 工具链以统一的 LLVM IR 文件作为跨平台软件分发的中间表示格式, 其可以转换为任意扩展指令集组合和微架构优化的二进制代码, 这样既避免了用户从源码进行构建, 也避免了按 Multilib 要求修改软件的构建系统, 更无需如 IFUNC 般对 C 代码进行修改, 并且具备了 Multilib 和 IFUNC 所不具备的细粒度架构和微架构支持. 此外, RuyiBuild 工具链提出一种基于动态模板的、面向 RISC-V 指令集多样性的兼容性感知多层级构建方法, 能够针对目标设备的具体指令集扩展组合与微架构特征进行动态检测与精细化优化, 极大提高了软件在不同设备上的实际性能表现.

如表 1 所示, RuyiBuild 工具链通过封装程序两次调用 clang、install 等工具, 从而实现无需修改目标软件源码或构建系统, 保持了软件生态的完整性与稳定性; 通过第 1 次调用, 构建满足最低兼容性要求的二进制代码以满足对兼容性的要求; 通过第 2 次调用, 构建满足最优性能的 LLVM IR, 继而结合目标平台的指令集和微架构特性, 实现贴近硬件特性的细粒度动态转换与精确优化; 此外, RuyiBuild 工具链与主流 Linux 发行版软件管理工具 (如 rpm、dnf) 无缝衔接, 显著降低了部署复杂性, 提升了用户体验与生态适用性.

表 1 当前构建方案对比

对比方案	无需修改源码	精细优化	不要求用户技术水平	易与rpm/dnf等集成
Multilib	否	否	是	否
IFUNC	否 (且难度大)	否	是	是
客户端源码编译	是	是	否	否
服务端源码编译并构造多仓库	是	否	半 (需要用户选择正确软件仓库)	是
RuyiBuild	是	是	是	是

Multilib 方法通过修改软件的构建系统, 如 Makefile, 将原来只编译一遍的动态库编译多遍, 此方法会进一步增加构建系统的复杂度与软件的维护代价, 因此, 只有少数关键基础库采用了此方法以提高兼容性, 如 glibc 使用 Multilib 方法实现了对 x86-64-v2、x86-64-v3 和 x86-64-v4 这 3 种指令集组合的支持. 另外, 使用此方法编译同一动态库的多个版本还需要同步修改 RPM (RPM package manager) 等软件包制作脚本, 带来更大的工作量.

IFUNC 方法的实现是所有方法中难度最大的, 首先需要编译器支持, 其次需要动态加载器支持, 同时还需要对被优化软件的源代码进行修改, 通过如#pragma 等方式标记需要特殊优化的函数, 提示编译器进行多版本编译.

客户端源码编译即使用手动或自动的方法, 在用户终端上进行软件编译. 因用户环境的不可控性, 势必存在不可控问题, 进而要求用户进行手动处理, 因此, 要求用户具备较高技术水平. 同时, 为支持用户手动介入, 难以充分利用发行版配套的自动化工具, 导致与 rpm/dnf 等包管理器集成困难, 所以只能局限于开发者小范围分发使用.

与 Multilib 和 IFUNC 方案相比, RuyiBuild 不需要对软件源码做任何修改, 并且提供了贴近硬件特性的细粒度控制; 与在客户端从源码编译相比, 降低了对用户技术水平的要求, 无需下载编译依赖, 也降低了对用户设备的性能要求; 与在服务端从源码编译相比, 减轻了操作系统发行版开发团队的工作量, 无需为无数种现有和潜在的指令集和微架构组合分别编译整个软件仓库.

通过本文提出的 RuyiBuild 工具链及其关键技术机制, 不仅有效解决了 RISC-V 平台软件生态中的碎片化问题, 还为 RISC-V 平台的规模化应用与推广提供了重要的技术支撑, 具有重要的学术和实践意义. 本文的主要研究贡献可以凝练总结为以下 4 点.

(1) 提出一种针对 RISC-V 平台指令集扩展与微架构多样性的跨平台软件构建与分发方案. 该方案基于统一的 LLVM IR 格式, 在不修改现有软件源码与构建系统的前提下, 实现软件跨平台的高效部署与精细优化, 有效解决现有方案 (IFUNC 与 Multilib) 所面临的软件生态碎片化难题, 并且达到类似从源码编译的性能.

(2) 提出并实现了基于动态模板的指令集扩展自适应与微架构精细化优化方法. 该方法能够自动检测目标设备的具体指令集扩展组合与微架构特征, 动态调整 LLVM IR 文件, 生成针对目标设备的精细优化二进制文件, 显著提高软件在具体目标设备上的性能表现.

(3) 设计并实现一套客户端与云端的双模式 LLVM IR 动态转换与资源自适应调度机制. 客户端动态转换机制通过实时监控设备资源使用情况, 实现转换过程资源自适应调度, 确保转换任务不会影响终端用户体验; 云端机制则通过批量化并行转换与自动化 RPM 仓库建设, 极大提高软件部署效率.

(4) 通过与 Linux 发行版 RPM 软件仓库的自动集成与构建机制, 实现 LLVM IR 软件包到常规 RPM 软件包的自动化批量转换, 用户可直接通过标准软件管理工具 (如 dnf) 安装, 极大降低软件部署复杂性, 提升用户体验, 并推动 RISC-V 生态系统的软件分发与部署效率.

本文第 2 节分析现有相关研究成果与技术方案的局限性. 第 3 节详细介绍 RuyiBuild 工具链的设计思想、整体结构与具体技术实现. 第 4 节通过实验与性能评估, 验证 RuyiBuild 工具链的有效性与先进性. 第 5 节对全文工作进行总结, 并对未来研究方向进行展望.

2 研究背景

为了更好地理解 RuyiBuild 工具链的设计动机与实现机制, 本节将介绍本文所依赖的关键技术基础, 包括

RISC-V 指令集结构特性、LLVM IR、传统软件构建系统构成与工作机制等. 这些基础知识为 RuyiBuild 工具链的构建、部署与转换能力提供了理论支撑与实现路径.

2.1 RISC-V 指令集架构

RISC-V[1]是近年来在学术界与工业界广泛受到关注的一种新型指令集架构 (instruction set architecture, ISA), 起源于 2010 年加州大学伯克利分校的项目. 其最显著特征在于开放性: RISC-V 的所有规范文档均对外公开发布, 且实现不受专利授权限制, 任何组织和个人均可自由构建符合该标准的处理器. 这一开放策略为处理器设计与创新提供了前所未有的自由度.

RISC-V 遵循精简指令集计算 (RISC) 理念, 强调指令结构简洁、编码规则统一、实现门槛低[13]. 在架构设计上, RISC-V 自诞生之初即采用高度模块化的扩展机制[2,14], 将指令集划分为基础集和若干可选扩展模块. 其中, RV32I 和 RV64I 分别为 32 位与 64 位基础整数指令集, 包含大约 40–50 条核心指令, 用于支撑基本的计算与控制流逻辑. 在此基础之上, RISC-V 通过添加字母后缀来引入标准扩展功能, 例如: M 扩展提供整数乘除指令; A 扩展提供原子操作指令; F/D 扩展提供单精度/双精度浮点指令; C 扩展提供 16 位压缩指令, 以减少代码体积. 多个扩展可以组合使用, 形成特定应用的 ISA 子集[15,16]. 例如, “RV64GC”通常表示 RV64I 基础指令集加上通用扩展集合 (IMAFD 等) 以及必要的控制状态寄存器和取指屏障扩展. 通过选择不同的扩展组合, RISC-V 可灵活覆盖从微控制器到高性能处理器的各种需求. 其规范将各扩展设计为彼此兼容, 避免冲突, 并且允许实现者根据需要增减扩展, 从而实现高度定制化.

然而, RISC-V 的这种模块化自由也带来碎片化和平台差异的挑战[17]. 不同芯片厂商根据自身需求选择的扩展组合可能互不一致, 从而造成二进制软件的通用性下降. 尤其是在需要发布二进制软件的领域 (如操作系统发行版), 过多的可选扩展会让应用程序难以确定“最低公分母”的 ISA 支持集. 此外, RISC-V 在不同应用平台上表现出不同的系统功能特性. 例如, 部分微控制器级实现不支持内存管理单元 (MMU) 与页表机制, 无法运行完整的 Linux 内核; 而高性能应用处理器则支持虚拟内存、向量计算、多级缓存等高级特性. 因此, 即便指令集一致, 系统软件在不同平台上的运行行为也可能存在较大差异.

RISC-V 的模块化设计在提升架构可定制性的同时, 也引发了软件构建、部署与分发过程中的兼容性挑战. 在 RISC-V 生态中实现软件的一次构建、多平台适配、动态转换部署, 需要构建工具具备足够的指令集感知能力、平台特征抽象能力与部署机制弹性. RISC-V 同时也给更多企业或其他单位进入 CPU 设计领域带来很多机会. 参与 CPU IP 设计的企事业单位从一两家 (x86 和 ARM), 扩展到数十家. 每家 CPU 设计单位设计的 CPU 往往有不同的微架构特性, 同样的二进制软件在不同的 RISC-V CPU 上可能会表现出巨大的性能差异[18]. 这也为软件的开发、编译和分发带来巨大挑战. 本文提出的 RuyiBuild 工具链, 正是在此背景下发展出的新型构建思路.

2.2 LLVM IR

LLVM IR 是 LLVM 编译器框架中的核心中间语言[19–21], 其在整个编译流程中承担着承上启下的枢纽作用. 作为一种面向静态单赋值 (static single assignment, SSA) 形式的中间语言, LLVM IR 能够精准地刻画程序的语义与控制流程, 为后续的优化与目标代码生成奠定了结构基础. 每个变量在其作用域内仅被赋值一次, 所有操作均通过带类型的中间寄存器 (通常以 % 符号开头) 完成. 这一机制极大地简化了编译器对数据依赖关系的分析, 也使得如值传播、死代码消除、常量折叠等优化操作更为直接可靠. LLVM IR 是一种强类型的低级类汇编语言. 它既抽象出通用指令系统 (如 add、sub、call、ret 等), 又保留了函数调用、内存模型、异常处理等高级语言语义. 同时, 它支持无限数量的虚拟寄存器, 避免物理寄存器资源受限所带来的瓶颈, 使得编译器在优化过程中拥有更大的调度空间[22].

当前 LLVM IR 一般用于编译器内部各组件之间的信息交换, 如 Clang、Rust、Flang 等编译器前端将高级语言转换为 LLVM IR, 然后通过编译器中端 (也可以是 opt 工具) 进行通用优化, 再通过编译器后端 (也可以是 llc 工具) 将 LLVM IR 转换为具体硬件架构的汇编代码. LLVM IR 也被用于 LLVM 的链接时优化 (LTO) 技术的实现. 使用 LTO 选项编译高级语言时, 如 clang -c -flto 时, 编译得到的结果是 LLVM IR 而不是真正的二进制目标代码 (如

Linux 系统上的 ELF 目标文件); 真正二进制目标代码的生成移到了链接时进行.

在跨平台能力方面, LLVM IR 具备天然的平台无关性[23,24]. 各类语言的函数调用约定、数据对齐规则等, 在 IR 层均以统一的形式表示, 便于后端针对不同目标架构生成对应汇编指令. IR 中的函数调用一般表现为 call 指令, 返回值则由 ret 指令处理, 参数传递通过显式声明完成. 这种统一抽象大大降低了编译器在前端与后端之间的适配成本. 在实际使用中, LLVM IR 既可以作为内存中的中间数据结构存在于编译器内部, 也可被序列化为二进制位码文件 (.bc) 或文本格式 (.ll) 文件. 这种二进制-文本双表示机制既有利于高效存储与传输, 也便于开发者调试与静态分析.

得益于其良好的抽象层次和平台独立性, LLVM IR 成为现代编译器体系结构中实现多语言前端+多架构后端解耦的核心桥梁[25,26]. 只需实现一个将新语言转换为 LLVM IR 的前端模块, 或将 LLVM IR 转换为新硬件架构指令的后端模块, 便可在现有基础上快速支持新场景, 极大提升编译器的可复用性和可扩展性. 在软件跨平台部署的实际应用中, LLVM IR 也展现出强大的能力[27,28]. 例如, 可将应用程序预编译为平台无关的 IR 文件, 部署至各终端后再由本地工具根据硬件平台生成本地机器码, 从而实现“一次构建, 多端适配”的部署模式. 对于像 RISC-V 这样指令集与微架构高度可定制的生态, LLVM IR 的这一特性尤为重要. LLVM IR 作为 LLVM 体系的核心中间表示语言, 既承载了语言语义的信息表达, 又支撑了跨平台优化与部署能力.

当前, LLVM IR 正处于快速发展与演进阶段, 导致在不同 LLVM 版本之间存在潜在兼容性问题, 限制了其在支持多版本工具链且快速迭代演进的操作系统发行版中的全面应用, 如 Fedora、Debian 等. 本文面向操作系统发行版的研发需求, 通过记录编译和链接阶段执行的命令, 将 LLVM 限制在同一大版本, 以确保可以成功将 LLVM IR 转换为目标代码.

2.3 目标代码适配技术

在指令集扩展与微架构多样性日益增强的背景下, 如何在保持软件可移植性的同时提升目标平台的运行效率, 成为构建系统设计的重要问题. 为此, 研究者和工程实践者提出两种主流机制: 间接函数支持 (IFUNC) 与多库支持 (Multilib). 二者分别在运行时与构建时提供了跨平台优化的路径, 在 x86、AArch64 等平台已有广泛应用, 在 RISC-V 等新兴架构中也逐步显示出其优势与局限性.

(1) IFUNC: 链接阶段的动态函数分发机制

IFUNC[29,30]是一种基于运行时符号分发的函数多实现机制, 它通过链接器和动态加载器的配合, 实现对函数符号的延迟解析. 开发者可为同一接口函数提供多个 ISA 优化版本并编译链接到同一个动态链接库文件, 同时注册一个解析器函数用于在程序加载时选择最适合的平台实现. IFUNC 的技术核心是将符号类型标记为 STT_GNU_IFUNC, 链接器保留相应重定位信息, 加载器在运行时调用解析函数获得最终函数地址, 填入 GOT 表中以支持高效跳转. 该机制允许开发者为同一个 API 提供多个 ISA 优化版本 (如基于不同 SIMD 扩展的 memcpy), 通过 cpu_features 或 getauxval() 等硬件探测方法在加载时做出最优选择. 它兼具运行时灵活性与调用开销最小化的优势, 并已被广泛应用于 glibc、OpenSSL、OpenBLAS 等系统库中用以适配如 AVX、NEON、RISC-V Vector 等指令扩展[31].

然而, IFUNC 机制仍面临以下挑战: 一方面, 它对编译器和动态链接器功能有较强依赖, 部分新兴平台 (如早期的 RISC-V 发行版) 对 STT_GNU_IFUNC 类型支持不完整; 另一方面, 其函数选择逻辑需开发者手动维护, 适配策略缺乏自动化与可扩展性, 限制了在大规模软件系统中的应用[2,5].

(2) Multilib: 构建阶段的多版本库并行布局

Multilib 是另一种面向多平台构建优化的机制, 它通过在构建阶段生成并部署多个库版本, 支持目标平台在链接时选择最合适的库变体. 每个变体针对不同的 ISA 组合或 ABI 配置 (如 32 位/64 位、硬/软浮点、压缩指令支持等) 进行独立构建与打包. 链接器在构建系统的辅助下根据目标参数选择匹配的目录与文件路径完成链接[29,32]. Multilib 在许多主流架构的发行版中已被广泛应用, 并通过如 GCC 的--with-multilib-list 配置参数得到工具链级支持. 研究者指出, Multilib 能够支持多个架构与 ABI 的并存构建, 满足面向通用平台的软件兼容需求, 是嵌入式与

通用 Linux 系统常见的部署策略之一[32–34]. 然而, Multilib 也存在天然的扩展性瓶颈. 在 RISC-V 这类指令集模块化程度极高的架构中, 可能的 ISA 扩展组合呈指数增长, 若逐一构建对应变体, 将导致构建与维护成本显著上升, 为所有扩展组合生成独立库的方式在资源有限或构建迭代频繁的场景中难以维系.

在 RISC-V 架构中, 传统的 IFUNC 和 Multilib 机制在支持生态兼容的同时, 也暴露出平台适配粒度粗、运行能力弱、部署开销高等问题. 为此, 构建系统亟需引入面向中间表示 (如 LLVM IR) 的统一构建中转层, 并结合平台特征感知机制实现自动化分发与精细化优化. 本文提出的 RuyiBuild 工具链, 正是在这一问题背景下设计的新型构建框架.

2.4 基于源码的性能优化技术

为了让二进制代码能贴近硬件指令集和微架构特性, 研究者和工程实践者开发出各种使用最贴近硬件的编译器选项制作的操作系统发行版, 如 Gentoo、FreeBSD Ports、MacPorts 等, 通过下载软件的源代码和相关编译依赖, 在用户设备上直接编译软件. 这样可以使用最贴近硬件的编译选项来编译软件, 得到尽可能好的性能. 这种方案需要用户自己编译软件, 并且可能需要自己解决编译过程中遇到的编译失败等问题, 对用户的技术水平提出了相当高的要求.

另外一种是基于源码的方案, 由用户从发行版的源码包 (如 openEuler 的 src.rpm) 使用不同的选项构建二进制, 并自己内部使用. 这种方案存在的问题包括: 对用户的能力提出了较高的要求; 因用户编译器、系统配置可能的差异, 无法保证二进制结果的一致性; 因源码包的复杂性, 用户设置的优化选项可能无法真正被使用.

3 RuyiBuild 工具链的技术设计与实现

RuyiBuild 工具链的总体流程如图 1 所示, 核心思想是构建透明双路径编译流程, 在不改动目标软件源码与原始构建系统的前提下, 化解软件兼容性和性能优化难题. 考虑到 Linux 发行版 (如 openEuler、Debian 等) 主流的 RPM、Deb 二进制包分发模式, 为避免因新增复杂度难以被接受, RuyiBuild 设计了完善的 LLVM IR 文件部署机制与 RPM 包自动化集成方案, 用户终端根据实际情况自主选择 RPM 包来源. 在软件构建阶段, 借助透明化包装技术同时产出传统 ELF 文件与平台无关的 LLVM IR 文件, 保障 LLVM IR 文件部署时与传统 ELF 文件结构一致, 助力终端与云端统一部署更新, 适配 Linux 发行版生态.

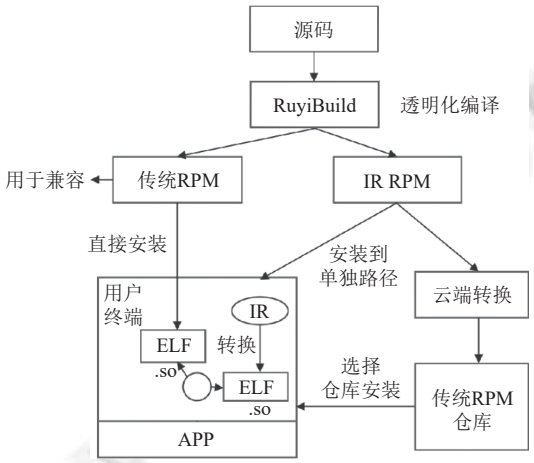

图 1 RuyiBuild 工具链总体路径

与此同时, 为应对 LLVM IR 在不同版本间兼容性不足的问题, RuyiBuild 设计了多层次的兼容性保障机制. RuyiBuild 在生成 LLVM IR 时记录完整的编译与链接命令及所用 LLVM 版本, 以确保后续转换过程版本一致; 在编译阶段采用双路径编译与透明包装机制, 确保 LLVM IR 文件与传统 ELF 文件路径结构严格对应; 在 IR 转换阶

段引入动态模板机制, 依据目标设备自动生成匹配的指令集与微架构模板. 通过这 3 种机制, RuyiBuild 实现了 LLVM IR 在终端与云端之间的稳定迁移与再构建. 此外, 作为保底, 如果存在 LLVM IR 不兼容问题, 则不会生成新的动态库, 使用传统方法编译的动态库依然可以使用.

3.1 透明化双路径编译与 LLVM IR 提取机制的设计与实现原理

LLVM IR 技术在操作系统发行版级别暂时无法广泛使用的一个主要原因是, 现有广泛使用的软件构建系统, 如 GNU Autotools、CMake 等均假定软件使用传统的编译方式, 使用 LLVM IR 技术可能需要对现有软件的组织方式进行重大改造, 其需要极大的软件开发工程量, 也会为软件增加非常多的复杂性. 这在工程实践中是不可接受的.

RuyiBuild 工具链的核心技术思想是利用 LLVM 中间表示 (LLVM IR) 在不修改目标软件源码与原始构建系统的情况下, 创建透明的双路径编译流程, 以解决 RISC-V 架构指令集碎片化问题所带来的软件兼容性与性能优化难题.

如图 1 所示, 通过 RuyiBuild 工具链, 两次调用 clang 等工具, 将源代码分别编译成为真实目标代码文件 (对于 RISC-V 和 Linux 的组合来讲为 ELF) 和 LLVM IR 文件; 之后, 将真实目标代码文件打包进入传统 RPM, 该 RPM 可被用户直接安装使用; 之后, 将 LLVM IR 文件以类似的文件夹结构打包进入 IR RPM. 对于 IR RPM 软件包, 既可以被直接安装到用户设备, 并通过使用相同版本 LLVM 转换为目标文件; 也可以在云端服务器使用相同版本的 LLVM 与编译命令, 依据目标平台支持的指令集, 转换为传统 RPM 文件, 并通过进一步构建 RPM 软件仓库进行分发.

如图 1 所示, 在用户终端内部, 以动态链接库为例说明 LLVM IR 文件的使用方式. 在 IR RPM 包未安装或尚未将 LLVM IR 转换为对应动态链接库之前, 用户程序默认使用传统 RPM 软件包提供的动态链接库, 一般其位于系统的/usr/lib64 文件夹下. 在 IR RPM 软件包安装后, 且 LLVM IR 文件成功转换成为动态链接库之后, 可以通过预设的环境与配置文件要求应用软件优先使用转换得到的动态链接库, 再次使用这些应用程序时即可实现性能提升. 类似地, 也可以提供优化的可执行程序, 并通过 PATH 环境变量等方式, 使用户优先使用优化过的可执行程序.

为了实现这一目标, RuyiBuild 以操作系统环境变量和编译工具包装 (wrapper) 脚本为基础, 精心设计了一套非侵入式的编译过程拦截机制, 如图 2 所示. 该机制通过环境变量 PATH 的动态设置, 覆盖标准构建工具 (包括但不限于 clang、clang++、ar、ln、install 等), 当构建系统调用这些标准工具时, 实际上调用的是 RuyiBuild 提供的同名包装脚本. 包装脚本的核心实现原理是对原始编译命令进行两次调用: 第 1 次调用完全保持原命令不变, 以生成传统的 ELF 目标文件或动态库, 确保原始构建系统的正常运行与兼容性; 第 2 次调用则在原始编译命令基础上追加生成 LLVM IR 的编译选项, 以生成对应的 LLVM IR bitcode 文件.

在具体实现中, 我们将原始编译命令中产生的路径信息, 特别是输出文件及其依赖的辅助文件的路径进行统一的抽象与重构, 以减少文件冲突与混乱. 系统通过识别路径特征, 对路径的空间位置进行语义分类, 并将其映射到用户界定的统一命名空间中. 相对路径在该命名空间中依据工作目录层级实现语义拓展, 而绝对路径则通过直接进行输出路径映射保持其编译上下文不变, 实现了传统编译产物与 IR 表征之间的严格对应, 使 IR 生成过程具有可追溯性与构建语义完整性, 保障了跨层编译分析流程的稳定性与一致性.

此外, RuyiBuild 工具链包装脚本还需要保证对原始构建系统的透明化, 尤其是在日志处理方面. 为了避免干扰原始构建系统的标准输出与错误流, 包装脚本在生成 LLVM IR 文件的过程中, 会将详细的日志信息统一重定向到单独的日志文件中. 若 LLVM IR 编译过程中发生错误, 包装脚本会自动删除之前生成的传统 ELF 文件, 并立即终止编译, 以便原始构建系统及时捕捉到错误信息并停止构建. 这种机制确保了 RuyiBuild 对原始构建系统的完全透明性与高度兼容性.

同时, 考虑实际软件开发中可能存在复杂的静态库构建场景, 即动态库的构建过程可能依赖于静态库中间产物, RuyiBuild 进一步对传统归档工具 ar 进行了包装扩展. 当构建系统调用 ar 工具生成传统静态库文件时, 包装脚本自动地在 LLVM IR 路径下进行对应的归档操作, 生成.a 格式的 LLVM IR 静态库文件. 这种扩展设计确保了 RuyiBuild 能够全面支持各种复杂构建场景, 而无需开发人员修改任何构建脚本或源码.

如图 2 所示, 为了保持软件现有的构建系统不做改动, RuyiBuild 包装了 clang、ld、install 等命令. 软件的构建系统首先调用 RuyiBuild 提供的 clang, 将源码文件分别编译成目标文件和 LLVM IR bitcode 文件. 随后软件的

构建系统再次调用 RuyiBuild 提供的 clang 命令对文件进行链接操作: 对于目标文件, clang 正常调用 ld 命令, 将目标文件链接成为真正的动态链接库; 对于 LLVM IR bitcode 文件, RuyiBuild 调用 llvm-link 命令, 将多个 LLVM IR bitcode 文件合并成为同一个 LLVM IR bitcode 文件, 并记录 clang 生成真正的动态链接库文件时使用的命令与使用的 LLVM 版本.

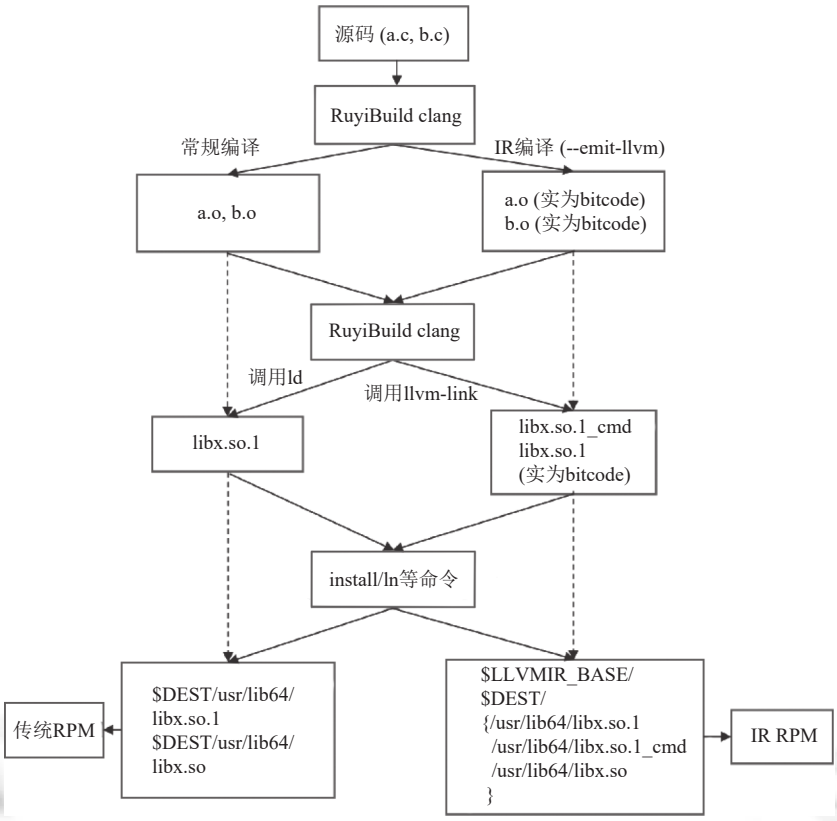

图 2 RuyiBuild 透明化编译方法

随后, 如图 2 所示, 软件的构建系统调用 install 或 ln 等命令, 将生成文件安装到某个临时文件夹: 对于目标文件, 这个文件夹由用户 (或 RPM SPEC 文件) 指定; 对于 LLVM IR bitcode 文件, 则被安装到 LLVM IR 输出文件夹中相对应的路径. 在软件的构建系统将所需文件分别按照 Linux 系统的规范安装到两个临时文件夹后, 可将这两个文件夹打包压缩为两个 RPM 软件包, 其中 LLVM IR 软件包文件将被从/usr/lib64/下移动到/usr/lib64/llvmir/目录下, 以避免文件冲突.

3.2 动态链接库 LLVM IR 合并与链接转换机制的实现原理

传统上, 程序或动态库等链接是在软件编译时在同一环境中进行的, 这样链接就可以获取和使用编译环境的很多信息, 这些信息包括: 参与链接的文件 (如编译产生的目标文件, 链接器脚本等), 链接的选项 (如优化级别、动态库版本信息等). 将这些信息带入后处理阶段, 将非常复杂. 与单个目标文件的编译不同, 动态链接库的构建通常涉及多个目标文件的链接过程. 为了支持这种复杂的链接场景, RuyiBuild 专门设计了一套基于 LLVM IR 文件合并与转换的技术机制, 如图 2 所示. 具体地, 当包装脚本识别出编译选项中包含动态链接标志-shared 时, 会首先调用原生 clang 命令, 生成传统的 ELF 格式动态链接库文件. 这一操作确保了原始构建系统的兼容性与有效性. 随后, RuyiBuild 调用 LLVM 工具链中的 llvm-link 工具, 将所有涉及链接的独立 LLVM IR bitcode 文件合并为单一的统一 LLVM IR 模块文件. 这种统一的模块化 IR 文件为后续跨平台适配与微架构优化提供了统一的基础表示.

为确保从 LLVM IR 到最终 ELF 格式转换过程的可复现性与准确性, 包装脚本自动记录原始的 clang 动态链接命令到一个专门生成的_cmd 文件中. 随后, RuyiBuild 包装脚本进一步修改该_cmd 文件, 将原始命令中所有的.o 目标文件引用统一替换为刚合并生成的单一 LLVM IR 文件, 并显式指定使用 LLVM 工具链中的 lld 链接器 (通过添加-fuse-ld=lld 选项). 这一设计选择源于传统 GNU 链接器 (ld) 无法直接识别 LLVM IR 文件格式的问题.

此外, 针对动态链接过程中可能涉及的符号版本控制脚本 (如--version-script 选项指定的版本控制脚本), 包装脚本自动复制该脚本文件至 LLVM IR 输出路径下, 并统一文件命名格式, 以确保转换过程中符号版本控制的一致性与精确性. 这种自动化的脚本处理机制进一步保障了 RuyiBuild 的完整性与易用性.

3.3 LLVM IR 文件部署与发行版 RPM 自动化集成机制的设计与实现

当前 Linux 上的主要软件分发方式为各大 Linux 发行版, 包括 openEuler、Debian、Fedora、Ubuntu 等. 它们分别使用 RPM 或 Deb 等二进制软件包格式. 如果不能与这些发行版进行紧密的集成, 将很大程度上限制本方法的广泛使用. 而当前主要发行版的软件打包方式往往非常复杂, 如果在这种复杂性上过分增加新的复杂度, 很可能无法为各发行版所接受. 为了有效解决 RISC-V 架构生态系统中软件分发场景下的复杂性与碎片化问题, RuyiBuild 工具链提出了一套完善的 LLVM IR 文件部署机制与发行版 RPM 软件包自动化集成方案. 该机制旨在实现对 LLVM IR 文件与传统 ELF 文件在系统部署阶段的结构一致性管理, 同时自动生成符合主流 Linux 发行版规范的软件包, 以便于终端设备与云端环境的统一部署与更新.

具体而言, 在软件构建阶段, RuyiBuild 工具链通过透明化包装技术, 同时生成传统 ELF 格式文件和与平台无关的 LLVM IR 文件, 这些 LLVM IR 文件统一存储于用户指定的环境变量 LLVMIR_BASEDIR 路径下. 为了确保 LLVM IR 文件在软件安装与部署环节中的结构与传统 ELF 文件保持一致, RuyiBuild 对系统常用的部署相关工具 (如 install、ln) 进行了透明的包装扩展, 安装过程如附录 A.1 所示.

通过自动化的文件安装与软链接镜像机制, RuyiBuild 工具链确保了 LLVM IR 文件与传统 ELF 文件结构严格对应, 从而极大方便了后续的跨平台转换与统一管理.

在发行版软件包自动化生成与 RPM 集成方面, RuyiBuild 工具链提出了一套自动扫描与 RPM 软件包生成机制. 具体实现中, 在整体软件构建过程完成后, RuyiBuild 工具链自动执行扫描命令, 如附录 A.2 所示, 以自动化方式查找安装根目录 (BUILDROOT) 下所有生成的 LLVM IR 文件及其辅助文件.

扫描完成后, RuyiBuild 工具链自动生成对应的 RPM 软件包规范文件 (RPM SPEC 文件), 并调用 RPM 构建工具, 自动生成包含这些 LLVM IR 文件的 RPM 软件包 (如 example-llvmir.rpm). 在 RPM SPEC 文件自动化生成时, RuyiBuild 会自动添加必要的 RPM 宏定义与安装脚本, 以确保 RPM 软件包安装后系统内同时部署 LLVM IR 文件与传统 ELF 文件, 如附录 A.3 所示.

用户安装该 RPM 软件包之后, 系统将同时拥有传统的 ELF 格式文件与 LLVM IR 文件, 分别位于标准系统路径 (如/usr/lib64) 与专用 LLVM IR 路径 (如/usr/lib64/llvmir) 下. 这种自动化 RPM 集成机制极大简化了软件发行版的维护成本, 便于用户与开发人员统一维护和升级.

此外, RuyiBuild 还提供了 RPM 安装后触发 LLVM IR 转换的自动化机制. RPM 安装完成后, 会自动调用后台服务进程, 触发后续客户端设备或云端环境中的转换任务, 以确保安装的软件在目标设备上经过精细化优化与适配. 这种机制进一步确保了 RuyiBuild 工具链从构建、部署到最终优化部署的一体化、自动化流程. 通过上述详细的自动化文件部署机制与 RPM 发行版集成流程, RuyiBuild 工具链实现了对 LLVM IR 文件与传统 ELF 文件的统一结构管理与自动化软件包集成, 显著简化了跨平台软件部署与维护过程, 极大提高了 RISC-V 生态系统的软件发行与部署效率.

3.4 客户端与云端 LLVM IR 动态转换与资源自适应调度机制的设计与实现

将 LLVM IR 转换为最终的二进制文件, 需要耗费大量的计算资源, 主要包括 CPU 和内存. 而用户终端设备, 往往性能并不充足, 同时对于响应时间、续航能力、噪声控制等方面有着苛刻要求. 为进一步提高 RISC-V 生态

系统中 LLVM IR 文件在终端设备与云端服务器上的实际部署效率与用户体验, RuyiBuild 工具链设计并实现了一套客户端与云端协同的双模式 LLVM IR 文件动态转换与资源自适应调度机制. 在云端对 IR 进行转换相较于用户完全从源码构建软件相比, 更能确保软件的一致性, 更能确保相应编译选项真正被使用, 也能降低用户的开发负担. 该机制充分考虑了客户端设备与云端服务器两类目标场景在计算资源、使用场景与用户体验需求上的显著差异, 分别设计了对应的动态转换策略与资源调度方案, 从而实现了从 LLVM IR 到目标平台精细优化的 ELF 文件的高效转换与部署.

如图 3 所示, 在客户端系统中, 首先创建一个只有单函数的 C 文件 (empty.c); 然后根据硬件的指令集信息, 使用 clang 将该 C 文件编译为 LLVM IR bitcode 格式的模板文件; 该 LLVM IR bitcode 模板文件包含了方便使用的、以 LLVM IR bitcode 格式存储的当前 CPU 的指令集信息.

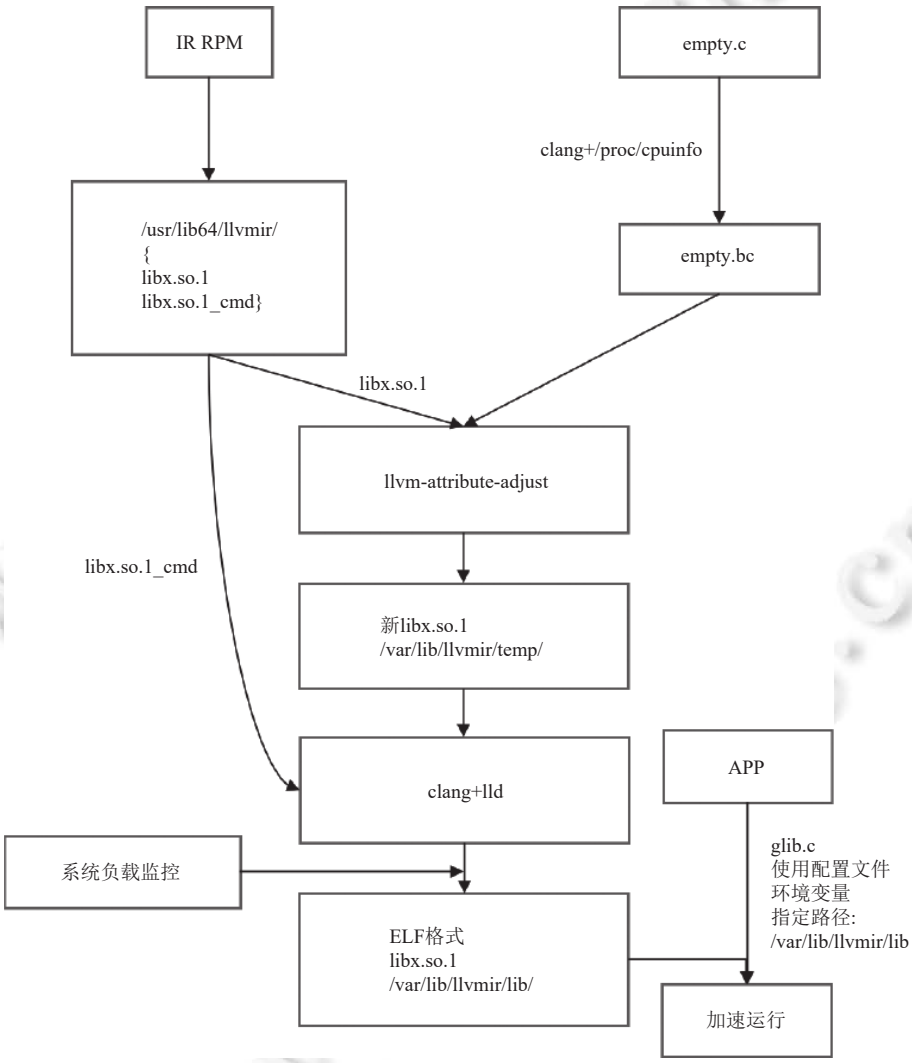

图 3 RuyiBuild 客户端 IR 转换

如图 3 所示, 用 llvm-attribute-adjust 工具, 基于模板文件提供的信息, 将图 2 中打包得到的 LLVM IR bitcode 文件进行修改, 将其中的原始指令集信息修改为模板文件中的最适合本机的指令集信息, 并生成一个新的 LLVM IR bitcode 文件. 可以根据记录的 LLVM 版本和编译动态链接库使用的命令, 使用与编译该软件时相同版本的

clang 和 lld 将新的 LLVM IR bitcode 文件转换为真正的动态链接库.

对于客户端设备而言, 由于计算资源受限且用户体验要求较高, RuyiBuild 部署了一个专门的后台服务进程, 以实时监测设备的资源使用状况 (包括 CPU 负载、内存使用情况等), 如图 3 所示. APP 侧通过 glibc 关联配置文件, 以环境变量指定 IR 依赖库路径 (/var/lib/llvmir/lib), 为转换后的加速运行提供支持. 当系统资源出现紧张状态 (例如 CPU 使用率超过 80%、内存使用超过 85%) 时, 后台服务进程自动向 LLVM IR 到 ELF 转换进程发送 SIGSTOP 信号, 立即暂停转换进程; 待系统资源恢复到正常范围 (如 CPU 使用率低于 40%、内存使用率低于 45%) 时, 后台服务进程又会发送 SIGCONT 信号, 恢复转换进程执行. 此外, 为避免转换任务占用过多资源, 后台服务进程还通过 Linux 的 cgroups 机制, 严格限制转换进程在特定 CPU 核心和内存配额内执行, 以确保客户端设备上的转换过程友好而稳定, 不会干扰用户正常使用体验. 当客户端检测到设备资源紧张或需生成大规模多架构二进制文件时, 会通过轻量化调度接口向云端提交 IR 转换任务; 云端根据任务元信息自动选择最匹配的转换环境执行批量转换, 并将生成的 ELF 文件与优化参数摘要返回客户端缓存. 当网络不可用或用户需即时转换时, 客户端则可在本地以受限资源模式独立执行任务.

在云端环境中, RuyiBuild 充分利用服务器丰富的计算资源与并行处理能力, 设计和实现了一套批量化的 LLVM IR 到 ELF 文件转换机制, 如图 4 所示. RuyiBuild 借助自动化脚本, 结合要支撑的架构和微架构列表, 先通过 llvm-attribute-adjust 为不同指令集组合生成对应新的 LLVM IR 文件, 再经 clang+lld 并行执行转换任务, 批量生成针对不同目标指令集扩展 (如 RV64GC、RV64GCV 等) 和微架构 (如 SiFive U74、香山 Kunminghu 等) 的优化 ELF 文件. 生成的文件进一步封装为对应架构的 RPM SPEC, 构建 RPM 包并接入 RPM repo, 不同用户可通过 dnf 工具调用各自文件. 这种批量并行机制, 依托云端流程实现多架构适配与高效转换, 极大提高了云端环境下的转换效率, 满足大规模部署需求, 为跨架构软件分发提供了高效的二进制产物供给能力.

如图 4 所示, 与在客户端上类似, 可根据预设置要支持的 CPU 指令集信息和微架构名称, 生成一系列的模板 LLVM IR bitcode 文件; 然后使用 llvm-attribute-adjust 工具, 根据 IR RPM 文件中的文件和模板文件, 生成一系列的临时 LLVM bitcode 文件; 将这些临时 LLVM bitcode 文件转换为适用于不同设备的真正的动态库文件并打包到不同的 RPM 软件包中.

具体来说, 为进一步提升实际的软件分发效率, RuyiBuild 实现了一种专门面向 LLVM IR RPM 软件包的批量转换与 RPM 仓库自动化构建机制. 这一机制的主要目标是将云端构建生成的所有包含 LLVM IR 文件的软件包 (即 llvmir.rpm) 批量转换成标准的常规 RPM 软件包 (即仅包含特定微架构优化后的 ELF 文件), 并自动构建 RPM 软件仓库, 以便终端用户直接通过 Linux 标准的软件管理工具 (如 dnf、yum) 进行安装.

具体实现过程如下: 首先, RuyiBuild 在云端服务器自动扫描指定目录下的所有以 llvmir.rpm 结尾的软件包, 并利用 RPM 工具自动批量解压 RPM 包, 提取出其中的 LLVM IR 文件及配套的辅助文件 (如_cmd 文件、版本脚本等). 紧接着, RuyiBuild 自动调用 LLVM 编译工具链, 批量执行 LLVM IR 到 ELF 文件的转换操作. 转换过程基于辅助文件中记录的原始编译和链接命令, 自动针对不同的目标指令集扩展和微架构进行精细的编译优化, 如附录 A.4 所示.

完成批量转换后, RuyiBuild 进一步自动生成 RPM 软件包的规范文件 (SPEC 文件), 并调用 RPM 构建工具, 批量生成常规 ELF RPM 软件包. 这些 RPM 软件包的版本号和相关的 RPM 宏定义中自动包含了目标的微架构与指令集信息, 以确保软件包的唯一性和明确性, 便于用户和管理员识别、安装和管理.

最后, RuyiBuild 工具链调用 RPM 仓库管理工具 (如 createrepo) 自动构建 RPM 仓库元数据, 在云端环境中形成标准的 RPM 软件仓库. 用户或终端设备只需配置对应的 yum/dnf 软件仓库地址, 即可直接通过标准的软件管理工具轻松安装这些预先优化好的 RPM 软件包, 如附录 A.5 所示.

上述批量 RPM 转换与仓库自动化构建机制的设计与实现, 极大地简化了终端用户的软件安装过程, 显著提高了 RISC-V 架构生态中的软件部署和分发效率. 终端用户无需再进行复杂的 LLVM IR 本地转换, 而是直接通过标准软件管理工具即可快速获得针对特定设备精细优化的软件包.

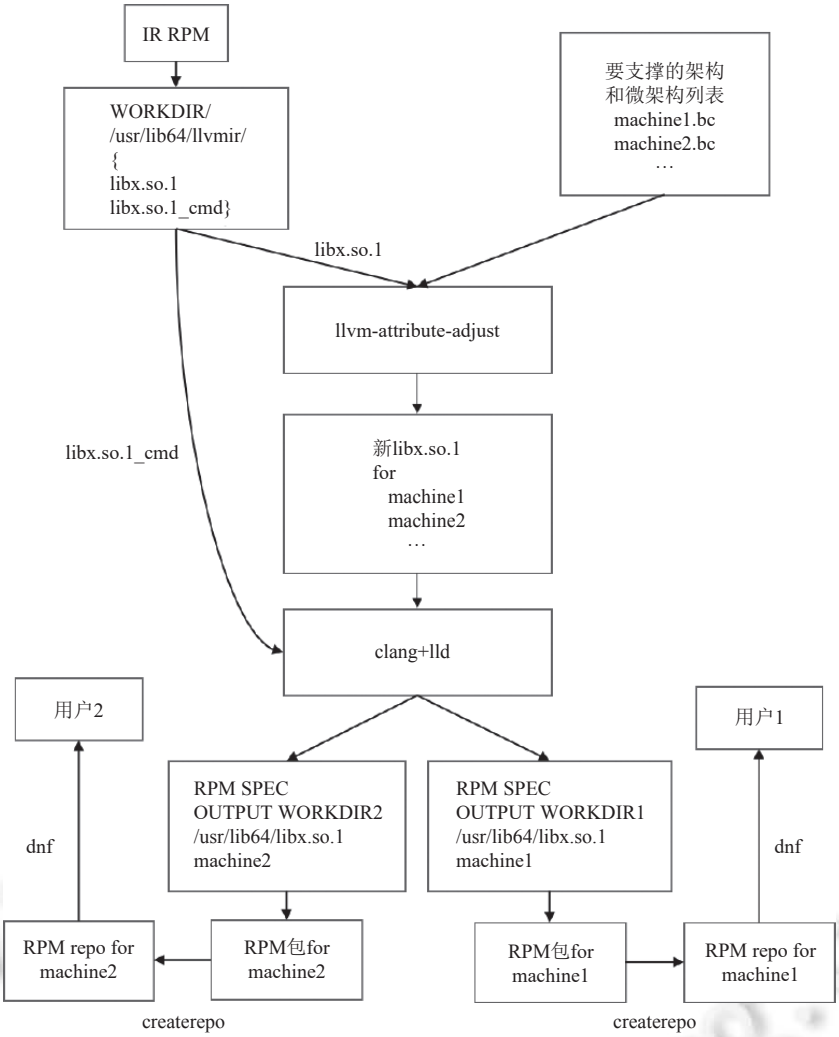

图 4 RuyiBuild 云端 IR 转换

通过上述客户端设备上的资源自适应调度机制、云端环境中的批量并行转换机制, 以及进一步设计实现的云端批量 RPM 转换与 RPM 仓库自动化构建机制, RuyiBuild 工具链有效地满足了不同应用场景对软件部署效率和用户体验的双重需求. 这一整套机制的实施, 不仅确保了客户端设备的转换任务不会影响用户体验, 也充分利用了云端服务器的资源优势, 极大简化了跨平台软件分发与部署的流程, 验证了 RuyiBuild 工具链在实际应用场景中的先进性与实用性.

3.5 基于动态模板的指令集扩展自适应与微架构优化实现机制

RISC-V 指令集架构因其开放性与模块化设计, 允许芯片制造商针对特定应用场景定制指令集扩展, 这一特征在提升硬件设计灵活性的同时, 也带来了严重的软件生态碎片化问题. 为了在这种碎片化的生态环境下实现真正意义上的软件跨平台兼容与精细化性能优化, RuyiBuild 工具链设计并实现了一套基于动态模板的指令集扩展自适应与微架构精细优化技术机制. 该机制旨在确保由统一 LLVM IR 文件构建的软件, 能够在部署具体目标设备时自动进行精确的指令集扩展适配与微架构定制优化, 达到最佳的性能表现.

具体实现上, RuyiBuild 工具链引入了一种动态模板比对与指令集扩展自动适配的流程. 首先, 在目标设备 (客户端或云端) 部署时, RuyiBuild 自动化脚本工具会自动查询目标设备的 CPU 指令集扩展特性, 特别是在 RISC-V

平台中主要通过解析系统提供的/proc/cpuinfo 文件或利用系统调用接口 (如 riscv_hwprobe 系统调用) 实现, 如附录 A.6 所示.

由于模板文件中仅包含一个简单函数, 因此编译生成的 LLVM IR 文件 (template.bc) 并不包含任何具体的业务逻辑, 但却完整包含了目标设备所支持的指令集扩展信息与相关 LLVM IR 属性. RuyiBuild 工具链随后利用这一模板 LLVM IR 文件, 自动调整待部署的目标软件 LLVM IR 文件, 确保其指令集标记与目标设备完全匹配.

具体而言, RuyiBuild 工具链实现或调用了专门的 LLVM IR 属性与指令集扩展比对与合并工具 (如 llvm-attribute-adjust). 该工具的技术实现原理是分析模板文件与目标 LLVM IR 文件的属性集合, 将模板文件的指令集扩展属性合并到目标 LLVM IR 文件, 以实现 IR 文件的指令集扩展精确匹配, 如附录 A.7 所示. 命令执行后生成 target_software_adjusted.bc 文件, 其内部的指令集扩展属性已严格匹配目标设备的指令集扩展组合. 最后, RuyiBuild 工具链调用 LLVM 工具链中的 lld 链接器, 生成精细化适配目标微架构与指令集扩展的 ELF 文件, 如附录 A.8 所示.

通过上述基于动态模板的指令集扩展自适应机制, RuyiBuild 工具链确保了最终生成的 ELF 文件精确适配目标设备的具体硬件特性, 避免了指令扩展不兼容的问题, 实现了真正意义上的跨平台兼容性.

除了指令集扩展自适应之外, RuyiBuild 工具链还进一步实现了微架构级别的精细优化机制. 具体而言, 在上述生成最终 ELF 文件的链接过程中, RuyiBuild 工具链会自动追加一系列 LLVM 编译优化标识 (如-O3、-flto、-ftree-vectorize 等高级优化选项), 并根据目标微架构的具体特征 (如乱序执行能力、流水线深度、缓存结构等) 自动调整 LLVM IR 优化参数, 以更好地发挥目标设备的硬件性能潜力. 这些优化选项与参数的组合通过反复的测试实验与性能评估确定, 在 RuyiBuild 工具链内部以预定义的配置文件形式体现, 工具链在转换过程中自动加载和应用这些优化参数, 配置文件如附录 A.9 所示. 在转换过程中, RuyiBuild 工具链自动加载配置文件, 并应用于 LLVM IR 到 ELF 文件的转换过程中, 以实现微架构级别的精细化性能优化, 转换过程使用的最优化编译选项如附录 A.10 所示.

综上所述, 通过上述基于动态模板的指令集扩展自适应与微架构精细优化机制, RuyiBuild 工具链有效解决了 RISC-V 生态系统中指令集碎片化所带来的软件兼容性与性能优化难题, 实现了统一的 LLVM IR 文件到多样化目标设备的精细化适配与优化. 这种机制不仅极大提高了软件跨平台部署的效率与可靠性, 也显著提升了软件在具体目标设备上的实际运行性能, 充分体现了 RuyiBuild 工具链的创新性与先进性.

4 实验设计与结果分析

4.1 实验数据集

为验证所提方案的性能效果, 本研究选取了 6 款具有代表性的软件进行测试, 涵盖多媒体处理、科学计算、数据库、图形渲染、信号处理及数据压缩等多个领域. 选取的软件需满足以下标准: 基于 C/C++编写的高性能共享库, 对运行效率有较高要求; 在科学计算、多媒体处理、数据库等计算密集型领域具有典型代表性; 具备通过向量指令、位操作等扩展指令集实现性能提升的潜力. 此外, 该测试方案同样适用于 Rust、Fortran 等 LLVM 框架支持的编程语言开发的软件, 选取软件如下.

- x264[35]: 开源 H.264/MPEG-4 AVC 视频编码库, 广泛应用于视频压缩领域.
- GNU GSL[36]: 开源数值计算库, 提供 CBLAS 标准的纯 C 实现, 包含丰富数学函数与算法.
- SQLite3[37]: 轻量级嵌入式关系型数据库管理系统, 支持零配置文件型存储.
- Pixman: 开源 2D 图形引擎, 专注于高效像素操作, 为上层图形库提供底层支持.
- KissFFT[38]: 轻量级开源快速傅里叶变换 (FFT) 库, 纯 C 语言实现, 适用于嵌入式环境
- xz: 为 Linux 系统常用高性能压缩工具, 基于 LZMA2 算法, 支持高压缩比.

4.2 评价指标及基准

本文采用性能耗时与数据处理速率作为核心评价指标来量化不同指令集扩展组合对软件性能的影响, 其中,

性能耗时包括编码时间 (x264)、单次操作时间 (SQLite3)、运算时间 (KissFFT)、压缩时间 (xz), 数值越小表明性能越优; 数据处理速率 (GNU GSL、Pixman) 以单位时间处理的数据量或性能测试数值衡量, 数值越大表明性能越优.

实验采用算能 SG2044 与进迭时空 K1 作为测试平台, SG2044 平台为 64 核心 RISC-V 架构 CPU; K1 平台为 8 核心 RISC-V CPU. 两平台均支持丰富的指令集扩展, 包括基础架构与核心扩展 (RV64I、M、A、F/D、C, 提供核心算术、逻辑运算及内存访问能力)、向量计算扩展 (V 系列, 包含 V、Zve32f/Zve64d 等向量长度与数据类型配置扩展及 Zvfh/Zvfhmin, 强化并行计算与 AI、图形处理效率)、位操作与内存优化扩展 (Zba/Zbb/Zbc/Zbs、Zicbom/Zicboz、Zawrs, 提升位级运算与内存访问效率)、系统与控制扩展 (Zicsr、Zifencei、Zicntr、Zicond 等, 保障系统稳定性与执行效率)、浮点增强扩展 (Zfa、Zfh/Zfhmin, 优化浮点运算性能)、压缩指令增强 (Zca/Zcb/Zcd, 提升压缩指令在特定场景的性能)、安全与特权扩展 (Svinval、Svnapot 等, 增强系统安全性与内存管理效率) 以及性能监控与调试扩展 (Zihpm, 支持细粒度性能分析与调试). 编译工具采用 llvm-project 的 main 分支 LLVM/Clang, 提交号为 2bcdfa198aa511479c46144c5cf95c7c685384ef (提交日期为 2025 年 6 月 19 日), 优化选项为-O3.

为验证所提方案的有效性, 两平台上实验均选取 5 种指令集扩展组合作为基准进行对比, 具体如下.

● RV64GCV: 在 RV64GC 基础上增加向量扩展 (V), 用于评估向量指令对计算密集型软件的性能提升效果; 其完整 RISC-V 指令为 rv64imafdcv_zicsr_zifencei_zve32f_zve32x_zve64d_zve64f_zve64x_zvl128b_zvl32b_zvl64b.

● RV64GV: 通过移除 RV64GCV 中的 C 扩展, 验证 C 扩展在特定场景下的性能影响; 其完整 RISC-V 指令为 rv64imafdv_zicsr_zifencei_zve32f_zve32x_zve64d_zve64f_zve64x_zvl128b_zvl32b_zvl64b.

● RV64GCV+B 扩展: 在 RV64GCV 基础上增加 Zba、Zbb、Zbc、Zbs 位操作扩展, 评估位操作指令与向量指令的协同优化效果; 其完整 RISC-V 指令为 rv64imafdcv_zicsr_zifencei_zba_zbb_zbc_zbs_zve32f_zve32x_zve64d_zve64f_zve64x_zvl128b_zvl32b_zvl64b.

● RVA23 子集: 基于 RISC-V RVA23 Profile 的子集 (包含 rv64imafdcv、Zicbom 等扩展), 验证新指令集标准的性能潜力; 其完整 RISC-V 指令为: rv64imafdcv_zicbom_zicboz_zicntr_zicond_zicsr_zihintntl_zihintpause_zihpm_zawrs_zfa_zfhmin_zca_zcb_zcd_zba_zbb_zbs_zve32f_zve32x_zve64d_zve64f_zve64x_zvfhmin.

● SG2044 和 K1 全指令扩展: 涵盖算能 SG2044 或进迭时空 K1 支持的所有指令扩展, 作为理论性能上限的参考基准. SG2044 平台完整 RISC-V 指令为: rv64imafdcv_zicbom_zicboz_zicntr_zicond_zicsr_zifencei_zihintntl_zihintpause_zihpm_zawrs_ zfa_zfh_zfhmin_zca_zcb_zcd_zba_zbb_zbc_zbs_zve32f_zve32x_zve64d_zve64f_zve64x_zvfh_zvfhmin_sscofpmf_sstc_svinval_svnapot_svpbmt; K1 平台完整的 RISC-V 指令为: rv64imafdcv_zicbom_zicboz_zicntr_zicond_ zicsr_zifencei_zihintpause_zihpm_zfh_zfhmin_zca_zcd_zba_zbb_zbc_zbs_zkt_zve32f_zve32x_zve64d_zve64f_zve64x_zvfh_zvfhmin_zvkt_sscofpmf_sstc_svinval_svnapot_svpbmt.

通过对比不同指令集扩展组合的性能差异, 可直接反映向量扩展、位操作扩展等指令集特性对软件性能的实际影响. 实验中, 采用多次测试取中位数 (如 KissFFT 重复 11 次) 的方式减少随机误差, 并通过一致性验证确保测试数据的可靠性. 若某指令集扩展组合在目标软件上的性能指标显著优于其他组合 (如耗时降低或速率提升超过预设阈值), 则表明该组合更适用于此类软件的优化场景.

4.3 实验方法

本实验以 6 款代表性软件 (x264、GNU GSL、SQLite3、Pixman、KissFFT、xz) 为测试对象, 在算能科技 SG2044 平台与进迭时空 K1 上开展性能测试, 重点对比 6 种指令集扩展组合 (RV64GC、RV64GCV、RV64GV、RV64GCV+B 扩展、RVA23 子集、SG2044 和 K1 全指令扩展) 对软件性能的影响. 实验中 x264[35]、GNU GSL[36]、SQLite3[37]、KissFFT[38]使用开源基准测试; Pixman 使用 Pixman 软件包自带的 lowlevel-blt-bench 命令, 具体命令为 lowlevel-blt-bench pixbuf; xz 测试方法为压缩 openEuler 24.03 SP1 RISC-V 版中的/usr/lib64/libLLVMRISCVCodeGen.a 文件, 该文件大小约 25 MB.

采用两种测试路径: 一是直接编译测试, 使用 LLVM/Clang (提交号 2bcdfa198aa511479c46144c5cf95c7c685384ef,

优化选项-O3) 分别以 5 种指令集扩展组合编译各软件, 通过对应测试工具 (如 x264-benchmark、cachebench 等) 运行并记录性能数据; 二是中间代码优化测试, 先利用 SG2044 全指令集编译生成 LLVM IR bitcode, 再通过 llvm-attribute-adjust 工具修改 IR 文件以适配不同指令集, 转换为 ELF 格式共享库后, 采用与直接编译测试相同的工具和方法进行性能评估.

为确保结果可靠性, 部分软件采用多次测试策略 (如 KissFFT 重复运行 11 次并取中位数) 以降低随机误差. 实验中, 通过对比两种测试路径的结果 (如 RuyiBuild 与 Clang 直接编译结果) 验证测试方法的一致性, 最终以性能耗时 (如编码时间、压缩时间) 或数据处理速率 (如 MB/s、MP/s (百万像素每秒)) 作为核心指标, 量化不同指令集扩展组合的性能差异.

4.4 实验结果与分析

为检验 RuyiBuild 在指令集适配中的效果, 本文以两款开发板为测试平台: 搭载进迭时空 X60 8 核 CPU、16 GB 内存的进迭时空 K1 开发板, 以及搭载玄铁 C920 64 核 CPU、128 GB 内存的算能 SG2044 开发板. 在此基础上, 对比 6 款软件在 5 种指令集组合下的性能数据, 其中, 使用 Clang 默认开启-O3 优化选项, SG2044 和 K1 上使用的前端编译最优选项如附录 A.10 所示, 具体表现见表 2 与表 3. 实验结果显示, 基于 RuyiBuild 工具链全自动生成的二进制文件与基于 Clang 程序员定制化编译的二进制文件在运行性能方面高度接近, 且在不同平台与各类指令集扩展组合下均展现出优于基线 (RV64GC) 的性能增益, 验证了其自适应方案的有效性.

表 2 RuyiBuild 指令集适配效果 (SG2044)

指令集	x264			SQLite3			GNU GSL		
	Clang (s)	RuyiBuild (s)	偏差率 (%)	Clang (ms)	RuyiBuild (ms)	偏差率 (%)	Clang (MB/s)	RuyiBuild (MB/s)	偏差率 (%)
基线RV64GC	39.033	—	—	155.272	—	—	10968.52	—	—
RV64GCV+B	**27.353**	28.045	2.53	**162.282**	163.738	0.90	**19047.38**	18646.11	−2.11
RV64GV	28.832	**28.662**	−0.59	165.579	**165.262**	−0.19	**18371.44**	17857.30	−2.80
RV64GCV	**28.660**	28.799	0.48	164.832	**161.201**	−2.20	**18212.91**	17465.15	−4.11
RVA23	**27.902**	28.254	1.26	166.481	**164.284**	−1.32	**19053.99**	18576.67	−2.51
SG2044	28.772	**28.276**	−1.72	**161.048**	163.231	1.36	**18767.27**	18520.25	−1.32

指令集	xz			KissFFT			Pixman		
	Clang (s)	RuyiBuild (s)	偏差率 (%)	Clang (s)	RuyiBuild (s)	偏差率 (%)	Clang (MP/s)	RuyiBuild (MP/s)	偏差率 (%)
基线RV64GC	24.832	—	—	62423	—	—	99.4415	—	—
RV64GCV+B	**23.168**	23.766	2.58	59394	**58056**	−2.25	327.908	327.842	−0.02
RV64GV	**24.768**	25.292	2.12	**59850**	60616	1.28	326.215	**326.982**	0.24
RV64GCV	24.731	**24.572**	−0.64	**58353**	60399	3.51	330.838	**331.383**	0.16
RVA23	**22.918**	23.151	1.02	**58029**	58665	1.10	325.723	**335.211**	2.91
SG2044	22.904	**22.839**	−0.28	58600	**58502**	−0.17	321.319	**330.469**	2.85

从整体性能偏差来看, SG2044 开发板中, RuyiBuild 与 Clang 在不同指令集扩展中的性能差异整体高度接近. 在 RV64GCV+B 组合中, x264 的偏差率为 2.53%, SQLite3 为 0.90%, GNU GSL 为−2.11%, xz 为 2.58%, KissFFT 为−2.25%, Pixman 为−0.02%; RV64GV 指令集下, 偏差率范围在−0.59% (x264)–2.12% (xz) 之间; RV64GCV 指令集中, 除 KissFFT 偏差率为 3.51% 外, 其余软件偏差均低于 1%; RVA23 子集与 SG2044 指令集下, 最大偏差分别为 2.91% (Pixman) 和 2.85% (Pixman), 整体波动幅度较小. 在 K1 开发板中, RV64GCV+B 组合中, x264 偏差率仅 0.48%, SQLite3 为 0.30%, xz 为 1.02%, KissFFT 仅 0.27%, 仅 Pixman 偏差率相对较高但也仅−2.70%, 整体波动幅度同样较小, 表明 RuyiBuild 的自动化处理能够精准复现 Clang 的编译优化效果.

从性能增益角度对比, 两平台所有指令集扩展组合 (RV64GCV+B、RV64GV、RV64GCV、RVA23 子集、SG2044、K1) 的性能显著优于基线 (RV64GC). SG2044 平台中, 以 x264 为例, 基线耗时 39.033 s, 而在各扩展指令

集下的耗时均降至 28 s 左右, 性能提升超 28%; GNU GSL 在基线中的处理速率为 10 968.52 MB/s, 在扩展指令集下提升至 17 465 MB/s 以上, 增幅超 60%; xz 的压缩时间从基线的 24.832 s 缩短至 22.839 s (SG2044 指令集), 优化效果明显; K1 平台中, 以 x264 为例, 基线耗时 616.255 s, 而在 RV64GCV+B 组合下降至约 413 s, 性能提升 33%, 同样在全部指令集扩展组合中表现出性能提升. 这种性能优势在 RuyiBuild 与 Clang 编译结果中表现一致, 说明 RuyiBuild 能够准确传递指令集扩展带来的性能红利. 特别值得注意的是, 在复杂指令集扩展场景中, RuyiBuild 的适配稳定性尤为突出. 例如, 在 RV64GCV 向量扩展下, KissFFT 在 Clang 与 RuyiBuild 编译下均较 RV64GV 基础指令集提升约 2.5%; 在 SG2044 全指令集下, SQLite3 的单次操作时间 (Clang 161.048 ms、RuyiBuild 163.231 ms) 均优于其他组合, 且两者偏差仅 1.36%. 这一结果证实, RuyiBuild 无需人工干预即可实现与 Clang 相当的优化效果, 其自动化适配机制能够有效应对 RISC-V 指令集碎片化带来的挑战, 为跨平台软件部署提供了高效且可靠的技术路径.

表 3 RuyiBuild 指令集适配效果 (K1)

指令集	x264			SQLite3			GNU GSL		
	Clang (s)	RuyiBuild (s)	偏差率 (%)	Clang (ms)	RuyiBuild (ms)	偏差率 (%)	Clang (MB/s)	RuyiBuild (MB/s)	偏差率 (%)
基线RV64GC	616.255	—	—	288.124	—	—	2 516.19	—	—
RV64GCV+B	411.444	413.431	0.48	300.632	301.542	0.30	3 589.45	3 563.34	−0.73
RV64GV	429.320	428.996	−0.08	305.687	306.493	0.26	3 524.21	3 393.45	−3.71
RV64GCV	418.791	419.725	0.22	302.606	303.089	0.16	3 569.60	3 424.79	−4.06
RVA23	409.888	410.649	0.19	300.364	303.245	0.96	3 469.40	3 450.42	−0.55
K1	412.358	410.511	−0.45	299.531	301.479	0.65	3 485.93	3 447.82	−1.09

指令集	xz			KissFFT			Pixman		
	Clang (s)	RuyiBuild (s)	偏差率 (%)	Clang (s)	RuyiBuild (s)	偏差率 (%)	Clang (MP/s)	RuyiBuild (MP/s)	偏差率 (%)
基线RV64GC	70.931	—	—	194 353	—	—	39.58	—	—
RV64GCV+B	66.390	67.066	1.02	195 862	196 391	0.27	137.93	134.2	−2.70
RV64GV	71.324	72.404	1.51	195 527	197 322	0.92	138.04	123.01	−10.89
RV64GCV	70.917	70.979	0.09	196 897	196 265	−0.32	136.33	136.23	−0.07
RVA23	66.328	66.629	0.45	195 754	196 276	0.27	138.37	135.66	−1.96
K1	66.535	66.305	−0.35	195 764	196 733	0.49	137.63	135.54	−1.52

4.5 实验结果讨论

实验数据证明了 RuyiBuild 的运行性能高度接近基于 Clang 的程序员手动设定编译. RV64GCV+B、RV64GV、RV64GCV、RVA23、SG2044 这 5 个指令集在 SG2044 开发板上的 Wilcoxon 检验结果 (p 值), 分别为 0.843 75、0.687 5、1.0、0.687 5、0.562 5; 在 K1 开发板上的 Wilcoxon 检验结果 (p 值), 分别为 0.843 75、1.0、0.687 5、0.687 5、0.843 75. 可以看出, p 值均大于 0.05, 说明在显著性水平 0.05 下, Clang 与 RuyiBuild 的性能差异均不显著. 在 98.3% (59/60) 的测试用例和指令集组合下, 如 x264 在 RV64GCV+B 扩展下 Clang 27.353 s vs. RuyiBuild 28.045 s (偏差仅 2.53%); GNU GSL 在 RV64GCV 下 Clang 18 212.91 MB/s vs. RuyiBuild 17 465.15 MB/s (偏差−4.11%), RuyiBuild 全自动生成的二进制文件性能与开发者使用 Clang 手动指定目标 ISA 编译的版本高度接近, 其中 91.7% (55/60) 偏差<3%, 最大偏差−10.89% 出现在 K1 平台 RV64GCV 配置下运行 Pixman. 这充分证明了 RuyiBuild 非侵入式捕获 LLVM IR 并在部署阶段进行自动化、目标感知优化的有效性. 其性能损失微乎其微, 却换来了无需修改源码/构建系统的巨大便利性和对未来任意 RISC-V 扩展组合的潜在支持能力, 这是传统 IFUNC (需源码级修改) 和 Multilib (需预编译海量组合) 无法比拟的. RuyiBuild 的核心创新在于其非侵入式的设计, 通过精心设计的编译工具包装 (wrapper) 和环境变量机制, 它实现了对现有软件源码和构建系统 (如 Autotools、CMake) 的零修改. 这种透明性确保了 RuyiBuild 能够无缝集成到现有的 Linux 发行版 (如 openEuler) 构建流程和 RPM/Deb 软件包管

理生态中, 显著降低了技术落地的门槛和维护成本. 这是对传统方案需要大规模修改构建系统和导致软件包数量爆炸式增长问题的根本性改进.

同时, 需要特别注意, RuyiBuild 的精准指令集适配可以保障兼容性与优化基础. 截至目前, 在实验过程中, 未出现因指令集不兼容导致的运行错误, 且在不同扩展组合下均观察到相对于基线未有扩展版本的显著性能提升, 如 x264 在所有扩展组合下耗时均从基线 39.033 s 降至 28 s 左右, 提升超过 28%; GNU GSL 速率从 10 968.52 MB/s 提升至大于 17 465 MB/s, 提升超过 60%. 这直接归功于 RuyiBuild 的动态模板指令集自适应机制, 这个提升是全自动化且显著的. 该机制确保在 SG2044 平台上, 无论目标 ISA 是 RV64GCV、RV64GV 还是更复杂的 RVA23 子集或 SG2044 和 K1 全扩展集, 最终生成的 ELF 都精确匹配了测试时指定的指令集属性, 为后续的微架构优化奠定了正确的基础, 规避了潜在的兼容性陷阱.

基于动态模板的指令集扩展自适应机制是 RuyiBuild 实现精准优化的关键. 该机制在部署时自动探测目标设备的实际 ISA 扩展 (如通过/proc/cpuinfo 或 riscv_hwprobe), 并利用一个根据探测结果编译生成的“空函数” LLVM IR 模板, 通过工具 (如 llvm-attribute-adjust) 自动调整待部署软件 IR 文件的指令集属性, 确保其与目标硬件精确匹配. 这个过程完全自动化, 无需开发者或用户干预, 彻底解决了因指令集扩展组合多样导致的兼容性风险, 并保证优化基础的正确性.

整个实验中测试了多种指令集组合 (5 种) 和多个软件 (6 款), 其优化版本的性能数据能够被系统性地获取和对比, 本身就依赖于 RuyiBuild 设计的云端批量并行转换与 RPM 仓库自动化构建机制. 该机制理论上可以高效地覆盖 RISC-V 生态中可能出现的指数级增长的指令集/微架构组合, 为用户提供“开箱即用”的最优二进制包. 用户最终通过简单的 dnf install 即可获得接近手工优化性能的软件 (如表 2 和表 3 所示), 这验证了其提升部署效率和用户体验的设计目标.

4.6 编译过程效率比较

我们对从源码编译上述软件所需时间和从 LLVM IR 转换得到相应的动态库所需时间进行对比, 如表 4 所示.

表 4 RuyiBuild 编译时间 (s)

类型	x264	SQLite3	GNU GSL	xz	KissFFT	Pixman
源码编译 (包括下载安装依赖)	227	644	344	216	14	111
从LLVM IR转换	133	84	160	10	1	31

源码编译使用 open build service 的 osc 命令进行, 使用-j64 -t 64 选项.

如图 5 所示, LLVM IR 转换构建动态库的耗时普遍显著低于传统源码编译. 其中, 例如, SQLite3 源码编译耗时达到 644 s, 而其 LLVM IR 转换仅需 84 s, 差距达一个数量级; x264 编译耗时 227 s, 而 IR 转换耗时仅 133 s, 也显示出超过 40% 的时间节约. 这一趋势在 GNU GSL 和 Pixman 等项目上同样存在.

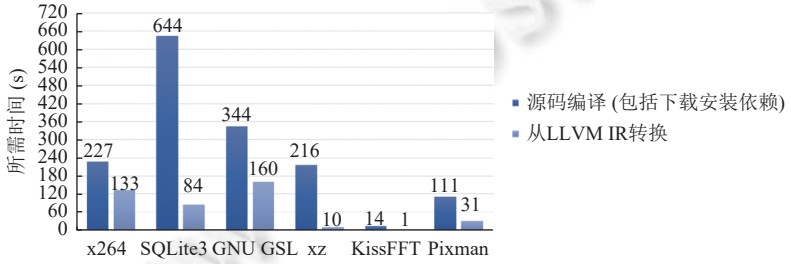

图 5 RuyiBuild 编译时间

此外, 我们进一步考察了 4 种方式 (Clang 编译、Clang LTO 编译、RuyiBuild 编译以及 IR→SO 转换) 在内存消耗与挂钟时间上的表现, 如表 5 所示.

表 5 RuyiBuild 编译表现对比

类型	x264	SQLite3	GNU GSL	xz	KissFFT	Pixman
Clang编译峰值内存 (KB)	141 164	740 512	149 964	500 128	71 812	116 844
Clang编译挂钟时间 (s)	57.69	443.20	153.08	41.59	3.05	18.91
Clang编译峰值内存 (-flto) (KB)	560 160	826 804	1 302 668	507 772	66 644	242 880
Clang编译挂钟时间 (-flto) (s)	154.92	394.27	249.89	49.75	3.14	31.95
RuyiBuild编译峰值内存 (KB)	180 052	739 688	250 484	502 884	73 004	116 352
RuyiBuild编译挂钟时间 (s)	109.78	628.81	358.96	117.19	6.47	41.42
IR→SO峰值内存 (KB)	596 756	739 448	1 380 704	130 976	64 348	280 544
IR→SO挂钟时间 (s)	133.65	85.23	160.76	11.00	1.21	31.22

比较单纯 Clang 编译和 RuyiBuild 编译可以看到, 整体内存峰值处于同一数量级; RuyiBuild 编译时间约为单纯 Clang 编译的 2 倍左右. 这一差异主要由链接/全程序优化等阶段 (LTO/链接时代码生成) 所致, 属于构建期一次性开销, 不影响交付产物的运行时性能与用户体验, 因此对业务侧“无感”.

内存消耗方面, 单纯 Clang 编译与 RuyiBuild 的结果在多数项目中较为接近. x264 在 Clang 编译下最大内存为 141 164 KB, 而在 RuyiBuild 中上升至 180 052 KB; GNU GSL 的内存消耗更为明显, 从 Clang 的 149 964 KB 增加到 RuyiBuild 的 250 484 KB. 相比之下, SQLite3 与 xz 的内存消耗峰值差异较小. 综合来看, 内存消耗量均维持在同一数量级.

如图 6 和图 7 所示, 比较 IR→SO 转换阶段和使用-flto 选项的 Clang 编译, 不管是时间还是峰值内存占用量, IR→SO 转换均没有超过使用 LTO 选项的 Clang 编译. 同时, x264、SQLite3、GNU GSL、Pixman 的内存峰值占用量均较为接近, 这是因为这些峰值均产生于链接阶段, LTO 选项会消耗较大内存. 实际上, IR→SO 的转换过程中也使用了 LTO 优化. 使用 RuyiBuild 编译软件所消耗的时间和内存均小于直接使用 Clang 编译的 2.5 倍, 整体编译时间与内存占用量仍处于同一数量级, 且由于编译由服务端完成, 可通过常规工程手段被吸收; 从 LLVM IR 向动态链接库的转换需要消耗一些时间, 其均短于使用 LTO 选项的 Clang 编译; 从 LLVM IR 向动态链接库的转换需要的内存大致与使用 LTO 选项的 Clang 编译相同, 综合考虑构建期一次性成本与通过 IR 管线带来的产物一致性、可追溯性与跨目标的可重复构建收益, 上述时间与内存开销处于可接受区间.

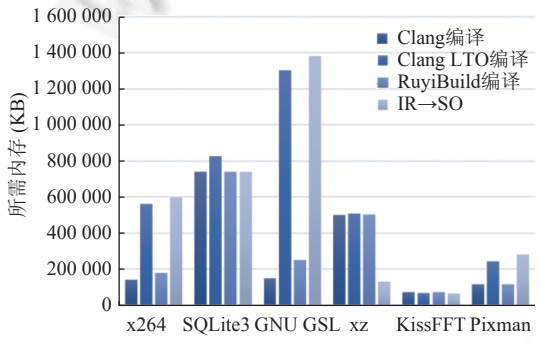

图 6 4 种编译方案的内存占用量

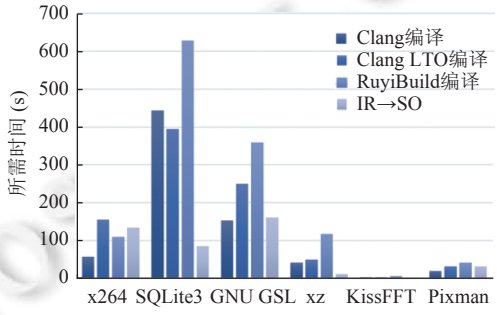

图 7 4 种编译方案的挂钟时间

5 总 结

随着 RISC-V 指令集架构在学术界与产业界的广泛应用, 其指令集开源以及模块化、可定制的设计理念虽显著推动了芯片领域的创新发展, 但也引发了软件生态的严重碎片化问题, 成为制约 RISC-V 生态系统扩展的关键挑战. 本文围绕该问题, 系统提出并实现了 RuyiBuild 工具链, 作为面向 RISC-V 平台的软件兼容性感知解决方案. RuyiBuild 以 LLVM IR 作为统一的中间表示形式, 在保持原有构建系统兼容性的前提下, 设计了一种透明化

的双路径编译机制, 能够在构建过程中自动提取并同步生成 LLVM IR 文件, 为后续跨平台部署奠定基础. 为支持软件分发与生态集成, RuyiBuild 构建了完整的 LLVM IR 部署与 RPM 包管理机制, 并通过自动镜像与一致性维护实现 IR 与传统 ELF 包的协同发布. 此外, 本文提出了一种基于动态模板的指令集扩展识别与微架构优化方法, 能够根据目标平台的硬件特性进行精准识别与动态重构, 从而实现细粒度的编译优化. 为进一步提升系统适应性, RuyiBuild 在客户端设计了资源感知的转换调度机制, 保障转换过程对用户体验的最小影响; 同时在云端构建了自动化批量转换与优化流程, 实现多架构环境下的高效部署. 通过编译拦截、IR 文件管理、部署集成与分发优化的全流程构建, RuyiBuild 实现了兼容性与优化能力兼具的跨平台软件构建范式, 克服了现有方案粒度控制粗、维护成本高与构建系统侵入性强等核心难题. RuyiBuild 工具链不仅为当前 RISC-V 平台上的软件构建与部署提供了可行、易用、高效的新范式, 更为未来跨架构软件分发与自动优化系统的研究提供了重要的参考路径. 针对部分 CPU 在启用扩展指令集后性能反降的现象, 我们将继续探索基于硬件特性感知的指令集参数精细微调机制, 结合程序概要分析等技术优化指令集与应用负载的匹配度; 研究人工智能驱动的转换控制策略; 实现动态自适应的转换决策, 构建可执行程序的跨平台优化与分发框架, 完善全类型软件的适配能力; 探索面向商业闭源软件的适配路径, 在保障知识产权的前提下, 设计安全可控的 IR 转换与优化机制, 推动技术在产业生态中的规模化应用.

References

[1] Waterman A, Asanović K. The RISC-V instruction set manual Volume I: Unprivileged ISA (Version 20191213). 2019. https://acsa.ustc.edu.cn/ics/download/riscv/The%20RISC-V%20Instruction%20Set%20Manual.pdf

[2] Cui EF, Li TZ, Wei Q. RISC-V instruction set architecture extensions: A survey. IEEE Access, 2023, 11: 24696–24711. [doi: 10.1109/ACCESS.2023.3246491]

[3] Alonso M, Andreu D, Canal R, Di Carlo S, Chenet C, Costa J, Girones A, Gizopoulos D, Karakostas V, Otero B, Papadimitriou G, Rodríguez E, Savino A. Validation, verification, and testing (VVT) of future RISC-V powered cloud infrastructures: The Vitamin-V horizon europe project perspective. In: Proc. of the 2023 IEEE European Test Symp. (ETS). Venezia: IEEE, 2023. 1–6. [doi: 10.1109/ETS56758.2023.10174216]

[4] Asanović K, Patterson DA. Instruction sets should be free: The case for RISC-V. Technical Report, UCB/EECS-2014-146, EECS Department, University of California at Berkeley, 2014.

[5] Celio C, Chiu PF, Nikolic B, Patterson DA, Asanović K. BOOM v2: An Open-source out-of-order RISC-V core. Technical Report, UCB/EECS-2017-157, EECS Department, University of California at Berkeley, 2017.

[6] Gautschi M, Schiavone PD, Traber A, Loi I, Pullini A, Rossi D, Flamand E, Gürkaynak FK, Benini L. Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. IEEE Trans. on Very Large Scale Integration (VLSI) Systems, 2017, 25(10): 2700–2713. [doi: 10.1109/TVLSI.2017.2654506]

[7] Mezger BW, Santos DA, Dilillo L, Zeferino CA, Melo DR. A survey of the RISC-V architecture software support. IEEE Access, 2022, 10: 51394–51411. [doi: 10.1109/ACCESS.2022.3174125]

[8] Declaring Attributes of Functions. 2016. https://gcc.gnu.org/onlinedocs/gcc-5.3.0/gcc/Function-Attributes.html#index-g_t_0040code_007bifunc_007d-function-attribute-3095

[9] libc6.1: Request ev67 specific build. 2004. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=229251

[10] Müllner C. [PATCH v2 14/15] RISC-V: Add Zbb optimized strlen as ifunc. Sourceware: Libc-alpha mailing list. 2024. https://sourceware.org/pipermail/libc-alpha/2024-May/157095.html

[11] Kumar R, Zyuban V, Tullsen DM. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In: Proc. of the 32nd Int'l Symp. on Computer Architecture. Madison: IEEE, 2005. 408–419. [doi: 10.1109/ISCA.2005.34]

[12] openEuler. Build result of openCV package on openEuler 24.03 LTS SP2 (RISC-V64). 2025. https://eulermaker.compass-ci.openeuler.openatom.cn/package/build?osProject=openEuler-24.03-LTS-SP2:epol:risc-v&packageName=opencv

[13] Kalapothas S, Galetakis M, Flamis G, Plessas F, Kitsos P. A survey on RISC-V-based machine learning ecosystem. Information, 2023, 14(2): 64. [doi: 10.3390/info14020064]

[14] Peccia FN, Haxel F, Bringmann O. Tensor program optimization for the RISC-V vector extension using probabilistic programs. In: Proc. of the 2025 IEEE/ACM Int'l Conf. on Computer Aided Design. Munich: IEEE, 2025. 1–9. [doi: 10.1109/ICCAD66269.2025.11241007]

[15] Armstrong A, Bauereiss T, Campbell B, Reid A, Gray KE, Norton RM, Mundkur P, Wassell M, French J, Pulte C, Flur S, Stark I, Krishnaswami N, Sewell P. ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS. Proc. of the ACM on Programming Languages, 2019, 3(POPL): 71. [doi: 10.1145/3290384]

[16] Bora S, Paily R. A high-performance core micro-architecture based on RISC-V ISA for low power applications. IEEE Trans. on Circuits and Systems II: Express Briefs, 2021, 68(6): 2132–2136. [doi: 10.1109/TCSII.2020.3043204]

[17] Hassan QF, Sagahyroon A. RISC-V: A comprehensive overview of an emerging ISA for the AI-IoT era. In: Hassan QF, ed. Advances in the Internet of Things. Boca Raton: CRC Press, 2025. 244–284.

[18] Lazzeri E, Forlin BE, Furano G, Ottavi M, Cassano L. An experimental comparison of RISC-V processors: Performance, power, area and security-special session paper. In: Proc. of the 2024 IEEE Int'l Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT). Didcot: IEEE, 2024. 1–6. [doi: 10.1109/DFT63277.2024.10753540]

[19] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the 2004 Int'l Symp. on Code Generation and Optimization (CGO 2004). San Jose: IEEE, 2004. 75–86. [doi: 10.1109/CGO.2004.1281665]

[20] de la Torre JC, Ruiz P, Dorronsoro B, Galindo PL. Analyzing the influence of LLVM code optimization passes on software performance. In: Proc. of the 17th Int'l Conf. on Information Processing and Management of Uncertainty in Knowledge-based Systems. Cádiz: Springer, 2018. 272–283. [doi: 10.1007/978-3-319-91479-4_23]

[21] Zhou ZD, Ren ZL, Gao GJ, Jiang H. An empirical study of optimization bugs in GCC and LLVM. Journal of Systems and Software, 2021, 174: 110884. [doi: 10.1016/j.jss.2020.110884]

[22] Jimborean A, Loechner V, Clauss P. Handling multi-versioning in LLVM: Code tracking and cloning. In: Proc. of the 2011 Workshop on Intermediate Representations, in Conjunction with CGO (WIR 2011). Chamonix: HAL, 2011.

[23] Moses W, Churavy V. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 1046.

[24] Garba P, Favaro M. SATURN-software deobfuscation framework based on LLVM. In: Proc. of the 3rd ACM Workshop on Software Protection. London: ACM, 2019. 27–38. [doi: 10.1145/3338503.3357721]

[25] Wang AJ, Yi XY, Yan YH. UPIR: Toward the design of unified parallel intermediate representation for parallel programming models. In: Proc. of the 2022 Int'l Conf. on Parallel Architectures and Compilation Techniques. Chicago: ACM, 2022. 530–531. [doi: 10.1145/3559009.3569646]

[26] Ünay E, İnan B, Yiğit E. Supporting custom instructions with the LLVM compiler for RISC-V processor. arXiv:2310.18353, 2023.

[27] Zakowski Y, Beck C, Yoon I, Zaichuk I, Zaliva V, Zdancewic S. Modular, compositional, and executable formal semantics for LLVM IR. Proc. of the ACM on Programming Languages, 2021, 5(ICFP): 67. [doi: 10.1145/3473572]

[28] VenkataKeerthy S, Aggarwal R, Jain S, Desarkar MS, Upadrasta R, Srikant YN. IR2VEC: LLVM IR based scalable program embeddings. ACM Trans. on Architecture and Code Optimization (TACO), 2020, 17(4): 32. [doi: 10.1145/3418463]

[29] Sarang AD, Choi SH, Park KW. Plotting OSS-based supply chain attack strategies and the defense failure. In: Proc. of the 25th Int'l Conf. on Information Security Applications. Jeju Island: Springer, 2024. 311–323. [doi: 10.1007/978-981-96-1624-4_24]

[30] Lins M, Mayrhofer R, Roland M, Hofer D, Schwaighofer M. On the critical path to implant backdoors and the effectiveness of potential mitigation techniques: Early learnings from XZ. arXiv:2404.08987, 2024.

[31] Agrawal V, Dabral A, Palit T, Shen YM, Ferdman M. Architectural support for dynamic linking. In: Proc. of the 20th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Istanbul: ACM, 2015. 691–702. [doi: 10.1145/2694344.2694392]

[32] von Hagen W. The Definitive Guide to GCC. Apress, 2011.

[33] Chen KJ, Arias O, Deng QX, Oliveira D, Guo XL, Jin YE. FineDIFT: Fine-grained dynamic information flow tracking for data-flow integrity using coprocessor. IEEE Trans. on Information Forensics and Security, 2022, 17: 559–573. [doi: 10.1109/TIFS.2022.3144868]

[34] Horan B. Cross compile environment. In: Horan B, ed. Practical Raspberry Pi. Berkeley: Apress, 2013. 105–124. [doi: 10.1007/978-1-4302-4972-6_6]

[35] x264-benchmark. 2025. https://github.com/wzssyqa/x264-benchmark/tree/no-abs-path

[36] GNU GSL. 2025. https://github.com/wzssyqa/cachebench/tree/gsl

[37] SQLite3 benchmark. 2025. https://github.com/wzssyqa/sqlite-bench/tree/use-system-libsqlite3

[38] KissFFT. 2022. https://github.com/project-gemmi/benchmarking-fft

附录 A

A.1 封装 install 和 ln

以文件安装工具 install 为例, 构建系统调用如下命令安装软件文件.

```
install -D -m 755 libexample.so.1.0.0 $DESTDIR/usr/lib64/libexample.so.1.0.0
```

RuyiBuild 包装脚本首先调用上述原始命令, 以传统方式安装 ELF 文件. 随后, 自动检测到该操作, 额外调用一次镜像命令, 在 LLVM IR 对应的存储目录中安装对应的 LLVM IR 文件.

```
install -D -m 755 $LLVMIR_BASEDIR/$(pwd)/libexample.so.1.0.0 \
  $LLVMIR_BASEDIR/$DESTDIR/usr/lib64/libexample.so.1.0.0
```

同时, 该包装机制还会自动处理与动态库相关的辅助文件, 如前述章节中提到的记录动态库链接命令的_cmd 文件、符号版本脚本_verscript 文件等, 也一并进行镜像安装.

```
install -D -m 644 $LLVMIR_BASEDIR/$(pwd)/libexample.so.1.0.0_cmd \
  $LLVMIR_BASEDIR/$DESTDIR/usr/lib64/libexample.so.1.0.0_cmd
install -D -m 644 $LLVMIR_BASEDIR/$(pwd)/libexample.so.1.0.0_verscript \
  $LLVMIR_BASEDIR/$DESTDIR/usr/lib64/libexample.so.1.0.0_verscript
```

软链接工具 ln 的包装扩展机制与上述思路类似. 当构建系统调用软链接创建命令:

```
ln -sf libexample.so.1.0.0 $DESTDIR/usr/lib64/libexample.so.1
```

RuyiBuild 工具链包装脚本额外自动执行镜像软链接操作:

```
ln -sf libexample.so.1.0.0 $LLVMIR_BASEDIR/$DESTDIR/usr/lib64/libexample.so.1
```

A.2 扫描文件

扫描命令如下.

```
find $LLVMIR_BASEDIR/BUILDROOT -type f \
  \( -name "*.so*" -o -name "*.a" -o -name "*_cmd" -o -name "*_verscript" \)
```

A.3 生成 RPM 包

自动生成的 RPM SPEC 文件片段如下.

```
%files llvmir
/usr/lib64/llvmir/libexample.so.1.0.0
/usr/lib64/llvmir/libexample.so.1
/usr/lib64/llvmir/libexample.so.1.0.0_cmd
/usr/lib64/llvmir/libexample.so.1.0.0_verscript
%post llvmir
# 在安装完成后, 触发后台服务执行转换任务
/usr/bin/RuyiBuild-trigger-conversion /usr/lib64/llvmir/libexample.so.1.0.0
```

A.4 批量转换

针对不同的指令集扩展组合或微架构选项, RuyiBuild 自动执行类似如下的批量转换命令.

```
# 批量转换示例 (针对不同指令集和微架构)
for rpm_dir in /cloud/llvmir_rpm_extract/*; do
```

```
  ir_file=$(find $rpm_dir -name '*.so.bc')
  cmd_file=$(find $rpm_dir -name '*_cmd')
  # 提取原始的转换命令
  original_cmd=$(cat $cmd_file)
  # 针对不同指令集扩展进行批量转换
  for isa in rv64gc rv64gcv; do
    elf_output=${ir_file%.so.bc}.$isa.so
    clang -shared $ir_file -march=$isa -fuse-ld=lld -O3 -o $elf_output
  done
done
```

A.5 使用云端转换版本

用户可以通过如下命令直接安装特定微架构优化版本的软件包.

```
# 用户端 dnf 安装示例
dnf config-manager --add-repo http://cloud.example.com/repos/rv64gc/
dnf install example-package
```

A.6 创建指令集模板

RuyiBuild 工具链自带的指令集解析脚本可自动提取目标设备上的 ISA 扩展信息.

```
# 示例: 自动提取 RISC-V 设备指令集扩展信息
ISA_EXTENSIONS=$(grep ^isa /proc/cpuinfo | head -n1 | awk '{print $2}')
echo "Detected ISA extensions: $ISA_EXTENSIONS"
```

上述脚本执行后, 会得到类似的信息: Detected ISA extensions: rv64imafdcv_zicsr_zifencei.

随后, RuyiBuild 工具链利用上述得到的指令集扩展信息, 自动生成一个仅包含空函数的 C 语言模板文件 (empty.c).

```
// empty.c: 空函数模板文件
int empty_function(void) { return 0; }
```

并利用目标设备的指令集扩展信息编译生成对应的 LLVM IR 模板文件.

```
# 生成模板 IR 文件
clang -emit-llvm -march=$ISA_EXTENSIONS -c empty.c -o template.bc
```

A.7 调整目标 IR 文件

调整目标 IR 文件指令集属性匹配模板.

```
llvm-attribute-adjust --template=template.bc \
  --input=target_software.bc \
  --output=target_software_adjusted.bc
```

A.8 编译生成优化的动态库

生成最终精细化优化的 ELF 格式动态库.

```
clang -shared target_software_adjusted.bc -fuse-ld=lld \
  -march=$ISA_EXTENSIONS -O3 -o target_software_final.so
```

A.9 配置转换器

针对特定 RISC-V 微架构的优化参数配置文件如下所示.

```
OptLevel = 3
VectorizeLoops = true
UnrollLoops = true
InlineThreshold = 500
EnableLTO = true
```

A.10 SG2044 和 K1 上使用的前端最优编译选项

SG2044 平台下 RuyiBuild 从 C 到 IR 使用如下命令:

march=rv64imafdcv_zicbom_zicboz_zicntr_zicond_zicsr_zifencei_zihintntl_zihintpause_zihpm_zawrs_zfa_zfh_zfhmin_zca_zcb_zcd_zba_zbb_zbc_zbs_zve32f_zve32x_zve64d_zve64f_zve64x_zvfh_zvfhmin_sscofpmf_sstc_svinval_svnapot_svpbmt

K1 平台下使用如下命令:

march=rv64imafdcv_zicbom_zicboz_zicntr_zicond_zicsr_zifencei_zihintpause_zihpm_zfh_zfhmin_zca_zcd_zba_zbb_zbc_zbs_zkt_zve32f_zve32x_zve64d_zve64f_zve64x_zvfh_zvfhmin_zvkt_sscofpmf_sstc_svinval_svnapot_svpbmt

RVA23U64 使用如下命令:

march=rv64imafdcvh_zicbom_zicboz_ziccrse_zicntr_zicond_zicsr_zifencei_zihintntl_zihintpause_zihpm_zimop_zawrs_zfa_zfhmin_zca_zcb_zcd_zcmop_zba_zbb_zbs_zkt_zvbb_zve32f_zve32x_zve64d_zve64f_zve64x_zvfhmin_zvkb_zvkt_smnpm_smstateen_sscofpmf_ssnpm_sstc_svade_svinval_svnapot_svpbmt

作者简介

屈晟, 高级工程师, CCF 专业会员, 主要研究领域为高性能操作系统, 系统安全.

苏运强, 硕士, 主要研究领域为高性能操作系统.

林哲远, 博士生, CCF 学生会员, 主要研究领域为软件工程.

燕言言, 博士生, CCF 学生会员, 主要研究领域为程序设计语言质量保障.

冯洋, 博士, 副教授, CCF 高级会员, 主要研究领域为程序设计语言, 编译技术, 软件工程.

赵琛, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为编译技术, 操作系统, 网络软件.