摘 要

在进行程序调试、分布式系统构建及机器学习模型训练等应用场景中，往往需要确保程序在不同机器间反复执行能够得到可重现的结果。现有的确定性重放工作难以同时满足确定性和可移植性的要求：一是保证输出的确定性需要对程序进行的额外修改来维护系统时间或标识符的唯一性；二是可移植性差，难以跨机器执行。针对上述问题，本文对基于RISC-V架构的容器化可重现方法展开研究，主要工作如下：

第一，本文对Linux系统下的不确定性进行研究，确定不确定性来源，并就确定性重放问题的国内外研究现状展开调查与分析。

第二，基于上述问题与内容，本文在x86架构下实现一种容器化可重现方法，建立可重现容器抽象，保证容器内程序强制以可重现的方式运行，且无需对源程序进行修改，并向RISC-V平台进行移植优化。该方法通过命名空间等隔离用户进程，利用ptrace对标识符、系统调用、信号等中的非确定因素进行追踪、拦截，实现可重现的软件输出。本文还通过模拟器将可重现容器抽象移植到RISC-V架构下运行。

第三，本文使用Gem5对RISC-V硬件平台进行全系统仿真，并进行容器化可重现方法的实验验证和性能评估。评估结果表明，与xxxx相比，…………。

**关键词：**RISC-V,记录重放,容器,虚拟化

ABSTRACT

……

**Key Words:**

目 录

[第1章 引言 1](#_Toc99121818)

[1.1 研究背景与意义 1](#_Toc99121819)

[1.2 国内外研究现状 2](#_Toc99121820)

[1.2.1 基于程序的可重现性研究——确定性执行 2](#_Toc99121821)

[1.2.2 基于环境的可重现性研究——确定性操作系统 3](#_Toc99121822)

[1.2.3 基于虚拟化的可重现性研究 4](#_Toc99121823)

[1.3 主要研究内容 4](#_Toc99121824)

[1.4 组织结构 5](#_Toc99121825)

[第2章 RISC-V指令集与可重现性研究 7](#_Toc99121826)

[2.1 RISC-V指令架构 7](#_Toc99121827)

[2.1.1 RISC-V基础指令集 8](#_Toc99121828)

[2.1.2 RISC-V扩展指令集 10](#_Toc99121829)

[2.1.3 RV32/64特权架构 11](#_Toc99121830)

[2.2 可重现性技术 13](#_Toc99121831)

[2.2.1 不确定因素来源 13](#_Toc99121832)

[第3章 容器化可重现方法设计与实现 15](#_Toc99121833)

[3.1 可重现方法工作流程 15](#_Toc99121834)

[3.1.1 避免数据竞争 16](#_Toc99121835)

[3.2.2 拦截系统调用 16](#_Toc99121836)

[3.2.3 异步事件 17](#_Toc99121837)

[3.2.4 共享内存 18](#_Toc99121838)

[3.2 进程内系统调用拦截 20](#_Toc99121839)

[3.2.1 拦截系统调用 21](#_Toc99121840)

[3.2.2 选择性拦截 21](#_Toc99121841)

[3.2.3 检查系统调用阻塞 22](#_Toc99121842)

[3.3 基于用户空间的容器设计 23](#_Toc99121843)

[3.3.1 进程标识符 23](#_Toc99121844)

[3.3.2 文件和目录 24](#_Toc99121845)

[第4章 基于RISC-V架构的可重现容器化设计与实现 26](#_Toc99121846)

[4.1 RISC-V容器化方法 26](#_Toc99121847)

[4.1.1 Docker容器 26](#_Toc99121848)

[4.1.2 基于QEMU的模拟器 26](#_Toc99121849)

[4.1.3 动态二进制指令翻译 26](#_Toc99121850)

[4.2 基于Gem5的RISC-V全系统仿真 26](#_Toc99121851)

[4.2.1 Gem5全系统模拟器 27](#_Toc99121852)

[4.2.2 RISC-V目标系统构建 28](#_Toc99121853)

[4.2.3 HiFive平台 28](#_Toc99121854)

[4.2.4 启动Linux系统 29](#_Toc99121855)

[第5章 实验设计与分析 31](#_Toc99121856)

[5.1 实验平台 31](#_Toc99121857)

[5.2实验环境配置 31](#_Toc99121858)

[5.3 可重现性方法功能验证 31](#_Toc99121859)

[5.4 可重现方法性能损耗分析 31](#_Toc99121860)

[第6章 总结与展望 32](#_Toc99121861)

[致 谢 33](#_Toc99121862)

[参考文献 34](#_Toc99121863)

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寻求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提出了一个免费、开源的指令集RISC-V（RISC Five），并凭借其灵活的模块化设计广泛应用于各种领域独特的高性能微处理器设计中[2][3]。

随着RISC-V软硬件生态的日益完善，逐渐对RISC-V架构下软件的可重现性提出了要求。一方面，RISC-V并行程序调试过程中需要循环执行程序并重现错误，以此提高程序的可靠性[4]；另一方面，RISC-V平台上训练和推理机器学习模型的需求日益增加，结果的可重现性能够追踪模型性能变化趋势，帮助开发人员寻找性能变化的原因[5]。因此，以较低的额外开销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，反复执行程序始终得到相同的结果；可移植性则保证程序无需过多修改即可在不同机器上部署、执行。在软件工程中，通常使用确定性重放（Deterministic Replay，或记录重放，Record and Replay, R&R）技术[7][8][9]实现程序的重现性。确定性重放技术分为记录和重放两个阶段，通过记录并重放程序执行过程，尽可能保证程序确定性的输出。确定性重放工具通过追踪、记录程序的执行踪迹（Execution Trace），在下一次运行该程序时按照记录的踪迹信息重现执行结果。按照重放系统范围，可将现有确定性重放工具分为程序重放和全系统重放。程序确定性重放技术的核心问题是查找并拦截程序执行过程中不确定性（Non- deterministic）的来源，如查找、拦截具有不确定性的系统调用和CPU指令，记录并重现它们返回的数据，设置周期性的进程检查点以实现在程序执行的任意时刻进行跳转[10]。除此之外，还存在确定性操作系统的解决方案，通过记录整个虚拟机[11][12]，或者修改系统内核[13][14]等方法来保证可重现性。

然而，目前的确定性重放工具并不能完全建立可重现抽象。程序确定性重放工具需要在源程序基础上进行修改，增加部署和维护成本，产生额外的性能和复杂性开销，并且通用性差，无法记录所有程序[7]；全系统重放记录整个虚拟机的方式更加复杂，修改内核同样增加部署和维护成本，且要求在特定的指令集架构和操作系统环境下实现[11][12]。因此，现有的确定性重放工具并不能完全满足可重现性的要求。

基于上述背景，本文针对RISC-V上的程序执行的可重现性问题，设计实现一种容器化可重现方法：在程序执行期间，通过命名空间隔离用户进程和主机系统，通过进程追踪拦截并修改不确定性的系统调用，基于QEMU的二进制翻译工具修改不确定性的CPU指令，并在RISC-V平台上完成实验验证与分析。

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，并不断应用到其它领域中，国内外研究人员已经对可重现性进行了深入的研究。目前的大部分可重现性研究基于之前的确定性重放工具实现。根据对程序记录和重放的范围，可以将确定性重放工具划分为以下三个方向，分别是基于程序的可重现性研究、基于执行环境的可重现性研究以及基于虚拟化的可重现性研究，本节从这三个方向对可重现性研究的国内外现状进行综述。

1.2.1 基于程序的可重现性研究——确定性执行

在19世纪80年代后期，研究人员逐渐认识到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[15]，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相关工作的研究方向与思路，即分为记录和重放两阶段，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。但Bugnet只支持特定API；

[50-58]

Jockey[5]可以实现与Flashback相似的功能，可作为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。

以低开销记录在多个内核上并发运行的应用 程序线程，具有数据竞争的可能性，极具挑战 性。PinPlay [34] 和 iDNA/Nirvana [7] 检测共享内存负载并报告高开销。SMP-ReV irt [18] 使用硬件页面保护跟踪页面所有权 ，并在具有大量共享的基准上报告高开销。D oublePlay [39] 运行应用程序的两个实例 ，因此当单独的应用程序可能使可用内核饱和 时具有很高的开销。ODR [5] 具有低记录开销 ，但重放可能非常昂贵，并且不能保证重现相 同的程序状态。Castor [29] 通过修改编译 器和运行时系统来检测同步代码，这为易于部 署创造了障碍，并且在存在数据竞争的情况下 无法可靠地重放。 一般的、低开销并行记录的最大希望似乎是 硬件支持。FDR [41]、BugNet [31]、Reru n [24]、DeLorean [30] 和 QuickRec [35] 等项目已经探索了低开销并行记录硬 件

1.2.2 基于环境的可重现性研究——确定性操作系统

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。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%。之后的大部分工作中都采用了逻辑时钟的方式增强可重现性。

[40-46]学术界

[47-49]工业界

此外，Aviram等人[7]、Hunt等人[8]、Bergan等人[9]均提出了操作系统级别的可重现抽象。

Flashback[4]能够记录和重放更多的系统调用和CPU指令，但Flashback必须作为操作系统拓展使用，修改操作系统内核，部署和维护困难，且只支持单线程程序的确定性重放；

Aviram 等人2010年提出了Determinator，这是一个围绕确定性设计的操作系统抽象。Determinator实现了操作系统级别的确定性，强制单个进程，甚至交互进程组确定性执行。Determinator通过拒绝用户代码直接访问硬件资源来强制执行确定性，包括实时时钟、周期计数器和可写共享内存等。

DDOS专注于

Scribe [26]、dOS [6] 和 Arnold [14] 通过 扩展具有记录和重放功能的 OS 内核来重放一 个进程或一组进程。内核更改使维护和部署更加 困难——除非将记录和重放集成到基本操作系统 中。但是向内核添加侵入性的新特性是有风险的 ，因此如果可以在内核之外很好地实现记录和重 放，那么将其移入内核可能是不可取的。

MEC [11] 以及后来的 Jockey [36 ] 和 liblog [21] 开始，就已经存在纯用户 空间记录和回放系统。这些系统不处理异步事 件计时和其他操作系统功能。PinPlay [34] 、iDNA [7]、UndoDB [1] 和 TotalView Re playEngine [22] 使用代码检测来记录和重 放异步事件时序。与 UndoDB 和 RR 不同，P inPlay 和 iDNA 在 strument 中全部加载

1.2.3 基于虚拟化的可重现性研究

容器技术起源于虚拟化。虚拟化技术是在一台主机上运行多个进程，将硬件资源抽象为虚拟逻辑对象的技术，包括计算机的硬件资源、存储设备和网络资源的虚拟等。虚拟化技术包括平台虚拟化、硬件虚拟化、应用程序虚拟化等，平台虚拟化技术允许在宿主机设备中运行多个异构的体系结构应用，通过虚拟机监视器（Virtual Machine Monitor，VMM，或称为Hypervisor）为用户提供抽象、虚拟的硬件环境。Popek和Goldberg等人1974年的论文[8] 为将系统软件视为VMM确立了三个基本特征：（1）保真。 VMM上的软件的执行与硬件上的执行相同，除非定时影响；（2）性能。 绝大多数来宾指令由硬件执行，而无需VMM的干预；（3）安全。 VMM管理所有硬件资源。VMM通过内核代码的二进制翻译实现虚拟化，在宿主机和虚拟机之间添加一层中间层，将宿主机处理器的指令代码转换、翻译成目标处理器的指令集，捕获文件执行时所需的系统调用。VMware ® Workstation、Virtual PC、 QEMU等均是采用的这种方法实现硬件的虚拟化。Adams等人[9]对基于x86架构下的软硬件虚拟化技术进行了比较，得出结论，硬件VMM的性能通常比纯软件VMM低。硬件虚拟化技术不具备性能优势的原因主要有2个：（1）它不支持MMU虚拟化。 （2）它无法与用于MMU虚拟化的现有软件技术共存。Shuja等人[10]根据针对ARM架构下移动虚拟化的硬件支持的最新进展，调查了基于软件和硬件的移动虚拟化技术，并介绍了CPU，内存，I / O，中断和网络接口的在移动设备中虚拟化面临的挑战和问题。他们的研究最后提出，在资源受限的移动设备上实施基于CPU的虚拟化解决会消耗CPU周期和内存空间，实现该方案的成本总是很高，而使用静态二进制转换实现虚拟化的解决方案开销更低。针对资源有限的边缘设备必须使用资源有效的技术来解决上述问题。Bernstein等人[11] 介绍了Docker和Kubernetes，前者是一个开源项目，可以自动化Linux应用程序的快速部署，后者是一个用于Docker容器的开源集群管理器。

ReVirt [17]

VMware [28]

全系 统模拟器 Simics 通过确定性重新执行 [2 0] 支持反向执行调试。已经努力为 QEMU [ 15, 16, 38] 和 Xen [18, 10] 添加一些记 录和回放支持。

1.3 主要研究内容

本文针对RISC-V架构上程序执行的可重现性问题进行研究，总体技术路线如图1.1所示：

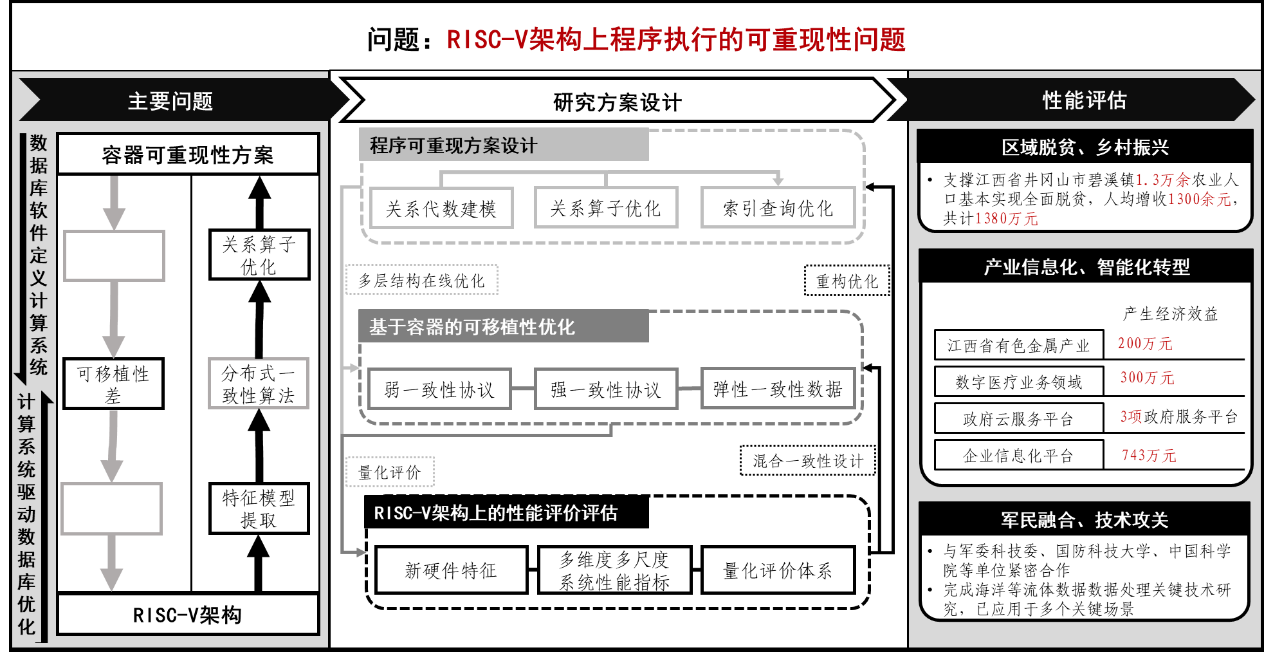


图1.1 基于RISC-V的可重现容器化方法研究路线示意图

本文在可重现性技术、容器化技术的基础上，提出一种程序执行的容器化可重现方法，在RISC-V架构上设计完整实验方案，测试特定应用在RISC-V架构上容器化可重现方法的额外性能开销，验证其可行性和有效性。具体工作如下：

1．容器化可重现方法设计与实现

本文在现有不确定性拦截工作的基础上，针对程序执行过程中由于访存冲突、外部信号、系统调用等导致的执行结果可重现性问题进行分析，通过进程对不确定性来源进行拦截与修改，结合容器化技术，设计并实现了一种程序执行容器化可重现方法。 该方法在隔离的容器中确定性执行程序，不需要对程序进行额外的修改。

2．基于RISC-V架构的可移植性优化设计与实现

本文对RISC-V指令、特权架构等不确定性来源进行分析，改进程序执行容器化可重现方法，将容器化可重现方法移植到RISC-V架构硬件平台中。基于全系统模拟器Gem5构建RISC-V硬件平台，设计完整实验方案验证RISC-V架构下的容器化可重现方法的可行性和有效性，并对方案产生的额外性能损耗进行分析。

1.4 组织结构

本文包括六个章节，主要用四个章节阐述基于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，从基于程序的可重现性研究问题、基于环境的可重现性研究问题以及基于虚拟化的可重现性研究问题三个角度，介绍国内外相关工作以及本文主要工作，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方法的相关理论知识，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化的相关理论知识。

第三章：提出了基于一种容器化可重现方法的设计与实现。首先介绍基于系统调用函数拦截和修改系统调用的可重现性方法设计，然后阐述如何在用户空间运行该方案，通过容器化技术隔离程序执行环境，以及拦截的不确定性因素的主要来源。

第四章：在上述容器化可重现方法基础上，设计并实现基于RISC-V架构的可移植性优化。首先介绍了基于全系统模拟器Gem5构建模拟RISC-V硬件平台的方法，其次在此基础上设计容器化方法优化程序可移植性，并分析RISC-V指令及其特权架构的不确定性来源，最终完成基于RISC-V架构的容器化可重现方法设计与实现。

第五章：实验设计与分析。本章在RIS-V平台上设计实验方案，通过并行程序基准测试集，验证方案设计的正确性，对方案产生的性能损耗进行分析。

第六章：总结与展望。对全文工作进行总结。

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，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化的相关理论知识。

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”，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。

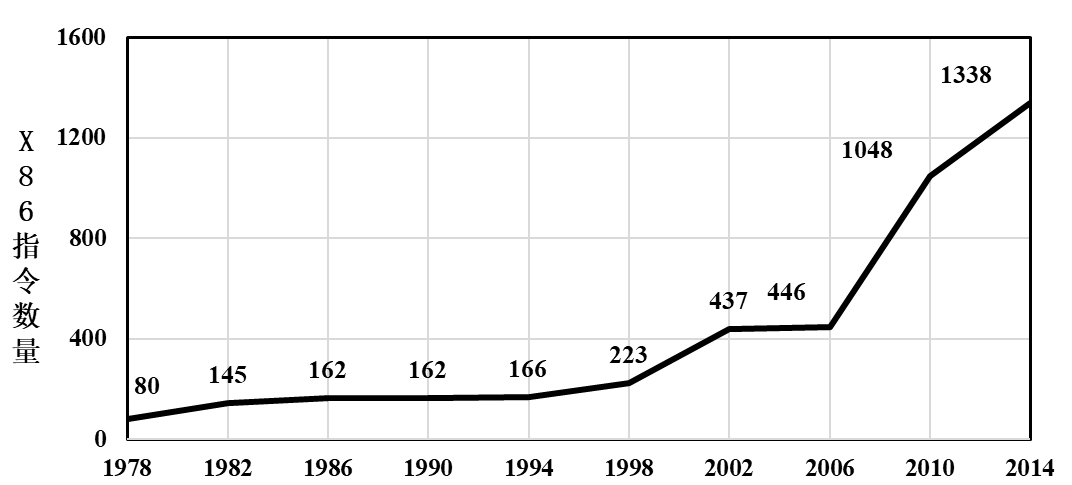
**（2）模块化设计**

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。

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**

图2.1 x86指令集架构中指令数量增长



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：

该指令执行5次数据加载并写入6个寄存器，但仅当条件码置位时才执行。此外，它将结果写入寄存器，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。

由于RISC-V的开源、免费、模块化架构，RISC-V被广泛应用于各种领域独特的高性能微处理器设计中。国外关于RISC-V的处理器如SweRV[13]、FABulous[15]等，国内也推出了“香山”[16]、“蓬莱”[17]等开源处理器核。

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，能够定制处理器指令集架构设计与实现提供必要功能。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2.4（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。

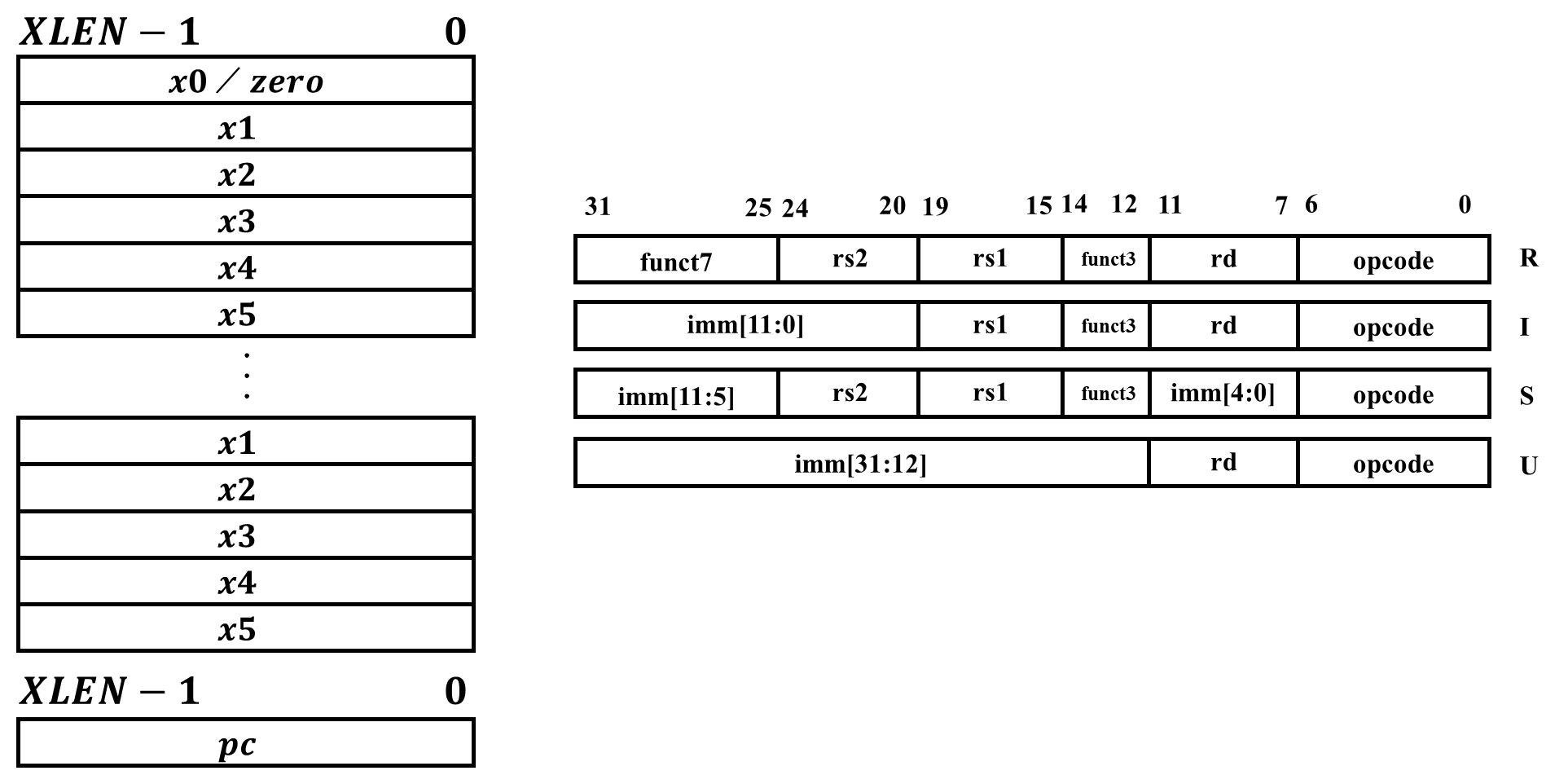


图2.4 （a）RV32I寄存器状态 （b）RV32I的4种基础指令格式

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。同时新增指令，用于操作低32位；新增了 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，保留了指令，新增用于操作寄存器低64位的指令。

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和存储操作上，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2.6所示：

表2.6 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**

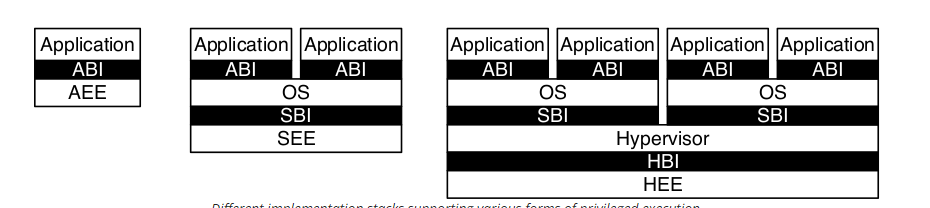
基于页面的内存类型（Page-based Memory Types）。

**（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。



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而开发的确定性重放技术。不确定性因素的来源包括：（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。确定性重放技术一般通过在应用程序执行过程中记录不确定性因素，并在按照记录的日志重放程序的执行过程，在循环执行中得到确定的数据流结果。随着确定性技术在除了调试以外的其他领域应用，如分布式一致性，确定性重放技术逐渐向更高要求的可重现性技术发展。

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[3]，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；（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。

确定性重放也常被称为记录重放（Record and Replay），首先介绍常见的不确定性因素来源，在后续章节中会设计并实现相应的可重现方案。

2.2.1 不确定因素来源

确定性代表着数据流确定性，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、获取计时器值等）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，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 Linux容器技术

今生是docker大流行的时代，而前世就是早于1982年的chroot工具，以及后面经过改进并且现在还在使用的lxc技术。早期的docker的代码实现基于LXC（0.9之前）。

Linux容器功能是基于 cgroups 和 Namespace 来实现的. 所以要了解 Linux 容器必须先了解 cgroup 和 Namespace.

2.1、cgroups（控制组）

cgroups 是将进程分组管理的内核功能.通过cgroups可以隔离进程, 同时还可以控制进程的资源占用(CPU, 内存等等)情况在操作系统底层限制物理资源，起到 Container 的作用。进程可用的cpu资源由cpuset指定。

2.2、Namespace (命名空间)

Namespace让每个进程组有独立的PID, IPC和网络空间.Namespace通过 clone系统调用来实现的.clone系统调用的第3个参数flags就是通过设置Namespace来划分资源的.

容器

2.2.3 跟踪进程系统调用ptrace

目前大多数基于软件的可重现方法是通过记录程序运行时的系统调用来实现的，本文的基本思路同样如此。使用系统调用函数 监视并拦截用户系统容器中程序执行的所有系统调用。 是Linux内核提供的用于进程追踪的系统调用，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 可以用来实现断点调试和系统调用跟踪等目的。

的函数定义如下：

被追踪的子进程中的信息转换为信号，传递给执行追踪任务的父进程。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。

被集成在GDB、Strace、Ltrace等系统调试工具中，但是 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。

首先使用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）。

第3章 容器化可重现方法设计与实现

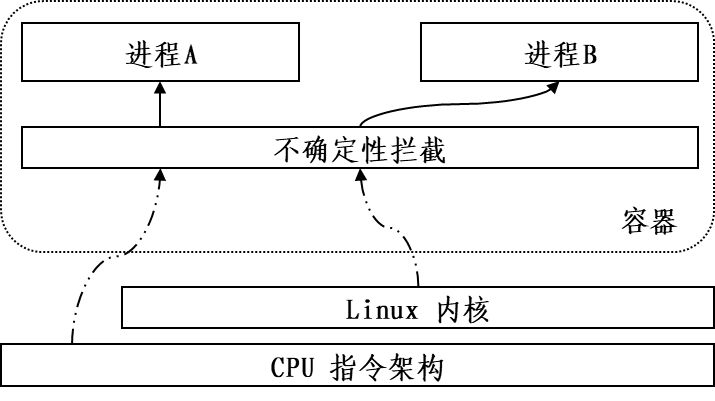
本章提出了基于一种容器化可重现方法的设计与实现。首先介绍了基于系统调用函数拦截和修改系统调用的过程方法，然后在此基础上详细阐述了如何在用户空间运行该方案，通过容器化技术隔离程序执行环境，以及拦截的不确定性因素的主要来源。本方案的结构框图如图3.1所示：

图3.1 容器化可重现方法设计框图

3.1 容器化可重现方法设计

可重现方法大多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。在第二章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。基于此提出一种容器化可重现方法，利用容器对执行程序进行封装，划分确定性程序状态和计算的边界。在记录阶段，监视并拦截容器内的所有非确定性来源，记录所有进入容器边界的输入，并通过重放非确定性和输入来重现容器内的执行结果。在重放阶段，容器内所有的状态和计算结果将与记录阶段相匹配。

具体实现中，使用2层嵌套的容器机制隔离用户程序，并基于进程跟踪系统调用 监视并拦截进程中的不确定性因素。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、用户命名空间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。

其次，使用 拦截容器中执行的所有系统调用。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。

最后，通过 命令隔离用户进程与主机文件系统的交互，并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。

通过这种设计，容器化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 数据竞争

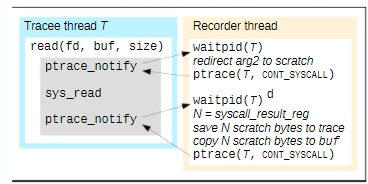
由于线程通常在多个内核上运行，不同线程对同一内存位置的数据访存竞争将会产生不确定性因素。 因此，采用通用方法 [17, 28, 1, 15]调度线程，同一时刻只运行一个线程。线程调度过程中，上下文切换时间是必须记录的不确定性因素来源。如果上下文切换发生在程序执行中不正确的位置，会出现由于数据竞争产生的错误。

这种方法比替代方案[7, 18, 34, 39, 29 ]，以及避免假设程序是无竞争的方案[14, 29]更有效，并且对于低并行度工作负载同样适用。对于具有持续高度并行性的工作负载，调度速度会大幅下降；但是高度并行的应用程序在测试较小数据集时的并行性有限，同样适用本方案。

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。 系统调用允许父进程监督其他子进程和线程的执行，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 系统调用或从特殊的 或 文件中读取随机数据，会为程序带来不一致的结果。容器需要使用拦截 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，和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。

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 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3.2展示了记录一个简单的读取系统调用进程，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。

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 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 [17, 33, 10] 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）。确定性[40]）在实践中并不适用于大多数CPU性能计数器[17, 40]。

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[32]。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、 和 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），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 进程跟踪过程中上下文切换的次数过多。如图3.2所示的调用拦截过程中，对于每个子线程系统调用需要执行四次上下文切换：两个 进程，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 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 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

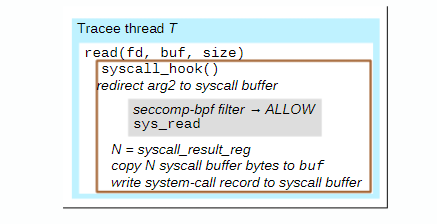
常用的在进程中拦截系统调用的技术是，使用动态链接在进行系统调用的C库函数上插入包装函数。在实践中，由于部分应用程序直接进行系统调用，并且由于C库函数的版本变化，这种方法具有很大的局限性。

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 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 监视并拦截系统调用。

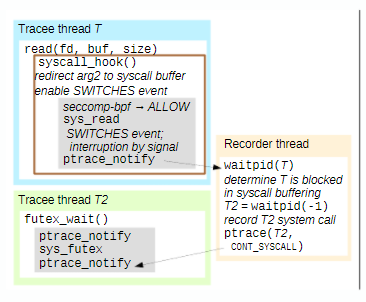
3.2.2 选择性拦截

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2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 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。

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）。因为容器一次运行一个子线程，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 说明了通过系统调用拦截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。在这个例子中T2完成一个（未被拦截的）阻塞 系统调用。

图表 1

3.3容器设计

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。

在外层容器中，使用标准化的容器工具Docker，将调用 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。

2层嵌套的容器化可重现方法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，并且不需要特权（root）访问。

3.3.1 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。

3.3.2 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（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。

和 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和的指令，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。 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。

本章将第3章中的可重现容器移植、优化到RISC-V架构硬件平台上。首先介绍了在RISC-V上实现Docker容器的过程，其次介绍基于QEMU的二进制翻译技术拦截系统调用的方法，最后介绍了一种在全系统模拟器Gem5中构建模拟RISC-V硬件平台的方法，最终完成基于RISC-V架构的容器化可重现方法设计与实现。

3.4 RISC-V 容器化可重现方法

上一章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，需解决2个问题：

（1） 在RISC-V架构上支持Docker容器；

（2） 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。

4.1.1RISC-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。

在本文中，我将首先展示如何拥有一个Risc-V虚拟机，在其中安装Golang和Docker，然后在这个环境中运行和构建容器。

从开发开始，我提供了一个基于Debian Sid的Risc-V虚拟机和一个完整的环境，在这个环境中，您可以开始在Risc-V架构上开发和构建应用程序。

在RISC-V架构上支持尚未上游。您可以查看此问题的进度。其中许多模块已经上游像x / sys和x / net。此外，许多库和应用程序已经支持VNDR，GitHub的集线器（Git Client），LabStack回声框架等RISC-V架构。在https://github.com/carlosedp/riscv-bringup上检查跟踪器。

要安装Go，请从此处下载tarball并使用命令安装：

启动虚拟机后，下载并安装Docker deb

安装后重新启动。现在你可以做docker info和docker version检查它是否工作(如果docker无法启动，只需运行sudo systemctl start dockeragain)。

4.1.1 基于QEMU的二进制翻译

作为 RISC-V 特权架构规范的一部分，RISC-V的硬件虚拟化支持是通过虚拟机管理模式（即H模式）规定的。基于目前H模式拓展规范，已经在QEMU中实现了KVM和Xvisor的完整功能。

QEMU通过内核代码的二进制翻译实现部分虚拟化功能。通过在宿主机和虚拟机之间添加一层中间层，将宿主机处理器的指令代码转换、翻译成目标处理器的指令，捕获文件执行时所需的系统调用。

ＱＥＭＵ 是一个具有跨平台的特性、可执行硬

件虚拟化的开源托管虚拟机，可通过纯软件方式实

现硬件的虚拟化，模拟外部硬件，为用户提供抽象、

虚拟的硬件环境。ＱＥＭＵ 既可实现全系统硬件虚

拟化，也可在 ＵｓｅｒＭｏｄｅ下通过为每个容器提供

特定的命名空间实现容器化设计。在 ＵｓｅｒＭｏｄｅ

下，ＱＥＭＵ 不会仿真所有硬件，而是通过内核代码

的 ＴＣＧ （ＴｉｎｙＣｏｄｅＧｅｎｅｒａｔｏｒ）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＿ｍｉｓｃ识别可执行文件格式并传递

至 ＱＥＭＵ。ｂｉｎｆｍｔ＿ｍｉｓｃ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 （ＩｎｔｅｒｍｅｄｉａｔｅＲｅｐｒｅ－

ｓｅｎｔａｔｉｏｎ），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 Ｈｙｐｅｒｖｉｓｏｒ执行代码

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

跳回到 Ｈｙｐｅｒｖｉｓｏｒ，Ｈｙｐｅｒｖｉｓｏｒ使用和协调 ＴＣＧ

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２为 ＴＣＧ 的工

作示意图。

第4章 实验设计与分析

本章在RISC-V平台上设计实验方案，通过并行程序基准测试集，验证方案设计的正确性，并对不同可重现方案产生的额外性能损耗进行分析比较。

5.1 实验平台

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，函数级（Function-Level）仿真有Spike[11]、QEMU[12]、FireSim[13]、RV8[14]；寄存器传输级（Register-Transfer-Level, RTL）仿真器包括Rocket[15]、BOOM[16]、Ariane[17]；FPGA级仿真模型Rocket Zedboard[18]等。函数级仿真速度快、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。因此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[21][22][23]，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，对多核条件下的可重现方法进行模拟验证。

5.1.1 Gem5全系统模拟器

Gem5是一个模块化的离散事件驱动全系统模拟器，它结合了M5和GEMS二者的优势[24]，且高度可配置、集成多种指令集架构和多种CPU模型的体系结构模拟器。Gem5广泛用于计算机体系结构研究，平衡仿真速度、准确性和开发速度。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[21][22]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。

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（通常是 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GNU/Linux Busybox发行版，Linux内核版本为5.10[23]。

本章在Gem5模拟器上模拟了一个RISC-V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4.1.2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。

5.1.2 RISC-V目标系统构建

目标是构建一个可以根据用户需求轻松扩展的基线 RISC-V 系统。此目标系统具有一组核心硬件源，包括 最少的外围设备。它能够运行系统软件，例如带有引导 加载程序和Linux内核的系统软件。目标系统的硬件配 置如图 1 所示，其中只显示了感兴趣的模块。除了总 线子系统外，还有两个主要子系统：CPU和HiFive Pl atform。橙色方框代表的块是我们工作中新添加的用 于成功启动系统软件的设备。在 CPU 子系统中，添加 了一个额外的 MMU 组件 PMA 检查器。HiFive 平台 基于 SiFive 的 HiFive 系列板，包含最少的关键外 围设备集。内核本地中断控制器 (CLINT) 处理软件和 定时器通过 MMIO 接口中断。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 路由到硬件线程。UART 和 VirtIOMMIO 对于内核启 动不是必需的，但对于可用的操作系统是必不可少的 。UART 提供交互式命令行终端，而 Vir tIOMMIO 提 供包含工作负载脚本和操作系统二进制文件的写时复 制根文件系统。

图 2 显示了 gem5 全系统仿真中由多个软件层组成 的堆栈。图中的 gem5 (FS) 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 ISA 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 FS 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。 我们的 gem5 RISC-V FS 模拟还支持不需要硬件辅 助虚拟化的管理程序，即 RISC-V H 扩展。Diosix 就是这样一个虚拟机管理程序。

5.1.3 HiFive平台

在本节中，我们将介绍新添加的设备或硬件模块的实 现细节。UART 模块内置于 gem5 中，而 VirtIOMMI O 模型是从 ARM 设置。重点放在平台和其他设备上，如 CLINT、 PLIC 和 PMAChecker。我们还讨论了特权指令和 C PU 模型的其他修复。本节由对检查点和设备树的支 持关闭。

在 gem5 中，系统配置被组织到称为平台的容器类中。 平台类是一个父类，具有一组标准化的外围设备和实用 功能，可以以分层方式扩展以自定义特定板的设置 / 系统。在 ARM 中，常见的 Platform 类是 RealV iew，而在 X86 中，常见的 Platform 类是 PC。在 RISC-V 中，我们将平台命名为 HiFive，对应 SiFi ve 的 HiFive 系列板卡。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 平台的设计易于 扩展，只需对来自其他 ISA 的设备进行移植所需的更 改最少。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 列表中并自动建立必要的连接。

我们使用 QEMU 作为完全相 同的执行路径的参考。使用相同 DTS 文件中描述的相 同系统设置，我们并排启动 QEMU 模拟器和 gem5 全系 统模拟器，并收集两种不同格式的执行跟踪。然后，我们 使用该工具包解析两条轨迹并在执行路径上进行比较， 以找到两条轨迹分歧的翻译块的位置。随后，我们使用上 述增强的远程 GDB 自动将断点插入块中，以识别错误 来自何处。

5.1.4 启动Linux系统

为了验证我们的实现，我们在第 2 节中使用 Berkele y 引导加载程序和 Linux 内核 v5.10 启动了目标系 统。为简单起见，该系统由四个 CPU 内核组成，每个内核 都有一个硬件线程。使用的文件系统是 BusyBox 磁盘映 像的一个端口。Linux 系统已在 gem5 提供的所有广泛 使用的四种 CPU 模型下成功启动：Atomic Simple、T iming Simple、Minor 和 DerivO3。我们进一步登录系 统并使用终端执行命令。BusyBox 中的命令可以正常运 行。检查点和恢复功能也已通过切换 CPU 型号进行了测 试。命令的这种正确功能表明内核的进程管理和调度程 序正在工作。由于调度器依赖于 CLINT 的定时器中断， 我们确信 CLINT 的定时功能得到了正确的实现。读取、 写入和移动文件的能力也证明了文件系统的正确功能， 它是 VirtIOMMIO 设备的控制器。此外，

交互终端功能正常，说明UART和PLIC模型配置正确 。

Linux OS 最重要的组件，包括进程管理、内存管理、 设备驱动程序和进程间通信，都已检查正常工作。我们 相信目标系统和 FS 模拟支持已正确实施。5.2实验环境配置

配置流程。

5.3 可重现性方法功能验证

运行包含getid()、等的程序。

5.4 可重现方法性能损耗分析

重复调用，记录执行时间。

第5章 总结与展望

……

……

致 谢

……

xxx（学生姓名落款）

年 月

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. 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.
14. 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.
15. 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.