分类号：　　密级：

U D C ：　　学号：

南 昌 大 学 硕 士 研 究 生

学 位 论 文

基于 RISC-V 架构的容器化可重现方法研究

Research of Reproducible Container Based on RISC-V Architecture

崔　傲

培养单位（院、系）：　数学与计算机学院

指导教师姓名、职称：　徐子晨 教授

申请学位的学科门类：　工学

学科专业名称：　　计算机科学与技术

论文答辩日期：

答辩委员会主席：_____

评阅人：_____

_____

年　月　日

书脊(提交论文电子版时，此页请删除)

一、学位论文独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得南昌大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名（手写）：　　签字日期：　　年　　月　　日

二、学位论文版权使用授权书

本学位论文作者完全了解南昌大学有关保留、使用学位论文的规定，同意学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权南昌大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编本学位论文。同时授权北京万方数据股份有限公司和中国学术期刊（光盘版）电子杂志社将本学位论文收录到《中国学位论文全文数据库》和《中国优秀博硕士学位论文全文数据库》中全文发表，并通过网络向社会公众提供信息服务，同意按“章程”规定享受相关权益。

学位论文作者签名（手写）：　　导师签名（手写）：

签字日期：　　年　　月　　日　　签字日期：　　年　　月　　日

论文题目					
姓　　名		学号		论文级别	博士□　硕士□
院/系/所		专业			
E_mail					
备注：					

□公开　□保密（向校学位办申请获批准为“保密”，＿＿＿年＿＿月后公开）

摘　要

通过记录、重放和确定性执行等技术重现程序中的并发错误，在程序编译、调试、性能优化等场景下具有重要研究意义和实用价值。随着新兴 RISC-V 指令架构软硬件生态的日益完善，在新硬件上支持跨指令集的程序可重现成为亟待解决的关键技术难题。现有研究往往重现开销高、可移植性差、重现结果无法保证，并不能满足跨平台程序重现的要求。针对上述问题，本文在充分分析传统程序可重现性研究基础上，设计并实现了一种基于 RISC-V 架构的容器化可重现方法，具体研究内容如下：

首先，研究现有记录重放和确定性执行技术，探索二者的局限性，分析用户进程执行过程中的不确定性来源，并就相关问题的国内外研究现状展开调查与分析。

其次，结合记录重放、确定性执行等工作中的部分先进技术，提出一种用户进程的确定性执行方法。通过追踪进程查找并定位不确定的系统调用，结合 LXC 容器构成确定性的独立执行环境，进一步优化多线程的访存竞争消解，实现线程确定性调度，保证用户进程的确定性执行。

最后，在确定性执行工作基础上，通过 Docker 容器技术，将用户进程确定性执行方法移植到 RISC-V 架构下，实现用户进程确定性执行方法的部署与优化。

本文整体方法的系统原型基于 GEM5 平台，实现了 RISC-V 指令集与程序重现方法重演。为了验证所提方法的有效性，设计并实现了一整套容器化可重现框架的验证方法，并通过与现有的记录重放工具进行比较表明，容器化可重现方案能够实现更加通用的确定性，同时具有更小的额外性能开销。

关键词：RISC-V，可重现性，确定性，容器

ABSTRACT

Reproducibility technology reproduces concurrent errors in programs by means of Record and Replay and Deterministic Execution, and has important research significance and practical value in program debugging and other fields. With the increasing improvement of the software and hardware ecology of the emerging RISC-V instruction architecture, providing reproducibility of program execution on new hardware has become an urgent problem to be solved. However, the existing work has high extra overhead and poor portability, and cannot fully meet the requirements of reproducibility. In view of the above problems, this paper fully analyzes the previous reproducibility research work, and designs and implements a containerized reproducible method based on RISC-V architecture. The specific work is as follows:

First, research the existing record playback and deterministic execution technologies, explore their limitations, analyze the sources of uncertainty in the execution of user processes, and investigate and analyze the current research status of related issues at home and abroad.

Secondly, a deterministic execution method of user process is proposed in combination with some advanced technologies in recording and replaying, deterministic execution and so on. The method finds and intercepts uncertain system calls by tracing the process, uses LXC container to provide a deterministic execution environment for the process, optimizes the memory access competition and thread scheduling mechanism of multi-threaded programs, and ensures the deterministic execution of user processes. Compared with the previous method of recording and playback, the additional performance penalty is reduced.

Then, using the container technology, the deployment and optimization of the deterministic execution method of the user process is realized under the RISC-V architecture. Use Docker to encapsulate the deterministic abstraction, implement a two-layer container nested with LXC, eliminate the impact of the underlying micro-architecture on portability, and isolate the uncertain impact from the file system; inside the container, use dynamic binary translation The technique replaces CPU instructions

that are non-deterministic during user process execution. The containerized reproducible solution avoids modification of programs and operating system kernels, reduces deployment and maintenance costs, and better ensures the determinism of user processes.

Finally, the RISC-V hardware platform is simulated on the Gem5 full-system simulator, and experiments are designed to verify the feasibility of the containerized reproducible scheme, and compare with the existing recording and playback tools. Experimental results show that the containerized reproducible scheme can achieve more general determinism with less additional performance overhead.

Key Words: RISC-V, Reproducibility, Determinism, Container

目 录

目录

第 1 章 引言

1.1 研究背景与意义

随着登纳德缩放定律和摩尔定律的终结，标准处理器性能提升的减速已成为了既定事实，新的体系结构“黄金时代”需要领域独特的设计语言和指令架构[1]。在开源软件生态的驱动下，加州大学伯克利分校设计了开源的精简指令集架构 RISC-V（RISC Five）。RISC-V 避免了传统指令架构（如 x86、Arm、MIPS 等）发展历程中累积的弊端，模块化的设计赋予它简洁、高效的特性，并于近年来涌现出一系列基于 RISC-V 指令架构的处理器设计[2][3][20-24]及软件拓展[16-19]。

随着 RISC-V 软件生态的日益完善，解决 RISC-V 架构上的程序可重现性问题变得愈发重要。可重现性的概念强调同一种方法在不同条件下获得一致性的研究结果。在计算机科学领域，可重现性意味着数据流的确定性，即每个输入重复映射到唯一的确定性输出，主要体现在如下三个应用场景：（1）软件包构建。保证发布的软件在任意节点上具有可重现性是很多机构（如微软在 C#和 VB 编译器中的工作[6]、谷歌在 Blaze/Bazel 构建工具中的工作等[74]）的研究目标之一；（2）并行程序调试。多核 RISC-V 处理器的设计逐渐增多，而并行程序调试过程需要保证错误的可重现性，借此改善程序的可靠性[4]；（3）高性能 RISC-V 处理器被用于深度学习模型的训练和推理，可重现的训练过程能够帮助研究人员追踪模型性能变化趋势，理解神经网络决策原因[5]；因此，基于 RISC-V 架构实现程序的可重现性在 RISC-V 软件生态的发展中有着重要意义。

可重现性可以进一步分解为两个较弱的子属性，即确定性和可移植性[6]：确定性保证相同输入情况下，程序反复执行始终得到唯一确定的结果；可移植性则保证在可接受的额外开销范围内实现不同机器上的确定性输出。

按照重现范围和实现方法，可重现性研究可以分为两类，分别为记录重放（Record and Replay, R&R）和确定性执行（Deterministic Execution）。记录重放技术[7-9]大多面向多线程工作负载，通常应用于程序调试领域。顾名思义，记录重放的实现分为记录和重放两个阶段：记录阶段追踪、保存程序的执行踪迹（Execution Trace）到日志文件，并在重放阶段按照日志记录的踪迹信息重放程

序的执行过程，得到相同结果。确定性执行的解决方案通过消除程序执行过程中受到的不确定性（Non-deterministic）因素影响，在系统级别保证程序执行的确定性。在实现中，确定性执行通过修改系统内核[25][26]等方法来构建确定性的系统抽象，针对操作系统组件如进程、线程、网络等，监视并拦截影响程序执行的不确定因素，强制确定性执行程序。

然而，目前的可重现性研究尚存在几个关键问题：一是确定性范围小，部分传统可重现方法不支持与操作系统文件目录和 I/O 操作交互[25][48]；通用性差，无法记录所有类型程序[7]；二是可移植性差，传统方法往往针对特定指令集架构（一般是 x86 架构）和操作系统环境下实现[11][12]；三是部署成本高，程序确定性重放工具的部署需要在源程序基础上进行修改[10]，或者修改操作系统内核[46]，或者增加重现专用硬件[28]，增加部署和维护成本；四是额外开销大，由于需要存储空间保存不确定事件的执行顺序，这类记录重放工具往往造成较大的额外性能开销，尤其是记录整个虚拟机执行踪迹的方案[11][12]；因此，现有的可重现研究并不能完全满足确定性和可移植性的要求，且存在一定的性能优化空间。

基于上述背景，RISC-V 上的程序执行的可重现性问题亟需提出一种兼顾确定性和可移植性，同时具有较低额外开销的方法。围绕上述基本问题，本文结合现有可重现性和容器技术，研究在 RISC-V 指令架构上的容器化可重现方法，重点研究通过 chroot 系统调用和命名空间等扩展确定性操作系统的边界，通过 Docker 容器技术增强可移植性，使用基于进程追踪的确定性执行方法替代传统的记录重放，减小方案的额外性能开销，最终在多核 RISC-V 模拟硬件上实现对容器化可重现方法的功能、性能验证。

1.2 国内外研究现状

可重现性研究源于并行程序调试需求，并不断应用到程序编译、性能优化、容错等其它领域中，国内外研究人员提出了一系列经典方法和解决方案，为后续工作提供支撑。按照重现范围和实现方法，可以将目前的大部分可重现性研究划分为以下两个方向，分别是记录重放和确定性执行。但是传统方法往往针对特定指令集架构和操作系统，缺少在 RISC-V 指令架构上的相关工作。随着近年来涌现出多个基于 RISC-V 指令架构的高性能开源处理器设计，解决面向 RISC-V 架构的可重现问题具有重要的研究意义和应用价值。

本章节将围绕上述问题介绍国内外 RISC-V 指令架构和可重现性相关工作，首先对国内外 RISC-V 指令集架构的研究现状进行综述，其次分别从记录重放和确定性执行两方面介绍国内外可重现性研究工作；最后介绍容器和虚拟化技术的发展历程，并对其中可重现性的相关工作进行综述。

1.2.1 RISC-V 指令集研究现状

RISC-V 是加利福尼亚大学伯克利分校提出的新一代精简指令集架构[13][14]，具有开源、模块化的设计。受到 Linux 等开源软件社区的影响，RISC-V 同样搭建了开源生态，由 RISC-V 国际基金会负责管理，并完全免费提供给任何个人或者机构设计微处理器架构。RISC-V 的开发者成立了 SiFive 公司，推出了 E 系列、S 系列和 U 系列板卡。同时，有越来越多的公司加入了 RISC-V 国际基金会，包括阿里巴巴、Qualcomm 和 Intel 等国际巨头，与活跃的开源社区一起不断地丰富 RISC-V 软硬件生态。

在模拟器领域，已经有多种纯软件模拟器支持模拟 RISC-V 指令架构。Spike[15]是一个 RISC-V 指令集模拟器，实现了 RISC-V 架构处理器的功能模型。Spike 忽略了 I/O 访问和内存事务等延迟，使用代理内核执行用户空间程序。QEMU 同样支持了 RISC-V 仿真功能[16]；FireSim 在云端提供了一个周期精确级 FPGA 全加速硬件仿真平台，运行在 Amazon EC2 F1 云 FPGA 上，已经提供了对 RISC-V 处理器核的支持[17-19]；Gem5 也相继添加了对 RISC-V 单核、多核处理器的全系统模拟。

国内外已经设计了一系列基于 RISC-V 的微处理器架构，如国外的推出的一系列开源乱序处理器核 Rocket[20]、BOOM[21]、BROOM[2] 等。国内也已经推出了基于 RISC-V 的 64 位高性能微处理器玄铁 910[22]、开源高性能处理器核“香山”[23]、可信高性能处理器核“蓬莱”[24]等一系列处理器架构。

1.2.2 基于记录重放的可重现性研究现状

在 19 世纪 80 年代后期，研究人员逐渐认识到并行程序调试过程中可重现性的需求，并基于确定性重放技术设计并实现了多种先进的记录和重放工具，这些早期的确定性执行方案为后期的多项技术提供了思路。1987 年，LeBlanc 等人[4]提出了 Instant Replay，作为一种重现并行程序执行行为的通用解决方案。在当

时的软件调试过程中，顺序程序可以通过不断地循环执行，发现并纠正程序执行中的错误。但是由于共享内存竞争的存在，对并行程序的多次执行可能会产生不同的结果。Instant Replay 通过记录重点事件发生的时序，将程序重放所需的信息保存在日志中，并在重放阶段读取日志信息还原程序的执行过程。Instant Replay 后来成为了多项确定性重放技术的前身，乃至发展出了分布式内存上的可重现模型。1988 年，Pan 等人提出了 Recap[27]，结合检查点和数据重放记录方法，在程序执行期间记录系统调用、共享内存读取结果及异步事件（信号）发生的时序。相较于 Instant Replay，Recap 实现了从某一个检查点重放的功能，避免了从程序头重新执行的弊端。但是 Recap 同样使用日志机制保存事件信息，增加了巨大的额外存储和性能开销。此外，Bacon 等人[28]通过记录硬件指令计数器保存共享内存访问的总顺序，之后基于硬件辅助机制的确定性重放技术基本上采用了同样的解决方案。

早期的确定性重放工作逐渐奠定了日后对相关并行程序确定性重放工作的研究方向与思路，即分为记录和重放两阶段，不同方法之间的区别在于记录信息的方式、对象不同，带来的额外性能损失也不同。如 PinPlay[29]提出一种易于使用的确定性重放框架，用于记录，确定性地重放和分析具有合理运行时和磁盘使用情况的大型并行程序的执行；Patil 等人[30]提出一种记录程序执行踪迹的运行时框架。但是上述框架需要检测共享内存的负载，增加了很大的额外开销。

由于记录阶段往往需要大量的内存空间保存程序执行信息，大部分工作负载难以接受确定性重放工具的额外开销。FDR[31]、BugNet[8]、Rerun[32]、DeLorean[33]和 QuickRec[34]等项目探索了基于硬件支持的确定性重放方法，降低记录与重放阶段的开销，但同时也出现了其它的问题。如确定性重放工具 BugNet[8]能够在程序崩溃之前持续记录程序信息，如寄存器内容等，并确定性地重放程序。Bugnet 不支持记录跟踪程序的 I/O 信息，中断和 DMA 传输信号等，降低了硬件支持的复杂度和记录信息的规模，但是也破坏了程序的通用性，不支持对具有上述信息的工作负载进行记录和重放。Jockey[10]可以将程序运行中的动态链接库拦截系统调用和 CPU 指令。但是 Jockey 需要作为目标进程的一部分运行，增加了部署和维护成本，并且不支持很多程序。

另外一些方案则着重于实现任意二进制程序的确定性重放，以适度的性能损耗为代价，提供程序的通用性。Devietti 等人提出了 DMP[35]，一个完全确定性的共享内存多处理器。DMP 将任意多线程程序的行为约束为输入的函数，保证

进程间通信的完全确定性。Bergan T 等人提出了 CoreDet[36]来消除并行程序中的不确定性。CoreDet 是一个编译器和运行时系统，能够确定性地运行任意多线程 C/C++ POSIX 线程程序。Calvin[37]提出了一个确定性的共享内存模型；RCDC[38]使用软硬件结合的方法，提出了一个高性能确定性执行架构；Dthreads[39] 通过替换 PThreads 函数库的方式实现了确定性的多线程系统。

1.2.3 基于确定性执行的可重现性研究现状

记录重放工具需要额外的存储空间保存事件发生的顺序，往往附加较大的额外开销，因此难以在资源受限的微处理器环境中应用记录重放工具。另外一种可重现性研究是确定性执行，即通过消除程序执行中的不确定性因素，强制保证程序执行期间的确定性。确定性执行主要面向操作系统中的进程、线程等组件进行修改，提供确定性的操作系统抽象。

学术界和工业界都提出了许多确定性执行系统，如通过修改操作系统内核等方式[11][12]实现全系统的确定性重放工作。1999 年，Rosse 等人在早期确定性执行方案的基础上，提出了一项确定性可执行框架 RecPlay[44]。通过使用标量时钟记录同步事件顺序，使用矩阵时钟在重放期间进行资源竞争检测。性能评估显示 RecPlay 的最坏情况执行时间开销为 25.9%。之后的大部分全系统重放工作中都采用了逻辑时钟的方式增强可重现性。

一部分研究人员设计了完全确定性的计算机系统。Devecsery 等人提出了 Eidetic Systems[45]，设计并实现了一个确定性的计算机系统，通过将系统分成重放进程组和跟踪进程组，分析组之间的信息流，使得跟踪组可以重放另一个进程组执行所需的数据。同时 Eidetic Systems 通过压缩文件记录信息来减少确定性重放的空间开销。Lee 等人提出了 Respec[46]通过修改 Linux 内核，在商用多核处理器上实现了一个确定性的操作系统，并且设计实现了并发执行记录和重放过程的机制。Mashtizadeh 等人提出了 Castor[47]，一个多核处理器应用设计的记录重放系统。

此外，Aviram 等人[48]、Hunt 等人[49]、Bergan[50]等人均提出了操作系统级别的可重现抽象。Aviram 等人 2010 年提出了 Determinator[48]，这是一个围绕确定性设计的操作系统抽象，强制单个进程甚至进程组确定性执行，实现了操作系统级别的确定性。Determinator 拒绝用户程序直接访问硬件资源，包括实时时钟、

周期计数器和可写共享内存等，来强制执行确定性。Hunt 等人提出的 DDOS[49]利用先前的工作提供了分布式系统的确定性执行工作，通过在每个节点上实现确定性，保证输出是输入的函数。dOS[50]提出了一个新的系统抽象，其中的 shim 抽象与 Linux 内核系统调用$ptrace$ 类似，能够追踪并拦截进程中的不确定因素。dOS 支持进程和线程的并行执行，但是 dOS 依旧使用记录重放工具实现文件系统交互中的确定性。

上述基于确定性执行的可重现性研究依旧存在一些问题，如通过更改操作系统内核保证可重现性的方法，使确定性的维护和部署更加困难；除非将确定性重放组件集成到基本操作系统中，但是向操作系统内核添加具有侵入性的新特性是有风险的；主要消除了线程调度中不确定性的影响，但是对确定性抽象的边界定义过窄，无法使用于对 I/O 操作、文件系统、信号等操作有要求的应用；针对特定的处理器和操作系统设计，难以在不同的微架构之间移植。

1.2.4 基于虚拟机和容器的可重现性研究

虚拟化技术（Virtualization）是在主机上将硬件抽象为虚拟对象环境的技术，能够模拟计算机的 CPU、内存、I/O 设备、网络设备等各种物理资源[51]，创建多个模拟环境，即虚拟机。虚拟化技术能够在计算机设备中运行多个异构的体系结构应用，通过虚拟机管理程序（Virtual Machine Monitor，VMM，或称为 Hypervisor）为用户提供抽象、虚拟的硬件环境，运行虚拟机管理程序的硬件称作主机，使用主机资源运行的虚拟机成为客户虚拟机。图 1.1（a）展示了虚拟机层次架构，Hypervisor 在宿主机和虚拟机之间添加一层中间层，将宿主机的底层硬件和上层应用程序分离，并提供资源的分配管理[52]。VMware、QEMU[53]等虚拟机软件均是采用这种方法实现硬件的虚拟化。

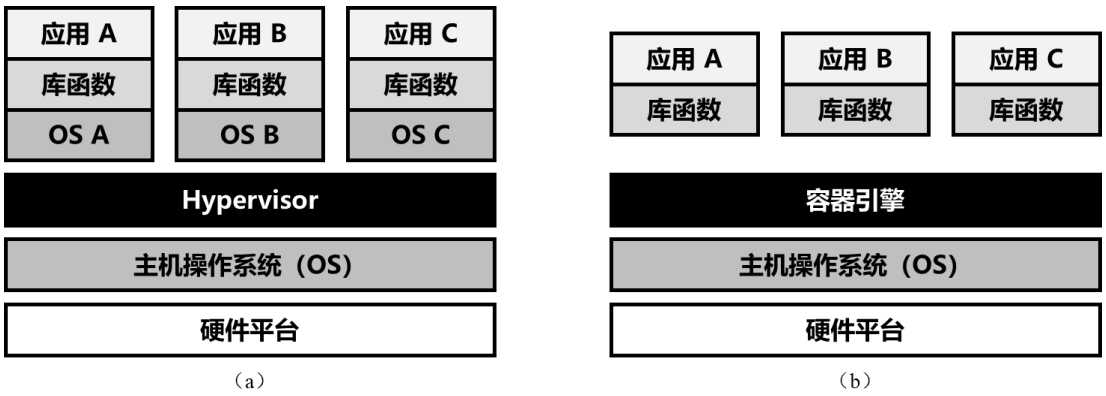


图 1.1 （a）虚拟机和（b）容器的架构比较

由于虚拟机可以隔离主机操作系统，消除程序对操作系统的依赖关系，许多研究人员提出了多种在虚拟机上的可重现方案。ReVirt[11]记录并重放了整个虚拟机的执行，它通过将记录日志的操作转移到虚拟机中，逐个重放虚拟机的指令。SMP-ReVirt[55]实现了对多核处理器虚拟机的记录和重放，使用硬件页面保护来检测和准确地重放多核处理器虚拟机中 CPU 之间的共享，同时允许重放整个操作系统和所有应用程序。但是 SMP-ReVirt 的高额开销是大部分工作负载和应用程序无法承受的。VMware 曾经使用 ReVirt 的技术实现 VMware Workstation 中的可重现性调试[56]，但是在后续的版本中舍弃了该功能。此外，QEMU[57-59]和 Xen[55][60]等虚拟机管理程序也设计了一系列可重现方案。但是在虚拟机设置回访的检查点的会增加更多的开销，并且将应用程序部署到虚拟机中也并不方便，所以需要基于更轻量级的容器技术设计可重现性方案。

容器技术架构如图 1.1（b）所示，与虚拟机相比较，容器共享主机操作系统内核，只隔离开应用程序和系统其它部分，是一种更加轻量级的虚拟化技术。最早的容器技术起源于 Linux 内核中用于切换根目录的 chroot 操作，结合 Linux 内核中控制组（Cgroups）、命名空间（Namespace）等功能发展出 Linux 容器（Linux Containers，LXC 技术）[54]。Docker 是目前常用的容器工具（0.9 版本之前的 Docker 是使用 LXC 技术开发的），提供了高度灵活的容器创建、部署及复制等功能，有效地隔离了应用程序和主机系统，并提供了将容器在不同环境中部署地可移植性。Navarro 等人在 Intel x86 处理器上设计了一个可重现容器[6]，验证了 12130 个软件包在容器内可重现的构建。

1.3 主要研究内容

本文总体研究路线如图 1.2 所示。其中，主要研究内容为 RISC-V 架构上的可重现性问题。可重现性进一步分解为两个子属性：确定性和可移植性，并将可重现性问题分解成用户进程的确定性执行问题和 RISC-V 架构上的可移植性优化问题。针对两个问题分别设计解决方法，其中确定性问题通过现有确定性执行研究中的追踪进程技术，消除用户进程执行过程中不确定性因素影响；可移植性问题通过容器技术隔离用户进程和底层硬件，实现在 RISC-V 架构上的移植优化；最后设计实验对容器化可重现方案的功能和性能进行验证。

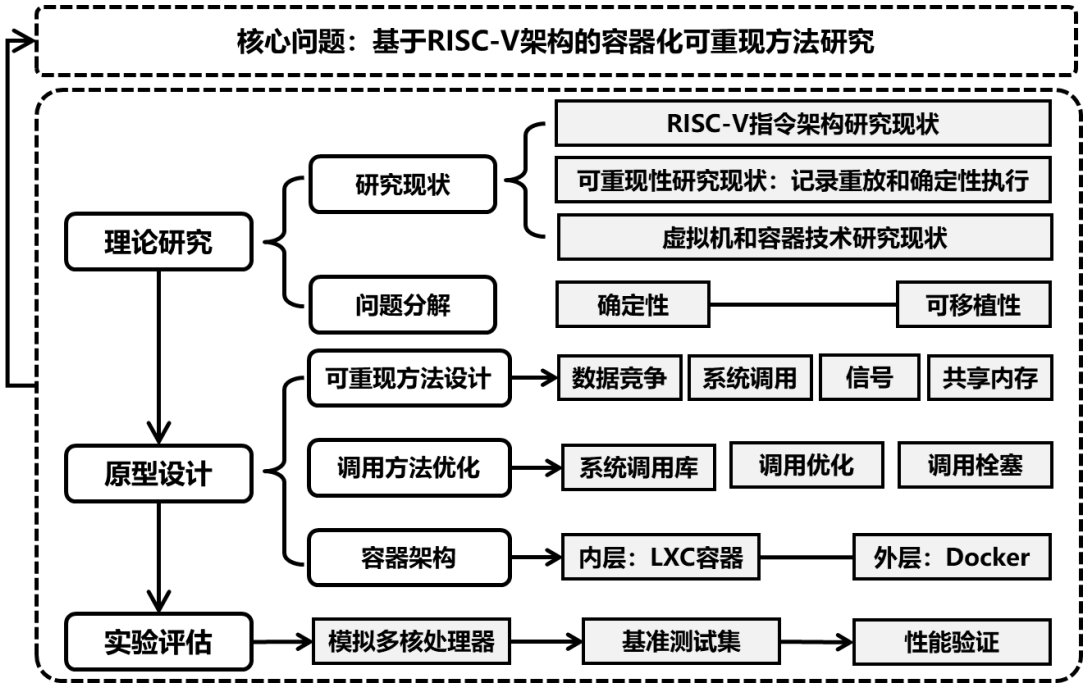


图 1.2 基于 RISC-V 的容器化可重现方法研究路线

本文的具体工作如下：

（1）用户进程的确定性执行

本文研究用户进程执行过程中的不确定性因素来源，结合确定性执行、记录重放等工作中的部分先进技术，设计并实现一个用户进程确定性抽象。通过追踪进程查找并拦截不确定的系统调用，通过 LXC 容器为进程提供确定性的执行环境，综合数据竞争、线程调度、共享内存等技术，保证用户进程的确定性执行。相较于使用记录重放的方法，降低了额外的性能损耗。

（2）基于容器的可移植性优化

为了在 RISC-V 架构下实现用户进程确定性抽象，使用 Docker 对确定性抽象进行封装，实现与 LXC 嵌套的二层容器，消除底层微架构对可移植性的影响，同时隔绝了来自文件系统的不确定性影响；在容器内部，利用动态二进制翻译技术替换用户进程执行过程中具有不确定性的 CPU 指令。容器化可重现方案避免了对程序和操作系统内核的修改，减少了部署和维护的成本，同时更好的保证了用户进程的确定性。

（3）在 Gem5 全系统模拟器上仿真 RISC-V 硬件平台，设计实验验证容器化可重现方案的可行性，并与现有的记录重放工具进行比较。实验结果表明，容器化可重现方案具有更小的额外性能开销。

1.4 组织结构

本文包括 5 个章节，主要用 3 个章节阐述基于 RISC-V 架构的容器化可重现方法研究工作，具体内容安排如下：

第一章：引言。介绍在 RISC-V 架构上设计和实现容器化可重现方法的研究背景和意义，从 RISC-V 指令集研究现状、基于记录重放的可重现性研究现状、基于确定性执行的可重现性研究现状以及基于虚拟机和容器的可重现性研究问题四个方面介绍国内外相关工作，介绍本文的主要研究内容并列出本文的组织结构。

第二章：RISC-V 指令集与可重现性研究。详细介绍了 RISC-V 指令架构和可重现性研究的相关理论知识，主要包括 RISC-V 指令集设计思想、RISC-V 基础指令集、拓展指令集及其特权架构、可重现性技术及其主要实现方法以及容器的相关理论知识。

第三章：该章节介绍基于 RISC-V 的容器化可重现方法的设计细节。首先介绍基于用户进程中使用系统调用函数$ptrace$拦截和修改系统调用的方法设计，以及数据竞争、共享内存等调度方法，然后介绍使用 LXC 容器封装用户进程和追踪进程，通过 Docker 隔离文件系统的方法拦截的不确定性因素的主要来源。

第五章：实验设计与分析。该章节在全系统模拟器 Gem5 上仿真 RISC-V 平

台，设置基准测试集和实验方案，验证 RISC-V 架构上的容器化可重现方法的可行性，并与传统记录重放工具进行对比，分析容器化可重现方法产生的性能损耗。

第五章：总结。对论文所做的工作进行总结。

第 2 章 RISC-V 指令集与可重现性研究

本章详细介绍了 RISC-V 指令架构和可重现性方法的相关理论知识。在 RISC-V 指令架构方面介绍了 RISC-V 基础指令集、拓展指令集及其特权架构的设计；可重现性技术中介绍了用户进程中常见的不确定性来源，以及确定性执行的追踪进程技术，并介绍了 RISC-V 上的相关容器技术。

2.1 RISC-V 指令架构

现有通用指令集架构如 x86、ARM、MIPS 等在长期发展暴露出了种种问题，日益发展的指令设计和封闭的生态环境不利于体系结构的继续发展。为了规避传统指令集架构的各种弊端，加州大学伯克利分校于 2010 年发布了 RISC-V，一个免费的开源精简指令集架构。RISC-V 在设计之初就为了顺应体系结构发展趋势制定了如下发展目标：

（1）开放性

现有商用指令集架构（如 x86、ARM 等）的使用需要支付昂贵的专利授权费用，限制了体系结构设计研发和转化的成本，不利于技术发展。RISC-V 的标准化工作完全由 RISC-V 基金会主持，并宣布未来“不受任何单一公司的浮沉或一时兴起的决定的影响”[69]，任何的个人或组织都可以自由使用 RISC-V 指令集架构进行处理器设计与开放工作。

（2）模块化设计

现有指令集架构在其长期的版本迭代中必须考虑向后兼容性，即新版本处理器不仅必须实现新的指令集拓展，还需要支持老版本的所有指令集拓展，如 Intel x86 指令集架构支持向后兼容，则 x86 64 位 CPU 的设计必须支持 x86 32 位指令架构，以此保证早期架构版本开发的应用在新的指令集中正确运行。如图 2.1 所示，长此以往导致现有 x86 指令集架构过于冗杂，指令集体量随时间大量增长。

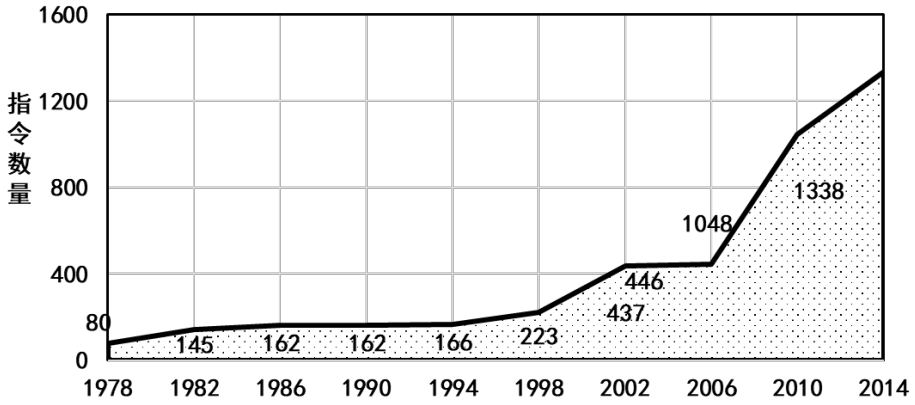


图 2.1 Intel x86 指令集中的指令数量在持续不断增长

RISC-V 作为完全崭新的指令集架构，采用模块化设计方式增加架构的可拓展性，仅需要支持基础指令集模块 RV32I 即可实现完整的处理器设计。RISC-V 在设计之初保留了操作码空间，面向特定领域（如深度学习、增强现实、图计算等）任务，研究人员可以为其添加自定义指令，针对领域独特应用的资源、能耗需求进行精细化的处理器设计。

（3）精简性

由于 20 世纪 80 年代的半导体制造工艺问题，处理器时钟频率偏低，当时的指令集设计目标是尽量在每条指令中实现更多的功能，且指令集包含多种不同的指令与格式。如 ARM-32 指令集中存在指令：

$$Ldmiaeq\ SP!,\{R4-R7,PC\}$$

该指令执行 5 次数据加载并写入 6 个寄存器，但仅当EQ条件码置位时才执行。此外，它将结果写入PC寄存器，因此它也执行条件分支。长指令及复杂指令格式的存在破坏了指令集架构的精简性。RISC-V 指令集架构避免了过于复杂的指令设计，实现同样的功能设计需要更少、更简洁的指令，能够设计出更小面积的芯片。

2.1.1 RISC-V 基础指令集

RISC-V 的核心设计为基本整数指令集，所有指令集实现及拓展中必须包含基本整数指令集。RISC-V 的基本整数指令集与早期的精简指令集（Reduced Instruction Set Computer，RISC）设计类似，但是不包括分支延迟间隙和可变指

令编码。基本整数指令集提供了一组最小指令集合，为定制微处理器指令集架构提供必要功能。RISC-V 的基本整数指令集主要区别在于整数寄存器位宽（XLEN）以及整数寄存器的数量，如表 2.2 所示，最新的 RISC-V 规范包含了 5 种基础指令集：

表 2.2 RISC-V 基础指令集

基础指令集	内容	版本	状态
RVWMO	弱内存次数指令集	2.0	正式批准
RV32I	32 位基本整数指令集	2.1	正式批准
RV64I	64 位基本整数指令集	2.1	正式批准
RV32E	32 位嵌入式整数指令集	1.9	草案
RV128I	128 位基本整数指令集	1.7	草案

RV32I 和 **RV64I**：2 种主要的整数指令集，分别提供 32 位和 64 位地址空间。

RV32E：RV32I 的子集变体，用来支持嵌入式微处理器。

RV128I：支持 128 位地址空间，为未来的 128 位处理器设计保留空间。

RVWMO：描述了 RISC-V 指令架构所使用的内存一致性模型。

以 RV32I 为例，如图 2.2（a）所示，RV32 定义了 32 个 32 位 x 寄存器（XLEN=32），x0 寄存器所有位被强制硬件布线为 0，通用寄存器 x1-x31 保存二进制指令；额外非特权寄存器 pc 用于保存当前指令地址，又称作程序计数器。

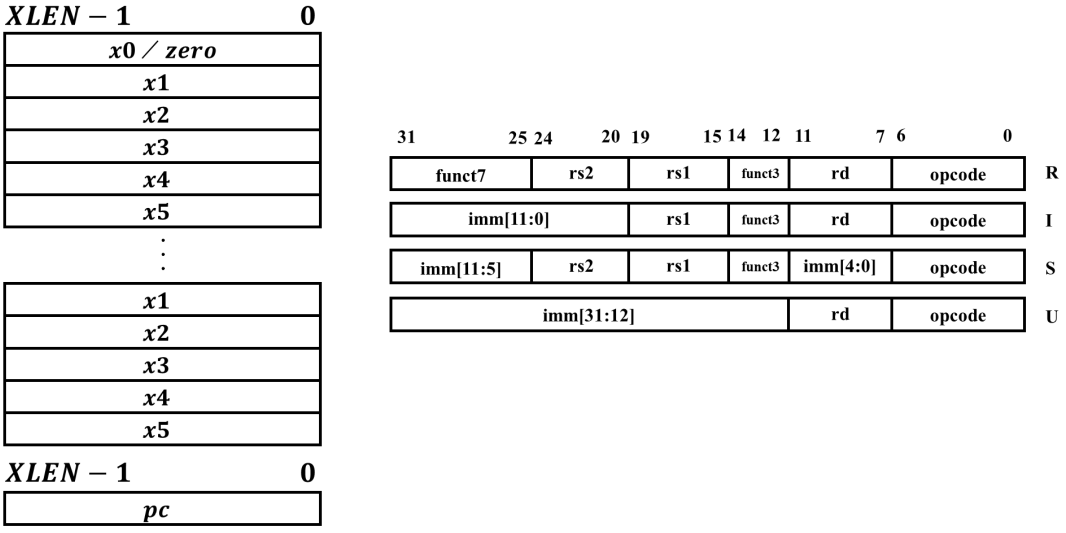


图 2.2 （a）RV32I 寄存器状态　　（b）RV32I 的 4 种基础指令格式

RV32I 有 4 种核心的指令格式：R/I/S/U，如图 2.4（b）所示。R 类型指令用

于寄存器-寄存器操作；I 类型操作用于短立即数和访存 load 操作；S 类型指令用于访存 store 操作；U 类型指令用于长立即数操作。所有指令格式固定为 32 位，且必须在对齐内存中的 4 字节边界。若目标地址非 4 字节对齐，则会在执行分支或无条件跳转指令时出现指令地址未对齐异常。此外 RV32I 拥有 2 种指令格式变体：B/J，分别用于条件跳转操作和无条件跳转操作。

RV64I 使用 32 个通用寄存器和 4 种指令格式，区别在于整数寄存器位宽 XLEN=64。同时新增$*W$指令，用于操作低 32 位；新增了$ADDIW$ 指令，用以产生 32 位的计算结果，再将其符号拓展至 64 位并忽略计算溢出。

RV32E 针对嵌入式环境，进一步精简了指令集设计，与 RV32I 相比，RV32E 仅使用 16 个通用寄存器和 pc 完成所有功能，并且使用专用的寄存器调用约定 ILP32E。

随着计算需求的增长，未来可能会需要超过 64 位的地址空间，因此最新的 RISC-V 指令集规范设计了 RV128I 基本整数指令集，将整数寄存器宽度扩展为 128 位，保留了$*W$指令，新增用于操作寄存器低 64 位的$*D$指令。

RVWMO 定义了 RISC-V 的内存一致性模型（RISC-V 弱内存排序），主要遵循 RC（Release Consistency）理论模型，使用较少的内存访问顺序约束。RVWMO 内存模型根据“全局内存次序”定义，即所有 hart（硬件线程，hardware thread）产生的内存操作的总顺序。通常，一个多线程程序有许多不同的可能执行，每个执行都有自己相应的全局内存顺序。“全局内存次序”定义在由内存指令生成的原语加载$(load)$ 和存储$(store)$操作上，受到部分定义的约束。任何满足所有内存模型约束的执行都是合法执行(就内存模型而言)。

2.1.2 RISC-V 扩展指令集

RISC-V 的模块化设计体现在，基于 RISC-V 基础整数指令集，可以选择面向特定任务的指令扩展。扩展指令集提供了包括整数乘除法指令、浮点数计算指令、原子指令、压缩指令、向量指令等扩展，并且仍在不断地修订、增添新的扩展指令集。根据最新的 RISC-V 规范文档，现有 RISC-V 扩展指令集主要有以下 24 种：

表 2.4 RISC-V 扩展指令集

基础指令集	内容	版本	状态
M	乘除法扩展指令集	2.0	正式批准
A	原子指令扩展	2.1	正式批准
F	单精度浮点扩展指令集	2.2	正式批准
D	双精度浮点扩展指令集	2.2	正式批准
Q	四精度浮点扩展指令集	2.2	正式批准
C	压缩指令扩展	2.0	正式批准
Counters	计数器和计时器	2.0	草案
L	十进制浮点	0.0	草案
B	位操作	0.0	草案
J	动态翻译语言	0.0	草案
T	事务内存	0.0	草案
P	组合 SIMD 指令	0.2	草案
V	向量操作	1.0-rc	草案
Zicsr	控制和寄存器	2.0	正式批准
Zifencei	屏障指令	2.0	正式批准
Zihintpause	提示暂停	2.0	已批准
Zam	非对齐原子操作	0.1	草案
Zfh	半精度浮点	0.1	草案
Zfhmin	半精度浮点最小集合	0.1	草案
Zfinx	整数寄存器单精度浮点	1.0.0-rc	冻结
Zdinx	整数寄存器双精度浮点	1.0.0-rc	冻结
Zhinx	整数寄存器半精度浮点	1.0.0-rc	冻结
Zhinxmin	整数寄存器半精度浮点最小集	1.0.0-rc	冻结
Ztso	全存储排序	0.1	冻结

2.1.3 RV32/64 特权架构

最新的 RISC-V 特权架构文档列出了 RISC-V 除用户模式（User Mode，U 模式）以外，拥有的 6 种具有更高权限的模式，如表 2.5 所示：

表 2.5 RISC-V 中的特权架构拓展

特权架构	内容	版本	状态
Machine ISA	机器模式	1.12	正式批准
Supervisor ISA	监管者模式	1.12	正式批准
Svnapot Extension	自然对齐的二次幂地址转换连续性	1.0	正式批准
Svpbmt Extension	基于页面的内存类型	1.0	正式批准
Svinval Extension	细粒度的地址转换缓存失效	1.0	正式批准
Hypervisor ISA	虚拟机管理模式	1.0	正式批准

1. 用户模式

用户模式（User Mode，U 模式）管理应用程序的执行过程，无法访问操作系统的内核资源。

2. 机器模式

机器模式（Machine Mode，M 模式）是 RISC-V 指令架构中 hart（Hardware Thread，硬件线程）可以执行的最高权限模式。在 M 模式下运行的 hart 对内存、I/O 设备和一些对于启动和配置系统来说必要的底层功能有着完全的使用权。因此它是唯一所有标准 RISC-V 处理器都必须实现的权限模式，不同的处理器可以根据应用场景需求选择是否支持其它 U/S/H 模式。

3. 监管者模式

监管者模式（Supervisor Mode，S 模式）用于机器中各种操作系统级别操作，如虚拟地址转换等。S 模式权限介于 M 模式和 U 模式之间。S 模式的核心是使用基于页面的虚拟内存实现内存保护。这是一种用于更复杂 RISC-V 处理器上的可选模式。S 模式的权限基于 U 模式和 M 模式之间，不能使用 M 模式下的控制状态寄存器（Control and Status Register，CSR）和指令。

在 32 位指令架构中，S 模式仅支持 1 种分页虚拟内存方案 Sv32；在 64 位架构中，S 模式定义了 Sv39、Sv48 和 Sv57 共三种分页虚拟方案，并将在后续规范中添加 Sv64 方案。RISC-V 特权架构规范通过了一系列虚拟内存部分的扩展，包括：

（1） “Svnapot”标准扩展：

自然对齐的二次幂（Naturally Aligned Power-of-2，NAPOT）地址转换连续性。

（2） “Svpbmt”标准扩展

基于页面的内存类型（Page-based Memory Types）。

（3） “Svinval”标准扩展。

细粒度的地址转换缓存失效（Fine-Grained Address-Translation Cache Invalidation）。

4. 虚拟机管理模式

虚拟机管理模式（Hypervisor Mode，M 模式）用于管理跨机器的资源，虚拟化 S 模式级别资源架构，简化经典虚拟化技术的使用，并提高虚拟化性能。图 2.3 为 RISC-V 特权架构能够支持的软件栈架构：

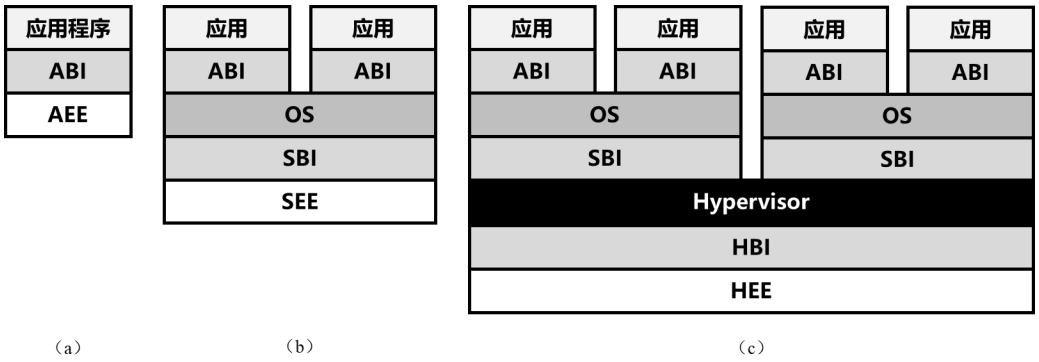


图 2.3 RISC-V 特权架构支持的软件栈及各层执行环境抽象

自顶向下，RISC-V 特权架构软件栈依次被抽象为：

（1）应用程序二进制接口（Application Binary Interface，ABI）；

（2）应用执行环境（Application Execution Environment,，AEE）

（3）操作系统（Operate System，OS）

（3）Supervisor 二进制接口（Supervisor Binary Interface, SBI）；

（4）Supervisor 执行环境（Supervisor Execution Environment，SEE）

（5）虚拟机管理程序 Hypervisor；

（6）Hypervisor 二进制接口（Hypervisor Binary Interface, HBI）；

（7）Hypervisor 执行环境（Hypervisor Execution Environment，HEE）

图 2.3（a）是最小系统，只支持运行单个应用程序的应用执行环境，应用程序和执行环境中通过二进制接口连接；图 2.3（b）增加了操作系统层，支持多个程序的并行执行；图 2.3（c）增加了虚拟机管理程序 Hypervisor，支持运行多个虚拟机操作系统，隔离了硬件平台的具体细节。

2.2 可重现性技术

可重现性技术最初是为了在存在不确定性因素的情况下实现并行程序循环调试，实现可重现性的基本思路是寻找程序执行中的不确定性因素，不确定性因素的来源主要包括：（1）共享内存访问竞争；（2）中断、异常等异步事件；（3）带有不确定性质的系统调用和 CPU 指令等。实际应用中需要考虑更多来源的不确定因素，将会在 2.2.1 节中介绍。可重现性技术已经成为程序调试、容错等领域必须考虑的因素之一。

可重现技术可分为记录重放和确定性执行。记录重放也常被称为确定性重放（Deterministic Replay），通常分为记录和重放 2 个阶段。在记录阶段，确定性重放工具记录程序执行流程中的系统调用中的不确定信息等，在重放阶段读取保存的信息并重现程序的执行过程。图 2.4 表示了一个简单的确定性重放工作流程模型，除了系统调用外，不考虑其它不确定性信息的影响。

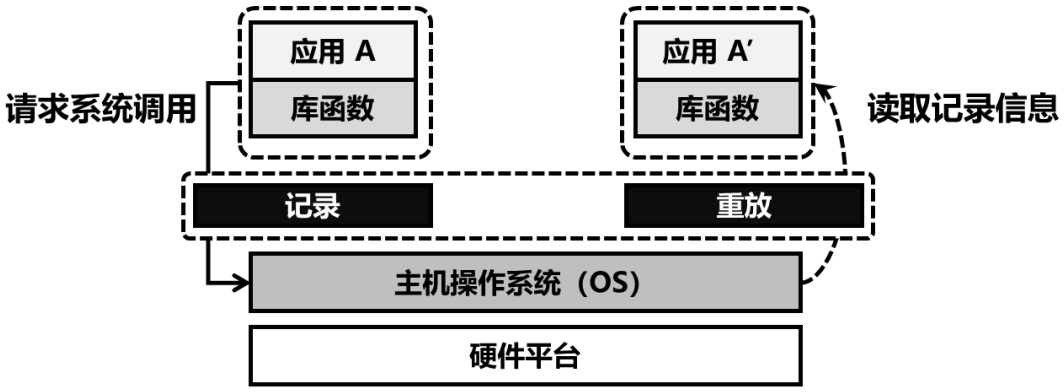


图 2.4 确定性重放的记录和重放流程示意

确定性执行技术一般通过构建一个确定性的执行环境来实现，为此需要在执行过程中查找不确定性因素，或者避免引入不确定性的操作。但不确定因素不是记录在存储空间中，而是通过拦截、替换、改写、隔离等操作来避免，以此实现具有普遍性的确定性执行。如图 2.5 所展示的一个确定性执行示意模型，从操作系统的进程、网络设备等中消除不确定性因素：拦截来自处理器的特权调用、为进程配置唯一的标识符（PID 等）、隔离文件系统等，从而确保程序强制确定性地执行。

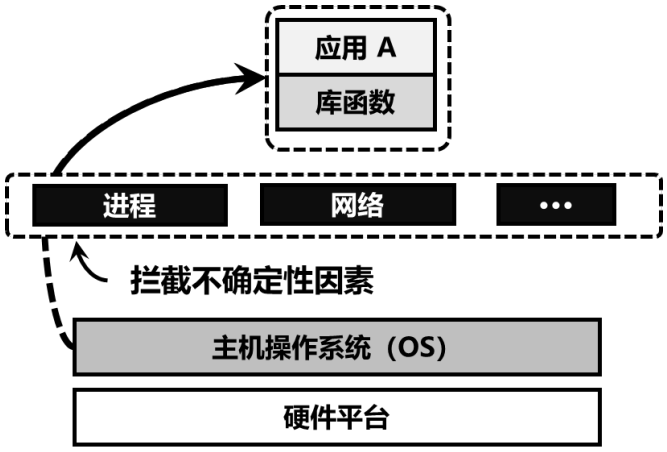


图 2.5 确定性执行的流程示意

关于可重现性的研究体现在以下几种应用场景中：（一）在并行程序调试过程中，循环执行并行程序往往具有随机性，无法得到确定的结果。开发人员使用确定性重放技术在用户系统上记录程序的执行踪迹（Execution Trace），并在开发系统中重现程序崩溃前的状态，为开发人员进行软件漏洞分析提供依据；（二）在分布式系统中，需要可重现性确保副本行为相同，满足分布式一致性的需求[70]；（三）在机器学习、科学计算、大数据分析等计算任务中，模型权重训练过程存在随机性。确定性重放技术可以记录程序执行过程中的执行踪迹，反应模型训练过程中性能变化趋势，帮助研究人员寻找性能变化原因。

本文设计并实现的容器化可重现方法属于确定性执行的范畴，同样通过避免在进程执行中引入不确定性因素来保证程序的可重现性，并使用容器对其进行包装。接下来将介绍常见的不确定因素来源，通过追踪进程实现的确定性执行方法，以及使用容器化技术的优势。

2.2.1 不确定因素来源

确定性代表着数据流确定性[71]，即在特定机器上，程序每次执行均会得到相同的返回值。但在程序实际执行环境中往往会存在多种不确定因素，导致循环执行程序的结果不同。本节对不确定因素的实际来源进行分析，列出已知的导致程序不确定性的来源，在后续章节会针对下列不确定性因素设计并实现可重现方案。

1. 系统调用

系统调用（System Call）被设计用来实现系统功能，由操作系统核心提供，运行于内核态，为用户空间进程和硬件设备的交互提供接口。与当前操作系统状态有关的系统调用（如获取进程标识符$getpid$、获取计时器值$getitimer$等）均具有不确定性。

2. CPU 指令

小部分 CPU 指令本身也是不确定的，尤其是特权指令，会在用户态引发异常。

3. 异步事件

外部与异常机制是 CP 对信号做出的一种反应，通常是异步的，是不确定性的主要来源。硬件中断是由硬件设备触发的，发生特定事件时与内核交互，如网

卡接收到数据包时触发硬中断。处理中断与异常会导致不确定的数据流结果。

4. 共享内存竞争

多核架构下的并行程序对共享内存的访问频率很高，来自不同进程的访存指令访问共享内存时会产生新的不确定性。

5. 线程间同步

如用于实现信号锁定的系统调用$futex$，线程间的调度机制也会带来不确定性因素。

6. 文件系统访问

文件系统访问也会带来大量不确定性因素，如返回文件夹名称操作。

2.2.2 追踪进程系统调用 ptrace

确定性执行工具往往需要追踪进程中的不确定信息，如之前

Linux 内核提供了一个追踪进程系统调用函数$ptrace$，可用来监视并拦截进程执行中的所有系统调用。$ptrace$实现在用户层利用一个进程（父进程）监视另一个进程（子进程或者线程）的执行。父进程可以拦截并修改子进程的系统调用、读取和写入子进程内存和寄存器等。通过 ptrace 可以用来实现断点调试和系统调用跟踪等目的。

ptrace 的函数定义如下：

$$\#include\ <sys/ptrace.h>$$

$$Long\ ptrace(enum\ \_ptrace\_request\ request, pid\_t\ pid, void\ *\ addr, void\ *\ data\ );$$

被追踪的子进程中的信息转换为 SIGCHLD 信号，传递给执行追踪任务的父进程。$ptrace$ 支持以下请求：

（1） 附加到跟踪进程上，或者从正在跟踪的进程中分离；

（2） 读取或写入进程的内存、已保存的寄存器状态等信息；

（3） 持续上述过程，直到读取到特定系统调用或者返回信号。

$ptrace$ 被集成在 GDB、Strace、Ltrace 等系统调试工具中，但是$ptrace$ 在使用中具有如下限制：

（1） 跟踪多个进程和线程会造成额外开销的增加；

（2） 读取子进程的内存和寄存器相关信息会增加巨额的额外开销，是直接

读取内核信息的 10 倍到 100 倍左右；

（3） 为了跟踪进程，跟踪程序必须成为被跟踪程序的父进程。为了附加到一个已经运行的进程，跟踪程序会破坏被跟踪程序的数据沿袭。

在确定性重放工具的设计中，首先使用 ptrace 监视用户系统执行程序的所有系统调用，判断为可重现的系统调用被允许通过；若系统调用存在不确定性因素，则对其进行拦截和更改，如包装系统调用或传递确定性的信息（如替换时间调用为确定的信息）。

2.2.3 容器化技术

容器化技术能够封装或隔离应用及其整个运行时环境，帮助开发人员在不同开发环境中快速部署应用，同时还可保留应用的全部功能。在兼容底层系统架构的情况下，与虚拟机相比，容器具有更好的可移植性和更少的开销。相较于传统的虚拟机，容器化技术无需使用虚拟机监控程序模拟硬件，共享同一个操作系统内核，所以部署应用所需要的开销更小。

早期的容器化技术基于 Unix 环境下的 chroot 命令，结合控制组（cgroups）和命名空间（namespace）等 Linux 内核特性，成为了构建容器的框架。

Chroot 源于早期的 Unix 系统，能够作用于正在运行的进程和它的子进程，改变进程的根目录。chroot 将进程放到指定的根目录执行，且该进程无法访问指定根目录之外的文件，实现了资源的隔离。

Cgroups 能够控制和限制一个进程或多组进程的资源使用，例如 CPU 时间、系统内存、网络带宽或这些资源的组合。通过使用 cgroups，系统管理员可以细粒度地控制分配、优先级、拒绝、管理和监控系统资源。在任务和用户之间合理分配硬件资源，提高整体效率。

Namespace 定义了 Linux 系统中标识符的可见范围。可以通过 namespace 将系统组件进行资源隔离，如主机名和域名（UTS）、根文件系统（Mount）、进程标识符（PID）等。

上述系统特性成为了 Linux 容器（Linux containers，LXC）技术的核心，并基于 LXC 构建了 Docker 容器。Docker 是目前最为常用的容器技术，包括可运行和构建新的分层镜像的简单命令行界面、服务器守护进程、含有预构建容器镜像的库以及注册表服务器概念。通过综合运用这些技术，用户可以快速构建新的

分层容器，并轻松地与他人共享这些容器。Docker 提供了更加完善的文件系统隔离机制，能够在容器中封装完整的系统镜像，并且只增加了少量的额外性能开销。

随着近年来 RISC-V 特权架构规范的更新（增加了对 Hypervisor 的支持），以及RISC-V软硬件生态的发展，RISC-V设备已经实现了对Docker的支持[72][73]。作为一种轻量级的虚拟化方法，Docker 和 LXC 容器更适合在资源受限的微处理器中使用。

2.3 本章小结

本章分别介绍了 RISC-V 指令架构和可重现性研究。RISC-V 指令架构是本文面向的研究平台，其设计模式与 x86、ARM 等传统指令集相比有较大不同。RISC-V 的特权架构为处理系统中断、实现虚拟化拓展提供了支持。可重现性中主要介绍确定性执行的工作原理和实现方法，同时介绍了 Linux 容器和 Docker 相关知识，为后续方案的设计与实现提供理论依据和方法支撑。

第 3 章 用户进程确定性执行方法设计

本章提出了用户进程确定性执行方法的设计与实现。针对数据竞争、系统调用、信号、共享内存等不确定性来源，通过追踪进程拦截不确定性，实现程序的确定性执行。本方案的结构框图如图 3.1 所示：

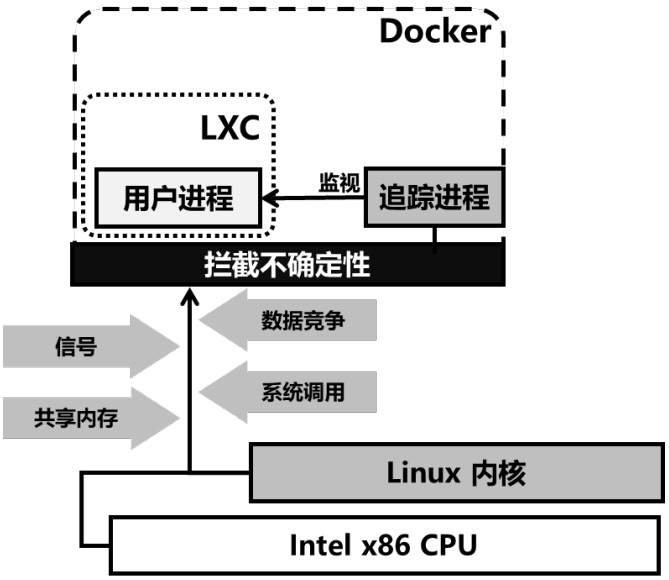


图 3.1 用户进程确定性执行方法设计框图

3.1 方法设计

可重现方法部分基于确定性重放工具来实现，但对进程、系统的记录和重放会附加大量的额外开销。在 2.2.1 章节的不确定性来源分析中观察到，具有不确定性的 CPU 指令较少，而不确定性因素的主要来源在于系统和软件层面[7]。因此首先考虑消除 CPU 指令以外的不确定性因素，之后再设计方案对 CPU 指令进行拦截。

基于此提出一种用户进程确定性执行方法，在容器内，监视并拦截用户进程的所有非确定性来源，记录所有进入容器边界的输入，并通过改写、替换非确定性因素来源的数据来保证容器内程序的确定性执行结果。

具体实现中，使用$ptrace$ 拦截容器中执行的所有系统调用。$ptrace$ 允许父进程追踪另一个子进程的系统调用，并读取和写入子进程的内存和寄存器（但是每次拦截事件需要额外的上下文切换）。基于此，容器中可重现的系统调用被允许通过，而不具有可重现性的系统调用被重现包装、替换。对于共享内存访问等造成的不确定性，下边会详细介绍其调度机制。

通过这种设计，容器内可重现性方案的问题核心在于监视并拦截容器和操作系统内核之间的不确定性因素，而不确定性因素的主要来源是数据竞争、系统调用、异步事件的时序和共享内存。

3.1.1 避免数据竞争

不同线程对同一内存位置的数据访存竞争可能会导致线程的阻塞或死锁，而线程调度过程中产生的上下文切换过程是不确定性因素来源之一。如果上下文切换发生在程序执行中不正确的位置，会出现由于数据竞争产生程序错误，并会影响程序的可重现执行[75][76]。传统方法使用的的线程调度方法[11][47][57]是保证同一时刻只运行一个线程，这种线程调度方法适用于低并行度的工作负载，但是对于具有持续高度并行性的工作负载，程序执行速度会大幅下降；或者假设程序执行中不存在竞争情形[26] [55][61]，但是显然这种特殊情况无法解决并行程序执行中的问题。

本文为追踪进程设置了一个线程调度器，用来允许一定数量的进程并发运行，并减少创建、销毁、调度线程带来的不确定性和额外开销。追踪进程在执行系统调用、进程创建和进程销毁时做出调度决策，避免线程堵塞。调度器由三个队列（Queue）组成，分别是并行队列（Parallel Queue）、等待队列（Runnable Queue）和阻塞队列（Blocked Queue）。并行队列包含当前正在并发运行的线程，等待队列和阻塞队列包含当前需要调度以进行顺序系统调用执行的线程。如图 3.2 所示。

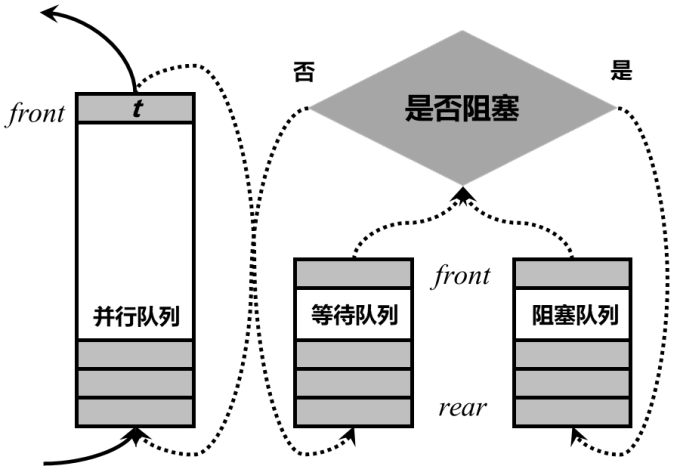


图 3.2 线程调度器

在起始状态，线程被创建之后进入并行队列的后端，并顺序执行。当并行队列前端的进程需要进行系统调用时，追踪进程接收到执行系统调用的请求，同时控制调度器将线程 t 移动到等待队列后端。等待队列前端的线程被允许执行下一个系统调用，并进行判断，如果这个系统调用不会造成阻塞；则线程返回到并行队列；如果将会发生阻塞，则将线程移动到阻塞队列的后端，并在之后重新尝试执行该调用。然后查询阻塞队列前面的线程，并判断其系统调用是否造成阻塞，并根据其结果相应地移动到并行队列或阻塞队列。

在具体实现中，对于线程 t 中可能造成阻塞的系统调用 s 进行判断，若 s 可以顺利执行且不会造成阻塞，则立刻执行线程 t，并将 s 从原有队列中删除，移动到并行队列；若 s 需要等待另一线程中的某个事件完成，则将 p 暂时移动到阻塞队列中。

3.2.2 系统调用

系统调用通过读写寄存器和内存将数据返回到用户空间，返回的信息是不确定性因素的主要来源。$ptrace$ 系统调用允许父进程（追踪进程，Tracer）监督其他子进程（Tracee）和线程的执行，并追踪子线程$entry$ 或$exit$ 系统调用的时间节点。当子线程进入内核执行系统调用时，它会被挂起并通知父进程；当容器选择再次运行该线程时，系统调用将完成，再次通知父进程，并记录系统调用结果。当$ptrace$ 追踪到需要被拦截的系统调用时，会替换带有不确定性的信息。例如 Linux 用户进程可以通过$getrandom$ 系统调用或从特殊的$/dev/random$ 或

$/dev/urandom$ 文件中读取随机数据，会为程序带来不一致的结果。容器需要使用$ptrace$拦截 $getrandom$ 系统调用，并用简单的 LFSR 伪随机数生成器生成的值填充指定的用户缓冲区。同样，$/dev\ /random$ 和$/dev/urandom$ 同样从伪随机数生成器中读取数据，从而避免随机数造成的不确定性。

在追踪进程的工作过程中，首先需要将子线程附加到跟踪线程上。附加命令和后续命令是针对单线程的程序，在多线程程序中，子线程以线程组的方式，每个线程可以单独附加到不同的跟踪程序，或者。$Ptrace$ 命令发送到特定的跟踪对象的调用方法为：

$$ptrace（PTRACE\_foo, pid, \ldots）$$

其中pid 是被追踪子线程的标识符。

如上节所述，可以通过一次只调度一个线程来避免数据竞争。但是，如果内核中的系统调用阻塞，容器必须尝试调度其他应用程序线程在阻塞中的系统调用完成时运行。正在运行的线程可能访问系统调用的输出缓冲区并与内核对该缓冲区的写入竞争。为了避免这种情况，容器将系统调用输出缓冲区重定向到每个线程的临时“暂存内存”。当$ptrace$ 追踪到一个阻塞系统调用完成的事件时，容器将暂存缓冲区内容复制到真正的用户空间目标，此时没有其他线程在运行，从而消除了数据竞争。

图 3.3 展示了拦截一个简单的$read$系统调用的过程，其中灰色框代表内核代码。 重放过程中，当下一个要重放的事件是应被拦截的系统调用时，在系统调用指令的地址处设置一个临时断点（记录在父进程中）。我们使用$ptrace$ 运行子线程，直到它遇到断点，并移除断点，将程序计数器进行到系统调用指令之后，并执行记录的寄存器和内存信息更改。这种方法最大限度地减少了容器和子线程之间的上下文切换次数。

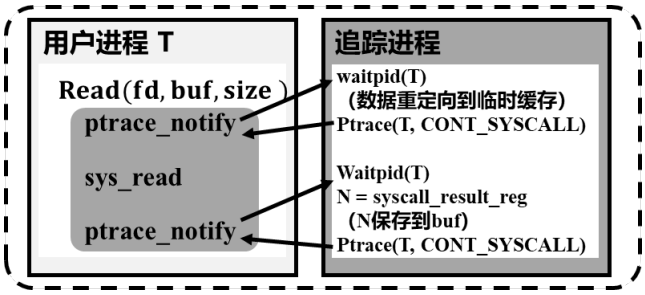


图 3.3 简单的拦截系统调用模型

3.2.3 信号

信号是不确定性的来源之一，可重现性需要支持同步信号和异步信号的拦截。对于受时钟信号控制的同步事件，如通过系统调用请求的时钟信息，容器将生成信号的时钟信号调用转换为阻止用户进程的暂停系统调用。然后，调用编排好的模拟时钟信号处理程序，得到模拟时钟数据，并通过$ptrace$ 向用户进程返回信息。之后退出暂停调用，恢复执行用户进程。在程序执行过程中，时钟调用始终被隔离在容器之外，容器内的用户进程只能调用模拟时钟信息。

对于外界的异步信号，可重现性要求记录执行阶段的线程收到的信号顺序，并确保在重放阶段，程序重现与记录阶段传递信号时完全相同的状态。之前的工作[11][60][62]使用 CPU 硬件性能计数器记录程序信号。要求每次执行给定的用户空间指令序列都会改变计数器值，该值仅取决于指令序列，而不是用户空间不可见的系统状态（例如缓存的内容，页面的状态表或推测的 CPU 状态）。

但是本文的可重现容器使用了纯软件方法实现可重现性，且后续进行了 RISC-V 架构上的移植优化，并没有使用 CPU 硬件性能计数器的方法。另一种方法是通过可重现的逻辑时钟完全重现异步信号的事件。但是，可重现性容器不支持在用户进程之间发送信号，且部分用户进程向自身传递的信号是自然可重现的，如$SIGSEGV$、$SIGILL$ 和$SIGABRT$ 可以在可重现状态下停止程序执行。

3.2.4 共享内存

由于同一时刻只执行一个线程，只要共享内存仅由子线程写入，容器中就避免了部分共享内存上的竞争问题。但是，可重现的进程可以与其他进程甚至内核设备驱动程序共享内存，其中代码存在执行与子线程访问竞争的写入的可能性。目前的 Linux 应用程序中，共享内存竞争问题仅在四种常见情况下发生：应用程序与 PulseAudio 守护程序共享内存、应用程序与 X 服务器共享内存、应用程序与内核图形驱动程序和 GPU 共享内存、以及 VDSO 系统调用。可以通过自动禁用 PulseAudio 和 X 共享内存的使用和禁用 GPU 应用程序来避免前三个问题 。

VDSO 系统调用是一种 Linux 优化，它在用户空间中实现一些常见的只读系统调用（例如$gettimeofday$），部分通过读取与内核共享的内存并由内核异步更新。可以通过修补它们的用户空间实现来禁用 VDSO 系统调用 ，以执行等效的真实系统调用。

3.2 系统调用拦截优化

上一节中的方法能够实现可重现性的功能，但在实验中会增加额外的性能开销，而主要原因在于$ptrace$ 进程跟踪过程中上下文切换的次数过多。如图 3.2 所示的系统调用拦截过程中，对于每个子线程系统调用需要执行四次上下文切换：两个$ptrace$_notify，每个都需要从子进程到容器的上下文切换。对于常见的系统调用，例如$gettimeofday$ 或从读取缓存中数据，即使是单个上下文切换的开销也比系统调用本身的开销大得多。为了尽可能减少上下文切换带来的性能损失，必须在处理某些常见的系统调用时避免上下文切换。

因此，需要设计系统调用过程中的性能优化方案，减少性能损失。在实现中，通过将一个系统调用库注入到记录的进程中，拦截常见的系统调用，在执行系统调用的同时避开$ptrace$ 的陷阱（Trap），并将结果记录到容器的专用缓冲区，并设立定时机制，定期将缓冲区数据传递到其跟踪进程中。

3.2.1 拦截系统调用库

常用的在进程中拦截系统调用的技术是，使用动态链接在进行系统调用的 C 库函数上插入包装函数。在实践中，由于部分应用程序直接进行系统调用，并且由于 C 库函数的版本变化，这种方法具有很大的局限性。

相反，当子进程进行系统调用时，容器通过$ptrace$ 监视子进程，并调用设置好的拦截库拦截、重写系统调用指令。常用的系统调用指令后面通常伴随一些已知的、固定的指令序列。例如，许多系统调用指令后跟随一个测试调用结果的指令序列 ：cmpl $0xfffff001, %eax。在拦截库中添加对应的的系统调用执行后指令，在$ptrace$ 监视到系统调用执行后，容器将系统调用指令及其后续指令替换为相对应的确定性系统调用。

理论上，所有系统调用指令都可以重定向到拦截库，但为简单起见，拦截库只包含了最常见的系统调用。对于其他系统调用，容器通过执行常规的$ptrace$ 监视并拦截系统调用。

3.2.2 选择性拦截

$ptrace$ 系统调用监控会触发所有系统调用的陷阱（Trap），但是我们的拦截

库需要避免触发特定系统调用的陷阱。现代 Linux 内核支持选择性地生成$ptrace$陷阱：seccomp-bpf。一个进程可以将一个以字节码表示的 seccomp-bpf 过滤函数应用于另一个进程，作为过滤器使用；然后，对于容器内用户进程执行的每个系统调用，内核调用过滤器，并传入用户空间中包括程序计数器的寄存器值。过滤器的结果指示内核是否允许系统调用，或者返回的程序执行失败信息，或者终止目标进程。过滤器执行的开销可以忽略不计，因为过滤器直接在操作系统内核中运行，并在大多数架构上编译为本机代码。

图 3.4 说明了使用进程内系统调用拦截记录一个简单的读取系统调用。实线框代表拦截库中的代码，灰色框代表内核代码。容器将一个特殊的内存页注入到每个被追踪的子进程的固定地址中（紧接在$execve$系统调用之后）。该页面包含一个系统调用指令——“未跟踪指令”。在程序计数器没有位于未跟踪指令处时，容器对每个记录的进程应用 seccomp-bpf 过滤器，为每个系统调用触发 $ptrace$ 陷阱。每当拦截库需要执行未跟踪的系统调用时，都会触发该指令。

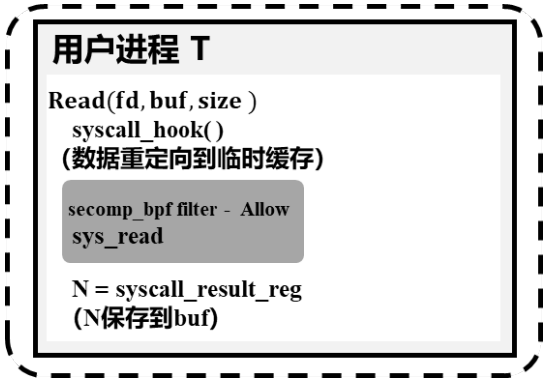


图 3.4 使用拦截库机制拦截系统调用

3.2.3 检查系统调用阻塞

一些常见的系统调用有时会导致阻塞（例如在空管道上执行读取操作）。因为容器同一时刻仅运行一个子线程，如果一个线程在没有被容器拦截到的情况下进入容器并阻塞系统调用，该线程将挂起并可能导致整个记录过程死锁（例如，如果另一个被跟踪的子线程即将写入管道）。每当未跟踪的系统调用阻塞时，需要容器挂起跟踪线程，以确保容器可以调度不同的跟踪线程。通过使用 Linux perf 事件系统来监控 PERF_COUNT_SW_CONTEXT_SWITCHES 配置。内核每次从 CPU 内核调度线程时都会触发这些事件之一。拦截库为每个线程监视这些事件，

并在每次事件发生时请求内核向被阻塞的线程发送信号。这些信号触发容器的 $ptrace$ 进程跟踪，同时阻止线程进一步执行。为了避免假信号造成的影响（例如，当线程由于正常的时间片到期而被取消调度时），该事件通常被禁用并在可能阻塞的未跟踪系统调用期间显式启用。尽管如此，在启用和禁用事件之间的任何时间点都可能发生错误的转换。我们通过仔细检查跟踪状态来处理这些边缘情况。

图 3.4 说明了通过系统调用拦截$read$系统调用时发生阻塞的情况。内核调度线程触发事件，向线程发送重新调度的信号，中断系统调用，并向容器申请调用$ptrace$ 跟踪进程。跟踪进程记录一个被拦截的系统调用在线程 T 中被中断，然后检查阻塞系统调用中的任何跟踪线程是否已经执行系统调用退出并调用 $ptrace$。在这个例子中 T2 完成一个（未被拦截的）阻塞$futex$ 系统调用。

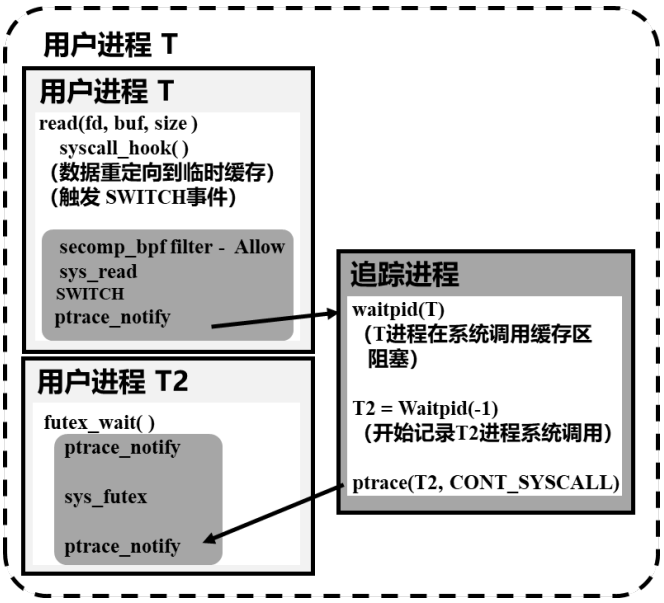


图 3.5 拦截阻塞的系统调用

3.3 本章小节

本章设计并实现了一种基于追踪进程的用户进程确定性抽象方法。该方法研究用户进程执行过程中的不确定性因素来源，结合确定性执行、记录重放等工作中的部分先进技术，设计并实现了一个用户进程确定性抽象。

第 4 章 基于容器的可移植性优化

为了避免底层微架构对可移植性的影响，同时消除来自文件系统的确定性，本章设计了一种 2 层嵌套的容器机制隔离用户程序，具体设计如下：

首先，为了满足任意应用程序的可重现性，且不需要修改内核或记录完整虚拟机状态，设计并实现了基于控制组$cgroups$、用户命名空间$Namespace$等技术的容器化方案。在纯软件命名空间中，隔离容器内进程，保证每个进程组有独立的 PID 等标识符，将用户进程确定性执行程序与外部的程序和文件隔离起来。

其次，使用可移植性更好的标准容器 Docker 提供更强的主机文件系统隔离，控制容器输入，提供文件目录系统的确定性。

4.1 容器架构

在系统调用之外，还存在来自进程标识符和文件系统不确定性因素，需要设计容器化方案隔离追踪进程。现有的容器技术（如 Docker）并不具备可重现性。Docker 无法提供确定性的输出，受到主机操作系统和处理器微架构的等各方面的影响，因为主机操作系统和处理器微架构的许多细节对于在容器内的用户进程是直接可见的。使用虚拟机可以提供更强的硬件抽象，隔绝部分硬件的影响，但缺乏确定性，而且对整个虚拟机实现确定性重放会增加大量的额外开销。

为了实现可重现性的目标，本文结合命名空间和 Docker 容器技术，设计并实现了 2 层嵌套的容器方案，如图 4.1 所示。在内层容器中，为了满足任意应用程序的可重现性，且不需要修改内核或记录完整虚拟机状态，设计并实现了基于控制组 cgroups、用户命名空间 Namespace 等技术的容器化可重现方案。在纯软件命名空间中隔离容器内进程，保证每个进程组有独立唯一的 PID 等标识符，将确定性执行程序与外部的程序和文件隔离起来。通过内层容器的包装，能够规避来自容器中进程的进程标识符等因素的影响。

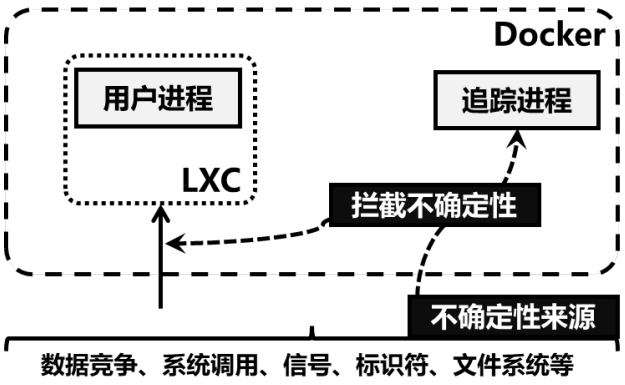


图 4.1 LXC 和 Docker 嵌套的容器设计

在外层容器中，使用标准化的容器工具 Docker，将调用$ptrace$ 的跟踪进程、内层命名空间容器、拦截调用库等功能模块封装到镜像文件中，利用$ptrace$ 监视并拦截从内层容器中进行的所有系统调用。Docker 提供了更加强大的文件隔离机制，并且提供了可移植性的优化，能够在不同机器间快速部署可重现性环境。在不考虑拦截 CPU 系统调用的情况下，使用 Docker 可以将上述方案快速部署到 RISC-V 架构上。

2 层嵌套的容器化方案这一功能实现了可重现性的要求：纯软件用户空间解决方案，支持未修改的二进制文件，屏蔽了底层微架构对用户进程的影响，并且不需要特权（root）访问。

4.1.1 进程标识符

由于容器内的用户进程通过命名空间隔离，容器内的进程只能接收到独立于容器外部环境的唯一进程标识符（PID）。用户进程不能命名容器外的任何进程。由于用户进程的创建和终止顺序已经被确定性的记录下来，并且 Linux 在每个命名空间中遵循顺序分配 PID 的原则，容器内的 PID 自然是确定性的。

4.1.2 文件系统

文件系统是不确定性因素的来源之一。为文件和目录提供可重现抽象的第一步是隔离用户进程拥有的主机文件系统的视图，通过 chroot 系统调用更改根目录地址完成。其次，基于命名空间的容器嵌套在 Docker 等标准容器中，可以更好地隔离主机系统上的文件。

Linux 命名空间控制着命名空间内部的系统标识符（如 uid、gid 等）到主机系统上标识符的映射，此映射也是容器输入的一部分。默认情况下，将主机系统上的用户映射到容器内的超级管理员 root 权限上，并将所有其他用户信息映射到 nobody 和 nogroup。返回目录条目的顺序由文件系统实现控制。为了使 $getdents$（用于获取目录条目）等系统调用可重现，容器在将目录条目返回给用户进程之前按名称对其进行排序。

$read$ 和 $write$ 系统调用是不可重现的，因为它们读取/写入的字节数可能比实际请求时读取更少字节的数据，尤其是在访问管道时经常出现这种情况。为了使这些系统调用在所有情况下都可重现，容器会自动重新执行部分$read$和$write$的指令，直到它们处理请求的字节数一致，或者读取返回 EOF。这是通过递减用户进程程序计数器以重新运行系统调用指令，并调整参数来实现的，例如，保证当前$read$在前一次读取结束的地方继续。

索引节点（Index nodes）是文件系统中挂载文件或目录的唯一标识符。$stat$ 系列系统调用向用户进程返回索引节点，并且仅仅返回一个固定值是不够的，因为许多用户进程通过比较索引节点的值来快速识别相同的文件。容器设计并实现了一个从不可重现的物理索引节点到可重现的虚拟索引节点的映射。需要特别注意确定何时创建新文件 f，因为操作系统可能会为f 回收物理索引节点，但容器必须分配新的虚拟索引节点以保持可重复性

文件时间戳为用户进程提供了一个时间顺序，可用于重构一个不可重现的时钟。容器虚拟化了文件时间戳。在 Linux 上，每个文件或目录都有三个关联时间：最后一次内容修改时间（mtime）、最后一次访问时间 （atime）和最后一次内容或元数据修改时间（ctime）。在容器中，我们总是将 atime 和 ctime 强制定义为 0。但是，在许多程序中只返回一个固定的 mtime 值会违反程序完整性查验。例如，从 GNU Autotools 配置通过创建一个新文件来检查时钟偏差，然后将其 mtime 与现有文件的 mtime 进行比较，如果 mtime 没有意义，则会引发错误。

4.2 RISC-V 容器化可重现方法

上一节设计的容器化可重现方案中，基于命名空间的容器嵌套在 Docker 容器中，以提供与主机更强的文件系统隔离。但是上述原型设计是在 x86 主机上构建的，且并未考虑不确定性的 CPU 指令影响。因此在设计基于 RISC-V 的容器

化可重现方法时，需要拦截不确定性的 RISC-V CPU 指令。本章将首先讨论在 RISC-V 架构下的 Linux 系统中使用 Docker 的问题，其次介绍了一种基于 QEMU 的二进制翻译修改 CPU 指令的方法。

4.2.1 RISC-V 容器

容器化技术能够封装或隔离应用及其整个运行时环境，帮助开发人员在不同开发环境中快速部署应用，同时还可保留应用的全部功能。相较于传统的虚拟机，容器化技术执行所需要的开销更小。Docker 是目前最为常用的容器技术。随着近年来 RISC-V 软硬件生态的发展，RISC-V 架构已经支持使用 Docker。

在 RISC-V 架构上使用 Docker，需要在 Linux 系统中安装 Golang，下载并构建 Docker 软件包，然后在这个环境中运行和构建容器。本文通过 Gem5 全系统模拟器构建了一个仿真 RISC-V 平台，并在硬件架构上安装了 Linux 内核 v5.5.0 版本的 Debian 10（简称为 Buster）操作系统发行版。具体细节在第 4 章实验设计与分析。

4.2.2 拦截 RISC-V CPU 指令

不可重现的 CPU 指令无法通过$ptrace$ 拦截，现有的规避 CPU 指令不确定性的方案往往是通过硬件辅助的程序计数器实现的，但是当前的 RISC-V 硬件不支持捕获所有不可重现的指令。为了在 RISC-V 架构上实现纯软件的容器化可重现方法，参考了 x86 指令集下利用编译器插入代码的确定性重放工具，在此基础上利用 QEMU 中的二进制翻译修改 RISC-V 指令，实现对不可重现性指令的拦截与修改，系统的设计框图如图 4.2 所示。

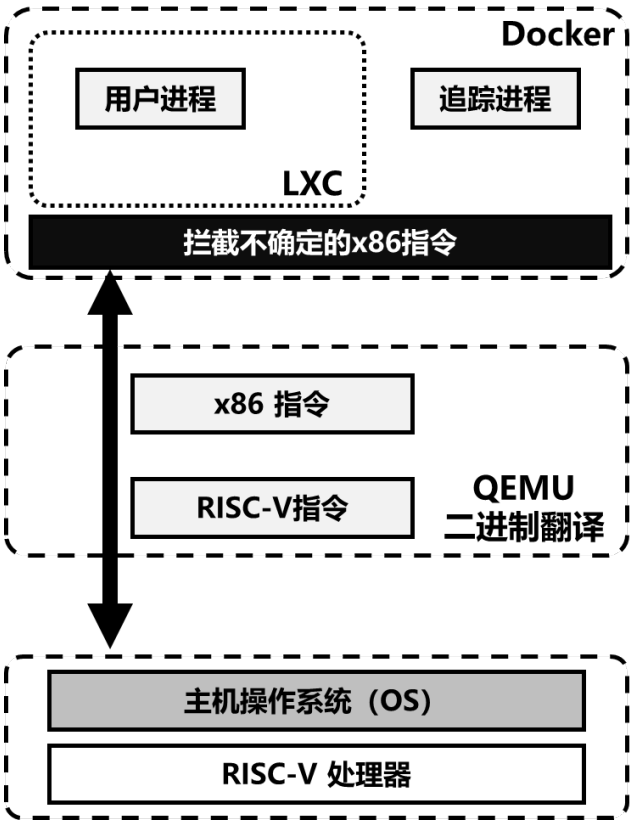


图 4.2 基于 QEMU 的二进制翻译工具拦截 RISC-V CPU 指令

现有的确定性重放工具主要针对 Intel x86 架构下的 CPU 指令的拦截[6]，如 **rdtsc** 和 **rdtscp** 指令。通过$prctl$ 系统调用返回当前周期的计数，使用指令的线性函数覆盖它们的不确定结果。其他不可重现的指令包括 **TSX** 指令、**rdrand**、**rdseed** 和 **cpuid** 等。

QEMU 中的二进制翻译通过内核代码的 TCG（Tiny Code Generator）模块对异构应用的二进制代码进行翻译和转换。异构文件在执行时，通过$binfmt_mis$识别可执行文件格式并传递至 QEMU。$binfmt_mis$是 Linux 内核的一种功能，它允许识别任意可执行文件格式，并将其传递给特定的用户空间应用程序，如仿真器和虚拟机。QEMU 将注册的异构二进制程序拦截、转换成本地指令架构代码，同时按需从目标架构系统调用转换成宿主机架构系统调用，并将其转发至宿主机内核。TCG 定义了一系列 IR（Intermediate Representation），将已经翻译的代码块放在转换缓存中，并通过跳转指令将源处理器的指令集和目标处理器的指令集链接在一起。当程序执行时，存放于转换缓存中的链接指令可以跳转到指定的代码块，目标二进制代码可不断调用已翻译代码块来运行，直到需要翻译新

块为止。在执行的过程中，如果遇到了需要翻译的代码块，执行会暂停并对需要进行二进制翻译的源处理器指令集进行转换和翻译，并存储到转换缓存中。图 4.3 为 TCG 的工作示意图。

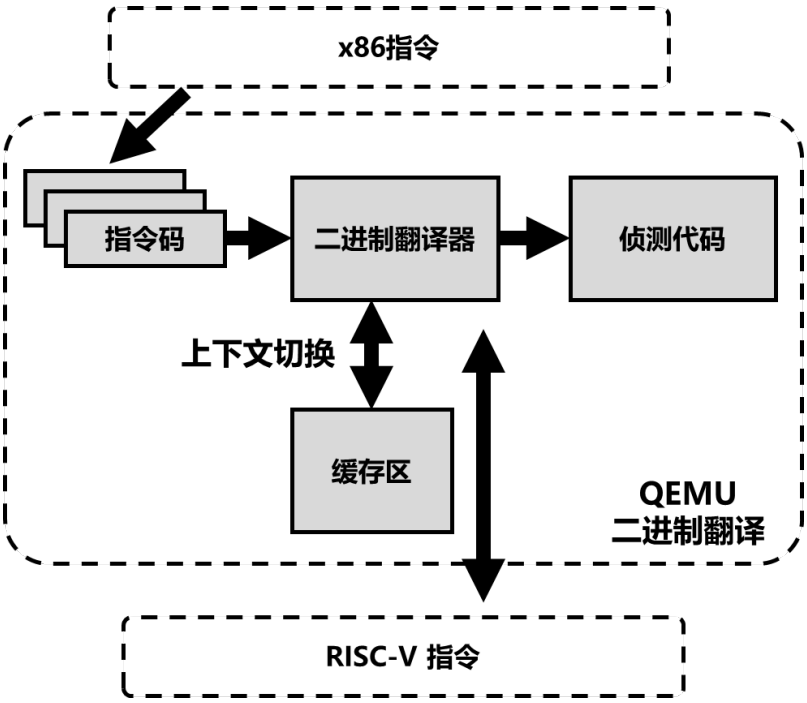


图 4.3 QEMU 中 TCG 模块结构图

4.3 本章小结

本章设计并实现了用户进程确定性执行基于容器的可移植性优化方法。为了在 RISC-V 架构下实现用户进程确定性抽象，该方法使用 Docker 对确定性抽象进行封装，实现与 LXC 嵌套的二层容器，消除底层微架构对可移植性的影响，同时隔绝了来自文件系统的不确定性影响；在容器内部，利用动态二进制翻译技术替换用户进程执行过程中具有不确定性的 CPU 指令。容器化可重现方案避免了对程序和操作系统内核的修改，减少了部署和维护的成本，同时进一步保证了用户进程的确定性。

第 5 章 实验设计与分析

本章首先在 Gem5 模拟器上部署了 RISC-V 架构芯片的多核全系统仿真，并在此架构上加载 Linux 系统，搭建对上述容器化可重现方案的功能、性能测试平台；然后，介绍实验测试使用的基准测试集；最后，与现有记录重放工具进行对比，设计实验完成容器化可重现方案的功能、性能测试，并对实验结果进行分析和评价。

5.1 RISC-V 仿真平台

在 RISC-V 硬件平台上实现体系结构实验中的思路需要增加额外的部署成本，因此本文选择使用性能优良的软件模拟器来验证容器化方案。目前有多种软件模拟器支持模拟 RISC-V 硬件平台，功能级（Function-Level）仿真有 Spike、QEMU、FireSim、RV8；寄存器传输级（Register-Transfer-Level, RTL）仿真器包括 Rocket、BOOM、Ariane；FPGA 级仿真模型 Rocket Zedboard 等。功能级仿真速度快、易于修改，但无法捕捉目标系统的时序。RTL 仿真速度慢、难以修改，但是可以精准模拟目标系统的时序周期。FPGA 仿真最准确和快速，但是需要较长的综合和布局布线时间，也更难以修改调试。

综合考虑，选择离散事件全驱动模拟器 Gem5 实现 RISC-V 平台仿真。Gem5 模拟器能够以全系统模式模拟多核 RISC-V 处理器[63-66]，支持模拟大部分 RISC-V 指令和系统调用，支持线程相关系统调用和同步指令，并能对多核条件下的程序设计和调试进行模拟验证。由于功能强大且开源开源，Gem5 被广泛用于体系结构领域的仿真及验证实验。用户可以通过 Python 接口选择不同型号的 CPU、系统模式和内存系统，以实现具有所需要的模拟处理器配置。同时为了保证仿真速度，Gem5 的关键性能模块通过 C++实现。对于每种指令集架构，Gem5 分别提供两种模拟模式：系统调用模拟 (System-call Emulation, SE) 和全系统 (Full System, FS) 模拟。之前的工作[63][64]保证 Gem5 能够在 SE 模式下支持模拟大多数 RISC-V 指令和系统调用。SE 模式能够快速实现用户工作负载的执行和分析。

此外，Gem5 可以通过 FS 模式准确模拟系统组件和硬件设备，并加载系统

软件（通常使用 Linux 内核）。 FS 模式支持包括虚拟内存、虚拟化、分布式系统、存储堆栈性能和网络相关等相关功能。Gem5-21.0 版本已经支持在模拟 RISC-V 硬件平台上运行的 Debian 10（简称为 Buster）发行版，Linux 内核版本为 v5.5.0。

本章在 Gem5 模拟器上模拟了一个 4 核 RISC-V 处理器最小系统，包括满足运行引导加载程序和 Linux 内核所需的最少硬件。由于 RISC-V 和 Gem5 的模块化设计，后期可以根据用户需要快速拓展支持的指令集模块和硬件设备。在 5.1.2 章中详细介绍了模拟的 RISC-V 目标系统。

在 Ubuntu 16.04 上，Gem5 模拟器上的全系统仿真步骤如下图 5.1 所示。首先，安装所有必需的依赖项；其次，建立 Gem5 文件夹，通过 Git 下载存储库；然后，配置 RISC-V 选项，编译 Gem5 源文件，构建 RISC-V 模拟平台；最后，对 Gem5 进行测试。

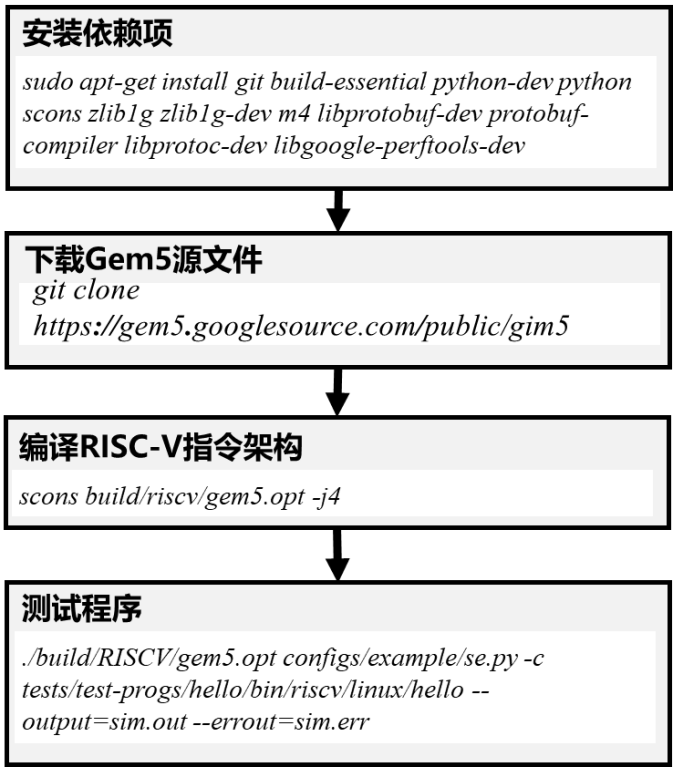


图 5.1 Gem5 模拟器安装及测试步骤

执行预编译好的测试程序，系统输出如下图 5.2 所示：

测试程序输出

```
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled Dec 25 2021 17:19:20
gem5 started Dec 28 2021 14:07:22
gem5 executing on llvm-vm, pid 61827
command line: ./build/RISCV/gem5.opt configs/example/se.py -c tests/test-
progs/hello/bin/riscv/linux/hello --output=sim.out --errout=sim.err

Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the address range
assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7000
**** REAL SIMULATION ****
info: Entering event queue @ 0.  Starting simulation...
info: Increasing stack size by one page.
Exiting @ tick 2988000 because exiting with last active thread context
riscv@llvm-vm:~/workspace/gem5$ cat m5out/sim.out
Hello world!
```

图 5.2 测试程序输出

5.1.1 RISC-V 目标系统构建

构建的目标系统是一个可以进行虚拟化拓展的基本 RISC-V 架构 Linux 系统。此目标系统具有一组核心硬件源，能够加载系统软件，例如带有引导加载程序和 Linux 内核的系统软件。目标系统包含两个主要子系统：CPU 和 HiFive 平台。在 CPU 子系统中，Gem5 添加了一个额外的 MMU 组件——PMA 检查器。HiFive 平台基于 SiFive 公司推出的 HiFive 系列开发板设计，包含最少的关键外围设备集合。内核本地中断控制器（CLINT）通过 MMIO 接口处理中断和定时器信号。平台级中断控制器（PLIC）负责根据优先级方案将来自外部源和外围设备的中断分发到硬件线程。为了启动内核并实现可用的操作系统，必须包含 UART 和 VirtIOMMIO 组件。其中，UART 提供交互式命令行终端，而 VirtIOMMIO 提供包含工作负载脚本和操作系统二进制文件的根文件系统。

Gem5 全系统仿真中由多个软件层组成堆栈。Gem5 全系统模拟模块包含具有所需配置的系统硬件模块。它还对硬件模块之间的交互进行建模。带有 RISC-V 指令集架构解码器的 CPU 模型处理来自操作系统层或用户应用程序的指令，这些指令可能处于不同的特权模式。Gem5 全系统模拟从解析 Python 配置 脚本和基于配置构建模拟器可执行文件开始。然后，模拟器加载引导加载程序和 Linux 内核来启动系统。当内核启动时，用户应用程序可以在后台或通过终端执

行。基于 Gem5 的 RISC-V 全系统模拟还支持不需要硬件辅助虚拟化的 RISC-V H 模式拓展。

5.1.2 HiFive 平台

在 Gem5 中，系统硬件配置被组织到称为平台（Platform）的容器类中。Platform 类是一个父类，包含一组标准化的外围设备和实用功能集合，能够以分层方式，扩展来自定义特定开发板的设置或系统。例如，在 ARM 指令架构下，常见的 Platform 类是 RealView；而在 X86 指令架构下，常见的 Platform 类是 PC。在 RISC-V 中，平台类被命名为 HiFive，对应 SiFive 公司的 HiFive 系列板卡。

HiFive 系列板卡的内存映射约定和外设地址是根据 SiFive U54MC SoC 数据表选择的。HiFive 平台包含可以添加其他非关键外围设备的最小外围设备集。这种基本配置不仅用于 HiFive 主板，还用于其他 SoC，例如 Kendryte K210。HiFive 平台被设计得易于扩展，只需对来自其他指令架构的设备进行少量的更改。提供了一个 PlicIntDevice 类以允许将外围设备轻松连接到 PLIC 中断控制器。HiFive 平台类中的一组实用程序功能还允许用户将新设备添加到列表中。

5.1.3 启动 Linux 系统

为了启动 RISC-V 架构下的 Linux 系统，需要使用 Berkeley 引导加载程序（Berkeley Bootloader，bbl）和 Linux 内核启动目标系统。为简单起见，该系统由四个 CPU 内核组成。系统成功启动后，登录系统并使用终端执行命令。Debian 系统中的命令可以正常运行。启动系统的步骤如下图 5.3：

图 5.3 Linux 系统设置及启动

在上述 RISC-V 硬件平台上，安装 Docker、QEMU 等依赖项。，并准备基准测试集，设计实验评估容器性能。

5.2 基准测试集

基准测试是计算机体系结构研究的定量基础，分别使用 2 种测试集进行实验：

（1） Linux 系统下的常用命令；

（2） PARSEC 基准测试集中的数据并行程序。

设计实验，对容器化可重现方法功能、性能进行测试。下边对这两个测试集进行介绍。

其次，实验使用前人工作中 2 种经典的记录重放工具 RR[7]，作为对照试验，分析性能的额外开销。RR 作为记录重放工具，执行程序分为记录和重放 2 个阶段。

5.2.1 软件包构建

实验设置选择 100 个 Debian 软件包的构建和编译过程作为实验对象，记录容器化可重现方法的运行时间，并与不使用可重现方法的基准测试进行相比。在确保软件包已经实现可重现性的前提下，通过记录其 5 次运行的平均时间来衡量其执行性能，并与不使用容器化可重现方法的基准测试情况进行比较。同时我们记录了所有软件包构建和编译过程中遇到的不确定性因素次数，计算了每个软件包构建的平均数，如表 5.1 所示：

表 5.1 测试程序及相应的测试参数

不确定性因素	平均次数
系统调用	823,324.25
数据竞争	396,474.88
信号	1,283.72
共享内存	6049.51
进程等待	113.98

此外，为模拟实际使用场景，本文在容器中运行常用的 Linux Shell 命令来测试方案的性能。使用的操作如表 5.2 所示：

表 5.2 测试程序及相应的测试参数

命令	描述	测试参数
date	多次运行查看容器内的时间信息	无
cp	复制 glibc 库中的数据	cp -a
make	构建 DynamoRio 程序	make -j4

（1） $date$ 查看容器内的时间信息。由于该命令执行时间很短，将其作为一个简单的功能测试模型。实际每次调用$date$ 命令获取的时间信息应当实时变化的，但是由于容器内的所有程序都具有可重现性，多次执行$date$ 可以得到确定的输出。

（2） cp 使用$cp\ -a$ 复制 glibc 2.31 标准库函数中的数据。glibc 所占磁盘空间大小为 732MB，共 15200 个文件。cp 是单线程程序，执行过程中需要大量请求读取等与文件系统相关的系统调用。

（3） $make$ 命令用来编译和构建各种程序，测试软件构建中的可重现性。实验使用$make\ -j4$ 命令（4 核）编译应用程序二进制翻译工具 DynamoRio v6.1.0。同时，DynamoRio[68]被设计用来修改程序运行时的执行指令。编译过程测试进程并行执行的性能。

5.2.2 PARSEC 基准测试集

PARSEC 基准测试集（The Princeton Application Repository for Shared-Memory Computers Benchmark）是一个由共享内存程序组成的基准测试套件[67]，收录了模式识别、数据挖掘等各个领域先进的多线程应用程序。PARSEC 内的每个应用程序都代表了其所在领域的最先进技术，并且为每个工作负载定义了 6 个不同规模的输入：

（1） $test$：一个很小的输入集，用于测试程序的基本功能；

（2）$simdiv$：一个非常小的输入集，但保证了与真实行为相似的基本程序行为，用于模拟器测试和开发；

（3）$simsmall$、$simmedium$和$simlarge$：规模不同的 3 个输入集，适用于使用模拟器进行微架构研究；

（4）$native$：用于本机执行的大型输入集合。

$test$ 和$simdev$ 仅用于测试和开发，用于研究的三个模拟器输入的规模各不相同，但更大的输入集包含更大的工作集和更多的并行性，并行程序中频繁地访存竞争带来了程序执行中的不确定性，适合测试容器的性能。

实验中使用的 PARSEC 版本为 3.0，选择其中 5 个具有不同特性的数据并行程序来评测容器的性能，所有测试案例均使用$simlarge$脚本运行。选用的程序如下：

（1） $blackscholes$：使用布莱克-舒尔斯模型（Black-Scholes Models）偏微分方程计算欧式期权组合的价格；

（2） $bodytrack$：使用粒子滤波（Particle Filter）追踪图像序列中无标记的人体 3D 姿势；

（3） $canneal$：使用缓存感知和模拟退火来最小化芯片设计时的布线成本；

（4） $fluidanimate$：模拟可压缩流体并用于交互式动画；

（5） $swaptions$：采用蒙特卡洛模拟（Monte Carlo Simulation）计算交换期权的价格。

所有基准测试程序的对比如下表 5.3：

表 5.3 基准测试程序性能比较

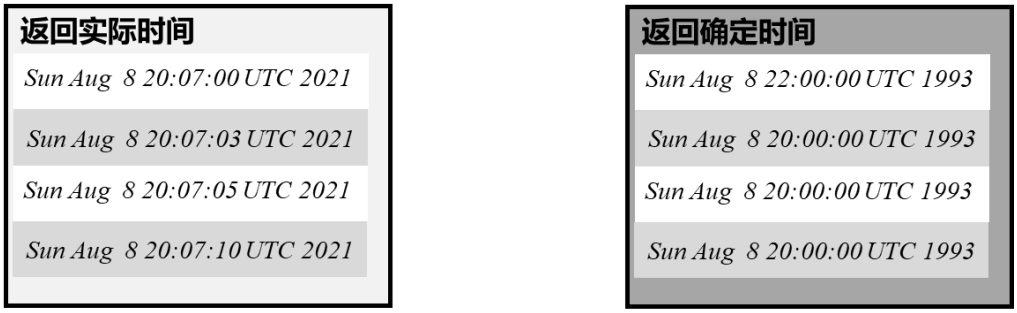
基准程序	问题规模	应用领域	共享访存	同步操作
$blackscholes$	65536 个期权	金融分析	少	很少
$bodytrack$	4 帧，4000 个粒子	计算机视觉	多	中
$canneal$	40 万个元件	电子工程	少	少
$fluidanimate$	5 帧，30 万个粒子	动画制作	很少	多
$swaptions$	64 个期权，2 万次模拟	金融分析	多	少

5.3 实验设计与分析

实验验证了容器中程序可重现性执行过程，测试了各种配置下程序的运行时间、存储空间等指标，并比较与原生执行、之前的可重现性工作 RR[7]的性能差异。RR 工具是目前最先进的记录重放工具之一，但是记录重放需要大量存储空间保存日志信息。

5.3.1 功能验证

首先需要验证能否成功在容器中可重现地执行程序，使用一个简单地示例程序$date$展示可重现性的功能验证，如下图 5.4 所示。$date$ 命令会实时返回系统时间，并且随着时间的变化而变化（图 5.4（a））。在可重现容器中调用$date$命令时，由于拦截并修改了底层请求时钟信息的系统调用，$date$ 命令会始终返回固定的时间信息（图 5.4（b））。



(a) (b)

图 5.4 $date$命令可重现性执行

在这个简单的功能性验证示例中，通过控制可重现性容器内部系统调用设置虚拟时间，在每次执行中输出的容器内部时间信息完全一致。

此外，还需要设置实验验证可重现方法支持的范围。实验设置选择 100 个 Debian 软件包的构建和编译过程作为实验对象，分别使用容器化可重现方法和 RR 工具可重现地执行上述过程，记录成功和失败的软件包数量。最终，容器中可重现地实现了 86 个软件包的构建过程，其中失败原因分别为不支持套接字操作（7 个软件包）、发送进程内信号（4 个软件包）、超时（3 个软件包）；而 RR 实现了 35 个软件包的可重现执行，主要失败原因为不支持$ioctl$ 调用（46 个软件包）和超时（19 个软件包），增加的平均额外性能损耗在 8.5 倍左右。在下一节中将会介绍二者的运行时性能比较。

5.3.2 整体方案性能损耗

本文的容器化可重现方案重点研究在 RISC-V 架构下实现可重现性，与此同时针对性能开销进行了适度优化，本节对容器化可重现方法的性能降低情况进行分析。实验设置选择 100 个 Debian 软件包的构建和编译过程作为实验对象，记录容器化可重现方法的运行时间，并与不使用可重现方法的基准测试进行相比。在确保软件包已经实现可重现性的前提下，通过记录其 5 次运行的平均时间来衡量其执行性能，并与不使用容器化可重现方法的基准测试情况进行比较。图 4.8 展示了 86 个能够成功可确定性执行的软件包构建过程中性能损耗与系统调用数量关系的散点图，其中纵轴是相对于基准测试的性能损耗倍数，以标准化后的对数刻度呈现；横轴为每秒执行的系统调用数量。

为了排除不同软件包构建过程相互之间的性能干扰，每个软件包的构建过

程后均重启模拟器进行试验。此外保证挑选的软件包构建基准时间大于等于 5 秒，以此减少极端情况下异常值的出现。

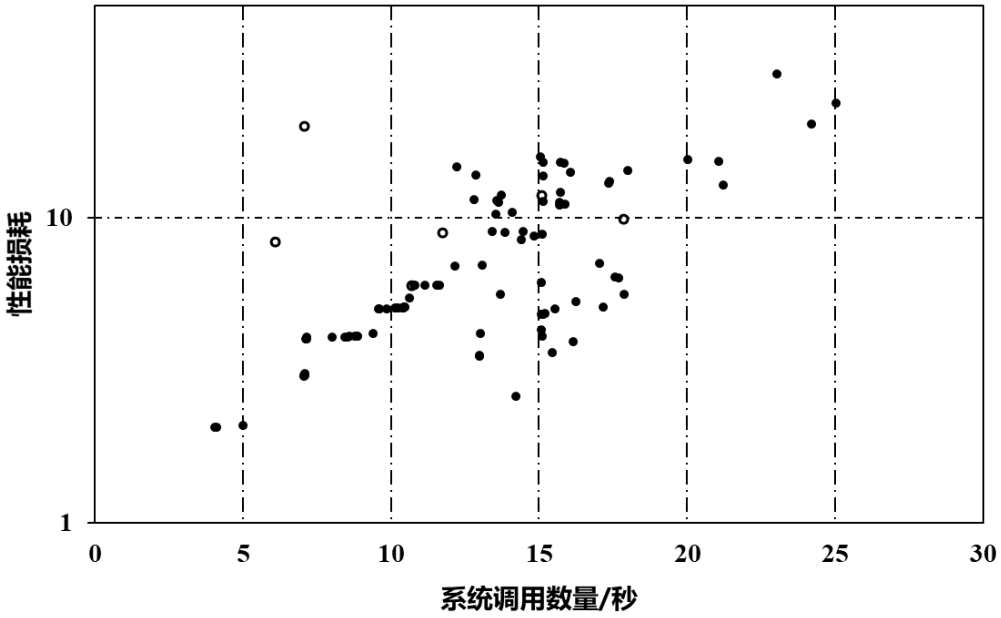


图 5.5 性能开销与每秒执行系统调用数量之间的关系

如图 5.5 所示，性能开销与系统调用数量整体呈现正相关关系。所有软件包构建过程的平均性能损耗在 7.9 倍左右，低于 RR 的 8.5 倍；每秒执行系统调用最多的软件包（每秒执行了 25000 次以上的系统调用）性能损耗也最高，降低了 31.9 倍左右的性能；与之相比，执行 5000 次系统调用/秒的软件包仅仅降低了约 2 倍左右的性能损耗。图 4.8 中的白色空心点表示使用多线程调度的软件包，黑色实心点表示不使用多线程调度的软件包。相比较而言，多线程软件包的执行速度更慢。

实验证明，频繁的系统调用会造成容器化可重现方法的性能损耗快速升高，同时进程调度也会增加额外的性能损耗。

4.3.3 运行时间

图 5.6 和图 5.7 展示了各种基准测试程序的运行时间对比，以不使用可重现性工具下程序的执行时间为基准，并与确定性重放工具 RR 对比，记录基准测试程序的平均开销。

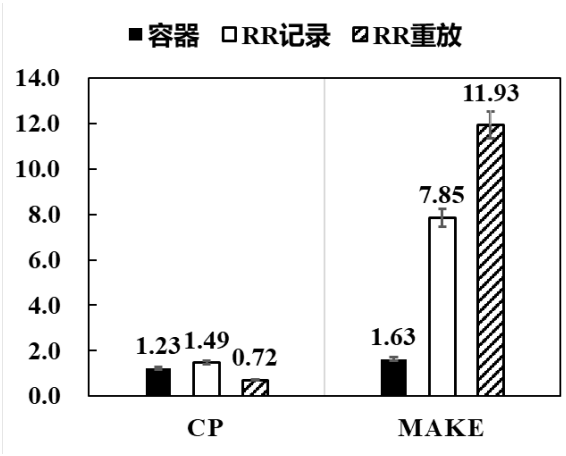


图 5.6 不包括 date 的系统命令执行时间比较

图 5.6 展示了cp 和$make$ 的运行时间，以不使用确定性重放工具作为其标准执行时间，展示使用不同方法的测试的执行时间与基准时间的比值。对于 RR 工具而言，分为记录和重放 2 个阶段，实验分别测试了这 2 个阶段的程序执行时间。每个测试运行 6 次，舍弃第 1 次结果，并计算剩余 5 次记录的算术平均值。图中的误差线表示了 95 % 的置信区间。由于cp 是单线程进行，而$make$ 使用了多线程（4 核）运行，$make$ 的开销明显高于cp。容器与 RR 之间比较，对于单线程运行的cp ，二者均实现了 1.5 倍以内的额外开销；但是对于多线程运行的$make$ ，RR 的额外开销变得十分巨大（最高 11.93 倍），而容器的额外开销依然保持在 2 倍以下（1.63 倍），容器方法具有更少的额外开销。

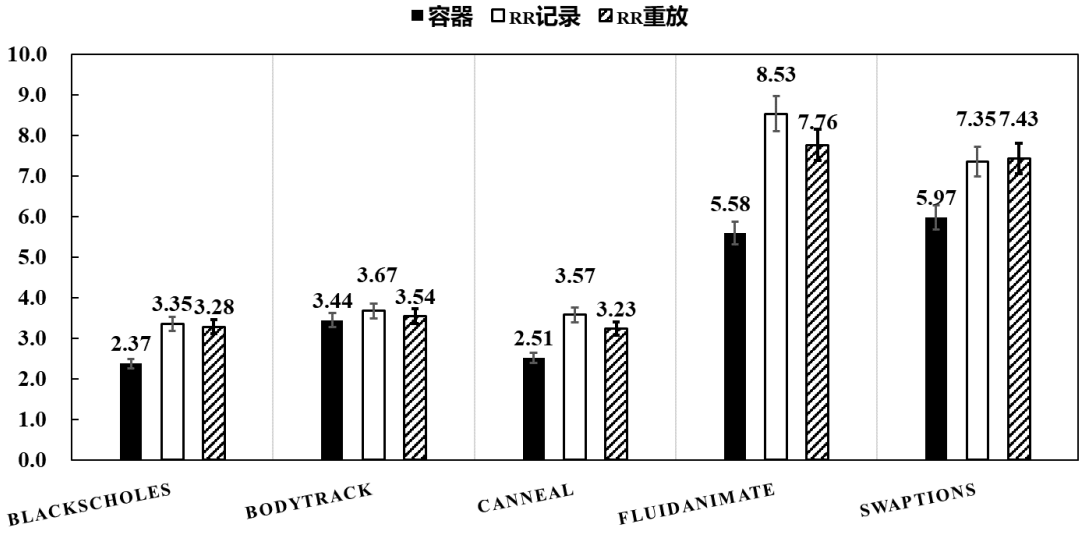


图 5.7 PARSEC 基准测试集执行时间比较

图 5.7 展示了 PARSEC 基准测试集中 5 个应用程序的平均执行时间开销，其中额外开销最多的程序是*swaptions*。由于*swaptions* 具有最多的共享访存操作数量，在可重现性方面的开销也会更多。相较于 RR，容器的额外开销总体低于 RR，大部分开销来源于容器的隔离和追踪进程的读取；而 RR 的记录重放过程由于需要大量的读写操作，不能保证提供高性能的可重现性（最大额外开销达到了 8 倍以上）。多次实验的平均结果表明，使用容器化可重现方法的运行时额外开销在 3.9 倍左右，而 RR 的记录阶段运行时开销在 5.3 倍左右，重放阶段运行时开销在 5.5 倍左右。

4.3.4 存储空间

实验记录并比较了容器和 RR 在存储空间方面的比较，跟踪包含三种方式消耗的存储空间：用于内存映射操作的复制（或硬链接)、复制文件块、以及所有其他跟踪进程的产生的文件（主要是事件元数据和系统调用的结果）。内存映射产生的文件主要来源于跟踪加载的可执行文件和库。

图 5.8 展示了每个工作负载的存储使用情况，单位为 MB。通过计算每个测试程序占用的内存空间的算术平均值得到如下结果。

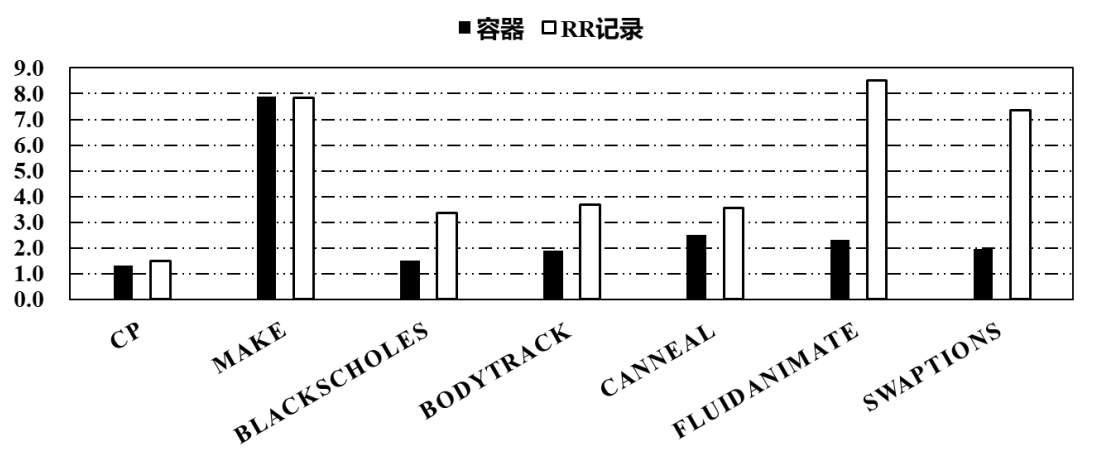


图 5.8 存储空间使用情况比较

实验表明，容器化可重现方法占用了更小的存储空间，实现了更加轻量级的可重现性。

4.3.5 容器性能损耗

容器化方法相较于裸机执行增加了中间层，优化了可重现性方案的可移植性，但是同时带来了确定性输出带来的开销以外的性能损耗。实验设计了性能对比，针对另外 2 种部署方式：（1）裸机执行；（2）虚拟机进行性能比较。

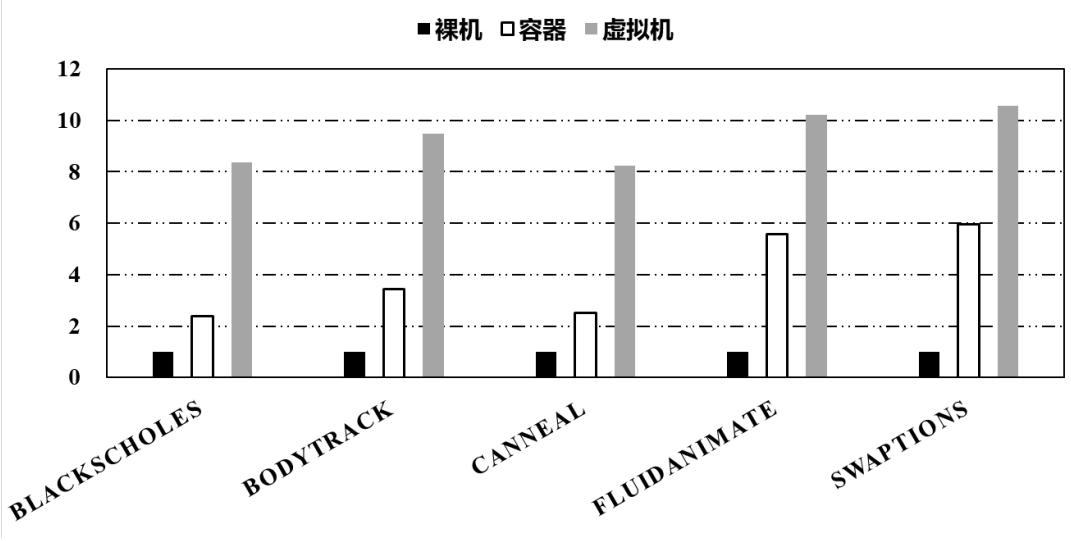


图 5.9 容器化方法性能损耗

图 5.9 展示了 PARSEC 基准测试集中的应用程序在 3 种部署方式下的执行时间，并以裸机执行的时间作为基准时间；虚拟机的方式执行时间较大，产生 10 倍以上的额外开销；容器化方法执行时间保持在 3.9 倍左右。相较于虚拟机方式，容器化方法更加轻量级，适合在资源受限的领域专用微处理器上使用

4.4 本章小结

本章在 Gem5 全系统模拟器上仿真 RISC-V 硬件平台，设计实验验证容器化可重现方案的可行性，并与现有的记录重放工具进行比较。实验结果表明，在软件包的可重现构建过程中，容器化可重现方案比传统记录重放工具 RR 支持了更多的软件，并且平均执行时间开销是基准测试时间的 7.9 倍，低于 RR 的 8.5 倍，随着系统调用次数的增加而增加；在 PARSEC 基准测试集的应用测试中，执行时间是基准测试的 3.9 倍，优于 RR 的 4.3 倍。因此，本文提出的可重现容器是 RISC-V 架构上更加轻量级的可重现性解决方案。

第 5 章 总结

本文探索了在模拟的 RISC-V 多核处理器设备中实现一个开源、高效的容器化可重现性方法。随着 RISC-V 指令集架构不断在各个领域应用场景中迸发生命力，为程序执行提供可重现性成为了亟需解决的挑战。之前的研究工作为 x86 指令集架构处理器提供了低开销实现用户进程确定性执行的方法，并通过容器技术实现了更通用的可重现性研究。本文针对 RISC-V 指令集架构这一新型硬件，实现了一种用户进程确定性执行的方法，并结合容器技术完成了在 RISC-V 上的部署，具体工作如下：

设计并实现了一种基于追踪进程的用户进程确定性抽象方法。该方法研究用户进程执行过程中的不确定性因素来源，结合确定性执行、记录重放等工作中的部分先进技术，设计并实现了一个用户进程确定性抽象。

设计并实现了基于容器的可移植性优化方法。为了在 RISC-V 架构下实现用户进程确定性抽象，该方法使用 Docker 对确定性抽象进行封装，实现与 LXC 嵌套的二层容器，消除底层微架构对可移植性的影响，同时隔绝了来自文件系统的不确定性影响；在容器内部，利用动态二进制翻译技术替换用户进程执行过程中具有不确定性的 CPU 指令。容器化可重现方案避免了对程序和操作系统内核的修改，减少了部署和维护的成本，同时更好的保证了用户进程的确定性。

本文在 Gem5 全系统模拟器上仿真 RISC-V 硬件平台，设计实验验证容器化可重现方案的可行性，并与现有的记录重放工具进行比较。实验结果表明，容器化可重现方案成功在 RISC-V 硬件上为程序执行提供了通用的可重现性，同时相较于记录重放工具 RR 增加了更少的额外性能损耗。

致　谢

参考文献

[1] Hennessy J L, Patterson D A. A new golden age for computer architecture[J]. Communications of the ACM, 2019, 62(2): 48-60.

[2] Celio C, Chiu PF, Asanović K, et al. Broom: an open-source out-of-order processor with resilient low-voltage operation in 28-nm CMOS [J]. IEEE Micro, 2019, 39(2):52-60.

[3] XiangShan-doc. UCAS & ICT, PCL. 2021. https://github.com/OpenXiangShan/XiangShan-doc

[4] LeBlanc T J, Mellor-Crummey J M. Debugging parallel programs with instant replay[J]. IEEE Transactions on Computers, 1987, 36(4): 471-482.

[5] Di Tucci L, Baghdadi R, Amarasinghe S, et al. SALSA: A domain specific architecture for sequence alignment[C]//2020 IEEE International Parallel and distributed processing symposium workshops (IPDPSW). IEEE, 2020: 147-150.

[6] Navarro Leija O S, Shiptoski K, Scott R G, et al. Reproducible containers[C]. Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2020: 167-182.

[7] O'Callahan R, Jones C, Froyd N, et al. Engineering record and replay for deployability[C]. 2017 USENIX Annual Technical Conference (USENIX ATC 17), 2017: 377-389.

[8] Narayanasamy S, Pokam G, Calder B. Bugnet: Continuously recording program execution for deterministic replay debugging[C]. 32nd International Symposium on Computer Architecture (ISCA'05), IEEE, 2005: 284-295.

[9] Srinivasan S M, Kandula S, Andrews C R, et al. Flashback: A lightweight extension for rollback and deterministic replay for software debugging[C]. USENIX Annual Technical Conference, General Track, 2004: 29-44.

[10] Saito Y. Jockey: a user-space library for record-replay debugging[C]. Proceedings of the sixth international symposium on Automated analysis-driven debugging, 2005, 69-76.

[11] Dunlap G W, King S T, Cinar S, et al. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay[C]. 5th Symposium on Operating Systems Design and Implementation (OSDI 02). 2002.

[12] Sheldon M, Weissman G V B. Retrace: Collecting execution trace with virtual machine deterministic replay[C]. Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2007). 2007.

[13] Asanović K, Patterson D A. Instruction sets should be free: The case for risc-v[J]. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, 2014.

[14] Waterman A S. Design of the RISC-V instruction set architecture[M]. University of California, Berkeley, 2016.

[15] Asanovic K, Avizienis R, Bachrach J, et al. The rocket chip generator[J]. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, 2016, 4.

[16] Herdt V, Große D, Pieper P, et al. RISC-V based virtual prototype: An extensible and configurable platform for the system-level[J]. Journal of Systems Architecture, 2020, 109: 101756.

[17] Farshchi F, Huang Q, Yun H. Integrating NVIDIA deep learning accelerator (NVDLA) with RISC-V SoC on FireSim[C]//2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2). IEEE, 2019: 21-25.

[18] Karandikar S, Biancolin D, Amid A, et al. Using FireSim to Enable Agile End-to-End RISC-V Computer Architecture Research[J]. 2019.

[19] Sá B, Martins J, Pinto S E S. A First Look at RISC-V Virtualization from an Embedded Systems Perspective[J]. IEEE Transactions on Computers, 2021.

[20] Asanovic K, Avizienis R, Bachrach J, et al. The rocket chip generator[J]. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, 2016, 4.

[21] Asanovic K, Patterson D A, Celio C. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor[R]. University of California at Berkeley Berkeley United States, 2015.

[22] Chen C, Xiang X, Liu C, 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]//2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020: 52-64.

[23] XiangShan-doc. UCAS & ICT, PCL. 2021. https://github.com/OpenXiangShan/XiangShan-doc

[24] Feng E, Lu X, Du D, et al. Scalable Memory Protection in the PENGLAI Enclave[C]//15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21). 2021: 275-294.

[25] Bergan T, Hunt N, Ceze L, et al. Deterministic Process Groups in dOS[C]. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10), 2010.

[26] Devecsery D, Chow M, Dou X, et al. Eidetic systems[C]. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 2014: 525-540.

[27] Pan D Z, Linton M A. Supporting reverse execution for parallel programs[C]//Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging. 1988: 124-129.

[28] Bacon, D.F., Goldstein, S.C.: Hardware-assisted Replay of Multiprocessor Programs. In: Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging. pp. 194–206. PADD '91, ACM, New York, NY, USA (1991), DOI: 10.1145/122759.122777

[29] Patil H, Pereira C, Stallcup M, et al. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs[C]//Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization. 2010: 2-11.

[30] Bhansali S, Chen W K, De Jong S, et al. Framework for instruction-level tracing and analysis of program executions[C]//Proceedings of the 2nd international conference on Virtual execution environments. 2006: 154-163.

[31] Xu M, Bodik R, Hill M D. A" flight data recorder" for enabling full-system multiprocessor deterministic replay[C]//Proceedings of the 30th annual international symposium on Computer architecture. 2003: 122-135.

[32] Hower D R, Hill M D. Rerun: Exploiting episodes for lightweight memory race recording[J]. ACM SIGARCH computer architecture news, 2008, 36(3): 265-276.

[33] Montesinos P, Ceze L, Torrellas J. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef? ciently[J]. ACM SIGARCH Computer Architecture News, 2008, 36(3): 289-300.

[34] Pokam G, Danne K, Pereira C, et al. QuickRec: Prototyping an Intel architecture extension for record and replay of multithreaded programs[C]//Proceedings of the 40th annual international symposium on computer architecture. 2013: 643-654.

[35] Devietti J, Lucia B, Ceze L, et al. DMP: Deterministic shared memory multiprocessing[C]//Proceedings of the 14th international conference on Architectural support for programming languages and operating systems. 2009: 85-96.

[36] Bergan T, Anderson O, Devietti J, et al. CoreDet: A compiler and runtime system for deterministic multithreaded execution[C]//Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems. 2010: 53-64.

[37] Hower D R, Dudnik P, Hill M D, et al. Calvin: Deterministic or not? free will to choose[C]//2011 IEEE 17th International Symposium on High Performance Computer Architecture. IEEE, 2011: 333-334.

[38] Devietti J, Nelson J, Bergan T, et al. RCDC: a relaxed consistency deterministic computer[J]. ACM SIGARCH Computer Architecture News, 2011, 39(1): 67-78.

[39] Liu T, Curtsinger C, Berger E D. Dthreads: efficient deterministic multithreading[C]//Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. 2011: 327-336.

[40] Merrifield T, Eriksson J. Conversion: Multi-version concurrency control for main memory segments[C]//Proceedings of the 8th ACM European Conference on Computer Systems. 2013: 127-139.

[41] Lu K, Zhou X, Bergan T, et al. Efficient deterministic multithreading without global barriers[J]. ACM SIGPLAN Notices, 2014, 49(8): 287-300.

[42] Merrifield T, Devietti J, Eriksson J. High-performance determinism with total store order consistency[C]//Proceedings of the Tenth European Conference on Computer Systems. 2015: 1-13.

[43] Merrifield T, Roghanchi S, Devietti J, et al. Lazy determinism for faster deterministic multithreading[C]//Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. 2019: 879-891.

[44] Ronsse M, De Bosschere K. RecPlay: A fully integrated practical record/replay system[J]. ACM Transactions on Computer Systems (TOCS), 1999, 17(2): 133-152.

[45] Devecsery D, Chow M, Dou X, et al. Eidetic systems[C]//11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 2014: 525-540.

[46] Lee D, Wester B, Veeraraghavan K, et al. Respec: efficient online multiprocessor replay via speculation and external determinism[J]. ACM Sigplan Notices, 2010, 45(3): 77-90.

[47] Mashtizadeh A J, Garfinkel T, Terei D, et al. Towards practical default-on multi-core record/replay[J]. ACM SIGPLAN Notices, 2017, 52(4): 693-708.

[48] Aviram A, Hu S, Ford B, et al. Determinating timing channels in compute clouds[C]//Proceedings of the 2010 ACM workshop on Cloud computing security workshop. 2010: 103-108.

[49] Hunt N, Bergan T, Ceze L, et al. DDOS: taming nondeterminism in distributed systems[J]. ACM SIGPLAN Notices, 2013, 48(4): 499-508.

[50] Bergan T, Hunt N, Ceze L, et al. Deterministic Process Groups in dOS[C]. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10), 2010.

[51] 徐子晨, 崔傲, 王玉皞, 等. 基于 RISC-V 架构的强化学习容器化方法研究[J]. 计算机工程与科学, 2021.

[52] Popek G J, Goldberg R P. Formal requirements for virtualizable third generation architectures[J]. Communications of the ACM, 1974, 17(7): 412-421.

[53] Bellard F. QEMU, a fast and portable dynamic translator[C]//USENIX annual technical conference, FREENIX Track. 2005, 41(46): 10.5555.

[54] Bernstein D. Containers and Cloud: From LXC to docker to Kubernetes[J]. IEEE Cloud Computing, 2014, 1(3): 81-84.

[55] Dunlap G W, Lucchetti D G, Fetterman M A, et al. Execution replay of multiprocessor virtual machines[C]//Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. 2008: 121-130.

[56] Sheldon M, Weissman G V B. Retrace: Collecting execution trace with virtual machine deterministic replay[C]//Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2007). 2007.

[57] Dolan-Gavitt B, Hodosh J, Hulin P, et al. Repeatable reverse engineering with PANDA[C]//Proceedings of the 5th Program Protection and Reverse Engineering Workshop. 2015: 1-11.

[58] Dovgalyuk P. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging[C]//CSMR. 2012: 553-556.

[59] Srinivasan D, Jiang X. Time-traveling forensic analysis of vm-based high-interaction honeypots[C]//International Conference on Security and Privacy in Communication Systems. Springer, Berlin, Heidelberg, 2011: 209-226.

[60] Burtsev A, Johnson D, Hibler M, et al. Abstractions for practical virtual machine replay[J]. ACM SIGPLAN Notices, 2016, 51(7): 93-106.

[61] Bhansali S, Chen W K, De Jong S, et al. Framework for instruction-level tracing and analysis of program executions[C]//Proceedings of the 2nd international conference on Virtual execution environments. 2006: 154-163.

[62] Olszewski M, Ansel J, Amarasinghe S. Kendo: efficient deterministic multithreading in software[C]//Proceedings of the 14th international conference on Architectural support for programming languages and operating systems. 2009: 97-108.

[63] Roelke A, Stan M R. Risc5: Implementing the RISC-V ISA in gem5[C]//First Workshop on Computer Architecture Research with RISC-V (CARRV). 2017, 7(17).

[64] Ta T, Cheng L, Batten C. Simulating multi-core RISC-V systems in gem5[C]//Workshop on Computer Architecture Research with RISC-V. 2018.

[65] Herdt V, Große D, Pieper P, et al. RISC-V based virtual prototype: An extensible and configurable platform for the system-level[J]. Journal of Systems Architecture, 2020, 109: 101756.

[66] Hin P Y H, Liao X, Cui J, et al. Supporting RISC-V Full System Simulation in gem5[C]//Workshop on Computer Architecture Research with RISC-V. 2021.

[67] Bienia C, Kumar S, Singh J P, et al. The PARSEC benchmark suite: Characterization and architectural implications[C]//Proceedings of the 17th international conference on Parallel architectures and compilation techniques. 2008: 72-81.

[68] Bruening D, Garnett T. Building dynamic instrumentation tools with DynamoRIO[C]//Proc. Int. Conf. IEEE/ACM Code Generation and Optimization (CGO), Shen Zhen, China. 2013.

[69] Patterson D, Waterman A. The RISC-V Reader: an open architecture Atlas[M]. Strawberry Canyon, 2017.

[70] Abadi, Daniel J. and Faleiro, Jose M. An Overview of De- terministic Database Systems. Communications of the ACM, 61(9):78–88, August 2018

[71] Lu L, Scott M L. Toward a formal semantic framework for deterministic parallel programming[C]//International Symposium on Distributed Computing. Springer, Berlin, Heidelberg, 2011: 460-474.

[72] Zandberg K, Baccelli E. Femto-Containers: DevOps on Microcontrollers with Lightweight Virtualization & Isolation for IoT Software Modules[J]. arXiv preprint arXiv:2106.12553, 2021.

[73] Lowe-Power J, Nitta C. The Davis In-Order (DINO) CPU: A Teaching-Focused RISC-V CPU Design[C]//Proceedings of the Workshop on Computer Architecture Education. 2019: 1-8.

[74] Micco J. The state of continuous integration testing@ google[J]. 2017.

[75] Lee D, Wester B, Veeraraghavan K, et al. Respec: efficient online multiprocessor replayvia speculation and external determinism[J]. ACM Sigplan Notices, 2010, 45(3): 77-90.

[76] Lee D, Chen P M, Flinn J, et al. Chimera: Hybrid program analysis for determinism[C]//Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation. 2012: 463-474.

攻读学位期间的研究成果

已发表论文：

徐子晨，**崔傲**，王玉皞，刘韬. 基于 RISC-V 架构的强化学习容器化方法研究. 计算机工程与科学[J]. 2021，43(2): 70-74.

Xu Z, Bai G, **Cui A**, et al. Power-aware throughput control for containerized relational operation[J]. CCF Transactions on High Performance Computing, 2021, 3(1): 70-84.