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：**确定性；RISC-V；记录重放；容器化

ABSTRACT

……

**Key Words:**

目 录

[第1章 引言 1](#_Toc98762608)

[1.1 研究背景与意义 1](#_Toc98762609)

[1.2 国内外研究现状 2](#_Toc98762610)

[1.2.1 基于程序的可重现性研究——确定性执行 2](#_Toc98762611)

[1.2.2 基于环境的可重现性研究——确定性操作系统 3](#_Toc98762612)

[1.2.3 基于虚拟化的可重现性研究 4](#_Toc98762613)

[1.3 主要研究内容 5](#_Toc98762614)

[1.4 组织结构 6](#_Toc98762615)

[第2章 RISC-V指令集与可重现性研究 7](#_Toc98762616)

[2.1 RISC-V指令架构 7](#_Toc98762617)

[2.1.1 RISC-V基础指令集 9](#_Toc98762618)

[2.1.2 RISC-V拓展指令集 10](#_Toc98762619)

[2.1.3 RV32/64特权架构 10](#_Toc98762620)

[2.2 可重现性技术 12](#_Toc98762621)

[2.2.1 确定性模型 13](#_Toc98762622)

[2.2.2 不确定因素来源 14](#_Toc98762623)

[2.2.3 确定性重放实现方法 15](#_Toc98762624)

[2.2.4 虚拟化与容器 15](#_Toc98762625)

[第3章 容器化可重现方法设计与实现 16](#_Toc98762626)

[3.1 基于ptrace的系统调用拦截 16](#_Toc98762627)

[3.2 可重现方法工作流程 17](#_Toc98762628)

[3.3 不确定性来源 17](#_Toc98762629)

[3.3.1 用户进程编号 17](#_Toc98762630)

[3.3.2 随机函数 17](#_Toc98762631)

[3.3.3 时间 18](#_Toc98762632)

[3.3.4 信号 18](#_Toc98762633)

[3.3.5 文件和目录 18](#_Toc98762634)

[3.4 容器中的不确定性来源 18](#_Toc98762635)

[第4章 基于RISC-V架构的可重现容器化设计与实现 20](#_Toc98762636)

[4.1 基于Gem5的RISC-V全系统仿真 20](#_Toc98762637)

[4.1.1 Gem5全系统模拟器 20](#_Toc98762638)

[4.1.2 RISC-V目标系统构建 21](#_Toc98762639)

[4.2 RISC-V容器化方法 21](#_Toc98762640)

[4.2.1 基于命名空间的容器架构 22](#_Toc98762641)

[4.2.2 基于QEMU的模拟器 22](#_Toc98762642)

[4.2.3 动态二进制指令翻译 22](#_Toc98762643)

[4.3 RISC-V容器化方法中的不确定性来源 22](#_Toc98762644)

[4.4 RISC-V特权级切换 22](#_Toc98762645)

[第5章 实验设计与分析 23](#_Toc98762646)

[5.1 软硬件平台 23](#_Toc98762647)

[5.2实验环境配置 23](#_Toc98762648)

[5.3 可重现性方法功能验证 23](#_Toc98762649)

[5.4 可重现方法性能损耗分析 23](#_Toc98762650)

[第6章 总结与展望 24](#_Toc98762651)

[致 谢 25](#_Toc98762652)

[参考文献 26](#_Toc98762653)

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，标准微处理器性能提升的减速已成为了既定事实，体系结构在新的黄金时代需要寻求新的前进方向[1]。加州大学伯克利分校提出了RISC-V（RISC Five），即第五代RISC架构。RISC-V并非是精简指令集简单的版本迭代，和前代相比它最大的优势在于开源和模块化，允许用户基于特定需求添加定制化拓展指令集。RISC-V由于其高度的灵活性在工业界和学术界均受到广泛关注，推出了一系列支持乱序执行的微处理器，如BROOM等[2]，将会应用在可穿戴设备、智能家居、机器人、自动驾驶及工业装置等领域的计算设备中，在边缘微设备的应用中具有广阔的前景。

随着RISC-V软件生态的日益完善，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；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，可重现性能够确保模型结果正确，帮助开发人员寻找模型性能变化的原因。因此，以较低的额外开销确保程序的可重现性在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[10]：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）技术实现程序的重现性。确定性重放技术通过记录并重现程序错误，尽可能避免不确定性因素对程序的影响。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是查找并拦截不确定性（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，记录并重现它们的调用，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[4][5][6]。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[7][8]，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[3]；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。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，设计实现一种轻量级的容器化可重现方法，在程序执行期间通过容器隔离不确定性的CPU指令和系统调用，同时满足对确定性和可移植性的要求，并在RISC-V架构下完成该方法的实验与分析。

1.2 国内外研究现状

国内外研究人员对可重现性进行了深入的研究，根据研究方向、内容和方法大致可以划分为三个方向，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

可重现性的需求在19世纪80年代后期就引起了研究人员的注意，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，最早应用于并行程序调试过程中，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等人提出了Instant Replay[1]，作为一种重现并行程序执行行为的一般解决方案。在当时的软件调试过程中，顺序程序可以通过循环执行的方式，发现和纠正程序执行中的错误。但是对并行程序的2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，保存程序重放所需的信息。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，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）发生的时间。相较于前者，Recap实现了从某一个检查点重放的功能，不必从程序头重新执行参加鞥下。但是Recap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等人通过记录指令计数器保存共享内存访问的总顺序，实现了基于硬件辅助机制的确定性重放技术。

确定性重放工具Bugnet[3]能够记录外部I/O事件并重放并行程序，但Bugnet只支持特定API；

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；

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。

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。

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。

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

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。

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：

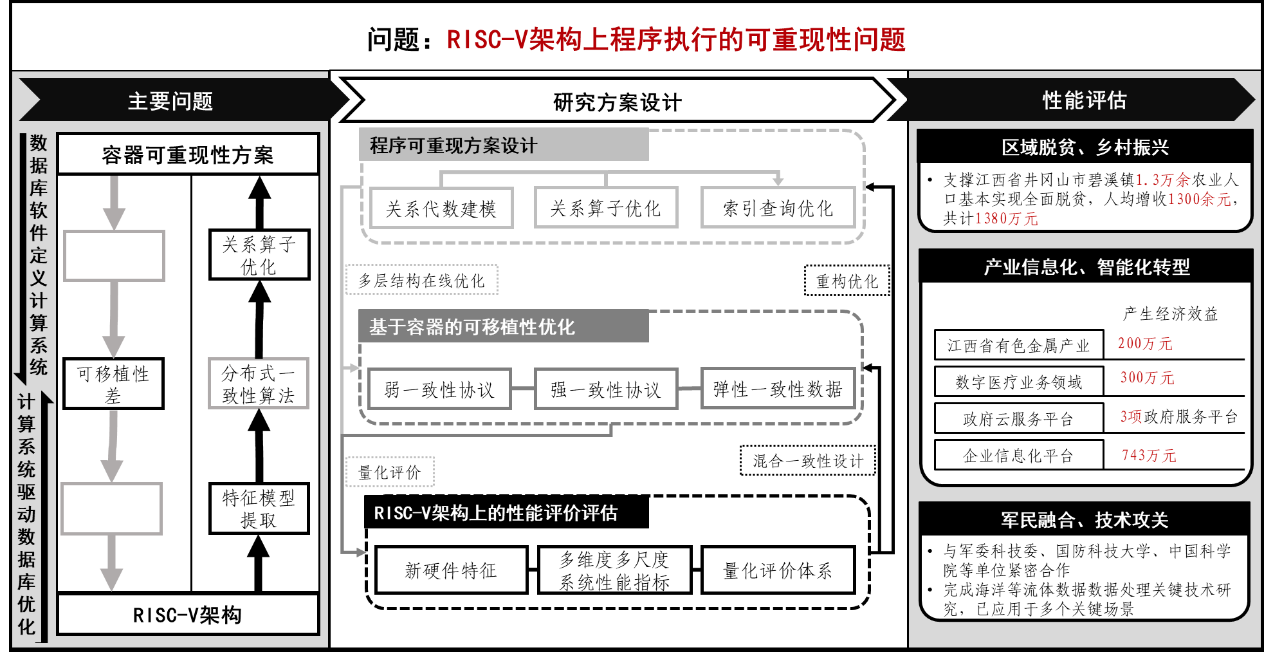


图1.1 基于RISC-V的可重现容器化方法研究路线示意图

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，每个章节内容安排如下：

第一章：引言。主要介绍基于RISC-V架构的容器化可重现方法的研究背景和意义，根据现有研究内容，将研究问题划分为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。介绍了国内外相关工作以及本文主要工作，并介绍了本文的组织结构。

第二章：RISC-V指令集与可重现性研究。详细介绍了本课题所采用的基础理论和相关技术，主要包括RISC-V指令集设计思想及其特权架构、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。

第四章：在第二章提出的容器化可重现方法基础上，设计并实现基于RISC-V架构的容器化可重现方法可移植性优化。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，并进一步讨论未来研究方向。

第2章 RISC-V指令集与可重现性研究

本章介绍详细介绍了本课题所采用的基础理论和相关技术，主要包括RISC-V指令集设计思想及其特权架构、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

RISC-V是一种新兴的开源精简指令集架构，由加州大学伯克利分校在2010年首次发布。为了避免x86、ARM、MIPS等现有体系结构长期发展暴露出的种种问题，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。

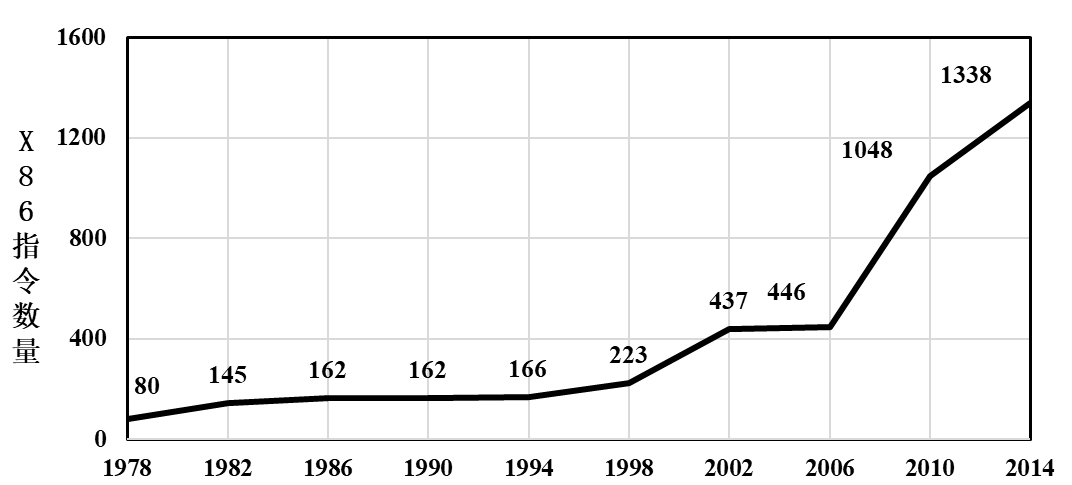
**（2）模块化设计**

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，即新处理器不仅必须实现新的指令集拓展，还需要支持过去的所有拓展，如x86指令集架构的64位拓展必须向后兼容32位甚至16位的x86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，提供大量自定义编码空间支持对指令集进行拓展，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，体现了强大的系统可定制化能力。

**（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指令的设计，降低了指令集文档的复杂程度，如表2.1所示

表2.1 各指令集架构规范手册对比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指令集架构 | 规范手册 | | 阅读时间 | |
| 页数 | 字数 | 小时 | 周 |
| RISC-V | 236 | 76702 | 6 | 0.2 |
| ARM-32 | 2736 | 895032 | 79 | 1.9 |
| X86-32 | 2198 | 2186259 | 182 | 4.5 |

由于其开放和免费的架构，以及可伸缩性、可拓展性和定制化的潜力，RISC-V已经被广泛应用于各个特定领域的微处理器设计中，如西部数据公司研发的基于RISC-V的通用架构 SweRV[13]、阿里巴巴公司研发的 64 位高性能嵌入式RISC-V处理器Xuantie-910[14]、Koch等人设计的嵌入式开源FPGA框架FABulous[15]、中国科学院计算技术研究所在RISC-V中国峰会发布的开源高性能RISC-V处理器核“香山”[16]、上海交通大学开源的基于RISC-V的可信执行环境安全系统“蓬莱”[17]。其中也包括了对 RISC-V 与不同应用领域结合方式的探索，如 Kadomoto等人[18]利用RISC-V芯片改善了无线总线接口技术，以促进对于小型机器人的研究；Di Tucci等人[19]将RISC-V应用于基因组处理，提出了专用领域架构SALSA。这些系统、工具利用RISC-V在资源依赖性、低功耗性、易用性、可定制性、可扩展性等方面的优势，能够高效、迅速而低成本地完成各自领域中的系统级任务。

2.1.1 RISC-V基础指令集

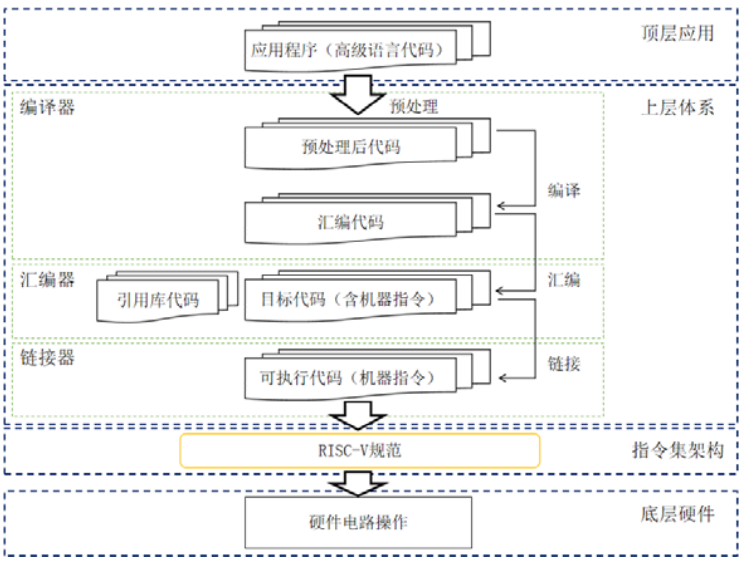


图2.3 RISC-V指令集架构在计算机体系结构中的层次

RISC-V的基础指令集提供了基本整数指令集，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）类似，但是不包括分支延迟间隙和可变指令编码。每个基本整数指令集提供了一组能够为编译器、汇编器、链接器、操作系统（结合额外特权操作）等提供必要功能实现的最小指令集合，围绕基本整数指令集能够构建定制处理器指令集架构实现。任何一种RISC-V指令集架构都必须完整地实现一种基础指令集。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，分别提供了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，用于未来的128位处理器设计。

**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，32位基本整数指令集使用32个通用寄存器（ 寄存器），和一个非特权寄存器(寄存器)，如图2.4（a）所示。标准调用约定了所有寄存器功能，其中， 寄存器所有位被强制硬件布线为0，也被称作零寄存器（zero）； 寄存器用于保存当前指令地址，又称作程序计数器；通用寄存器 被用作保存程序运行相关指针，共4个； 和用作临时寄存器，共7个；及 由被调用者使用，共12个； 用于保存调用的参数，共8个。

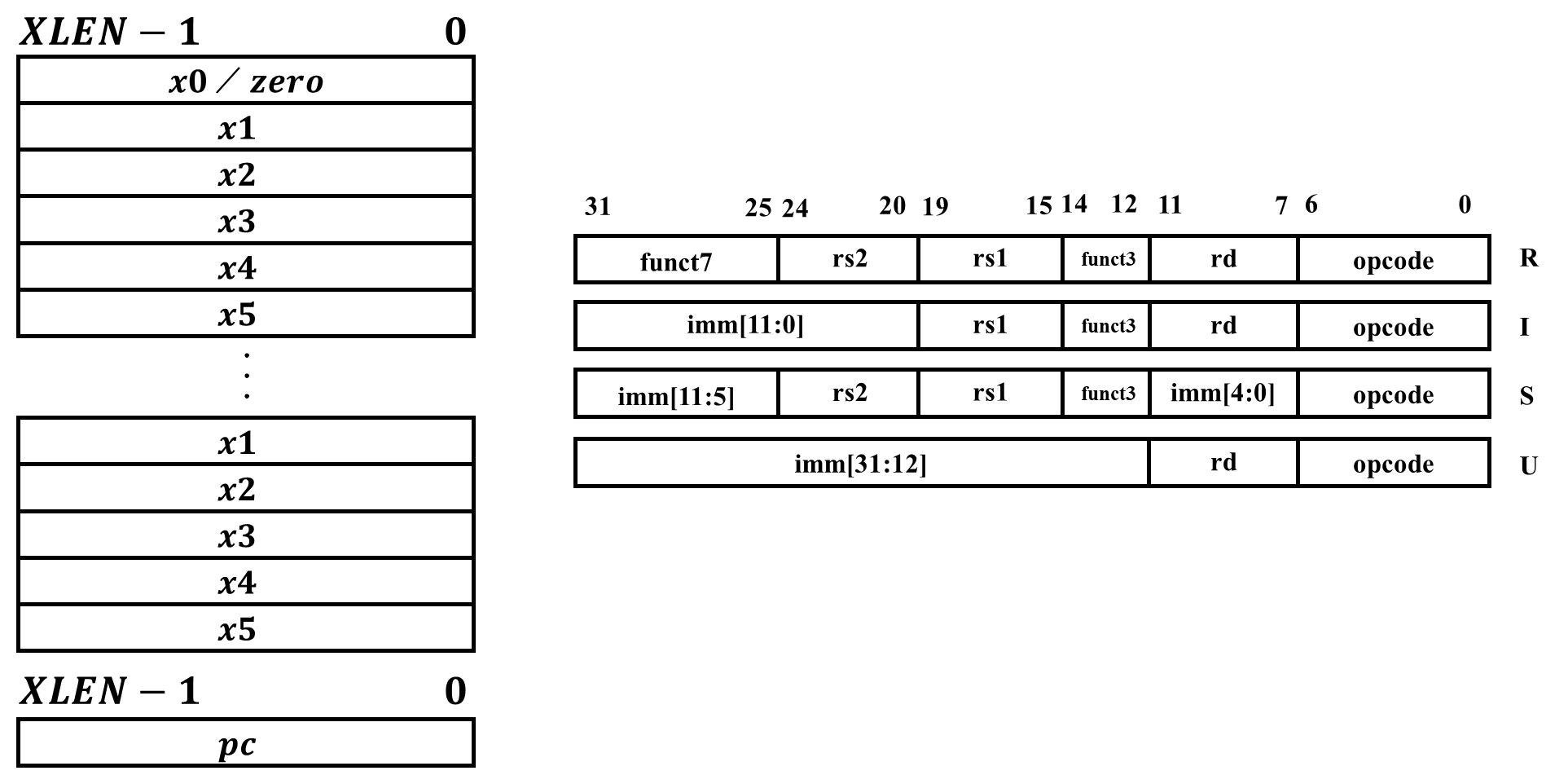


图2.4 （a）RV32I寄存器状态 （b）RV32I的4种基础指令格式

RV32I有4种基础指令格式：R/I/S/U。指令集中的任何指令都可以根据操作数的数量、种类、规模以及自身的功能需求，选用其中一种格式。所有这些指令格式都是32位固定长度，并且必须在内存中对齐到4字节的边界。图2.4（b）显示了这 4 种基础的指令格式。

RV64I同样使用32个通用寄存器和4种基础指令格式，但是将整数寄存器和所支持的用户地址空间扩展到了64位。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，将可用的整数寄存器数目从32减少到16，即只使用 和 完成所有指令功能，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，将整数寄存器宽度拓展为128位，保留了指令，新增了用于操作低64位的指令。

RVWMO指令集定义了RISC-V的内存一致性模型，主要遵循RC（Release Consistency）理论模型，用较少的内存访问顺序约束。RVWMO 对程序的执行提出了次序上的要求即在一个多线程程序有多种不同可能的执行，而每种执行有其自己相应的全局内存次序。这里的“全局内存次序”是指所有硬件线程所产生的内存操作的总体次序。RVWMO 指令集为加载和存储 等内存操作提供了一组内存模型原语，用于对相关指令进行标注。表2.3列举了这些标注的内容及其含义。

表2.3 RVWMO指令集对内存操作指令的标注

|  |  |  |
| --- | --- | --- |
|  | 与处理器一致 | 顺序一致 |
| 加载 | acquire-RCpc | acquire-RCsc |
| 释放 | release-RCpc | release-RCsc |

2.1.2 RISC-V拓展指令集

RISC-V的扩展指令集用于为ISA提供特定方面的功能操作指令，现有的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

RISC-V指令集架构拥有4种权限模式，除用户模式（User Mode，U模式）以外，RISC-V设计了两种具有更高权限的模式：机器模式（Machine Mode，M模式）、管理模式（Supervisor Mode，缩写为S模式）以及监视模式（Hypervisor Mode，缩写为H模式），其中H模式暂时处于草案状态。

RISC-V通过

M-mode被设计用来拦截和处理异常。中断和异常是不确定性的主要来源。S-mode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-mode的权限基于U-mode和M-mode之间，不能使用M-mode下的CSR和指令。

机器模式（缩写为 M 模式，M-mode）是 RISC-V 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 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。实际上简单的 RISC-V 微控制器仅支持 M 模式。这类系统是本节的重点。

机器模式最重要的特性是拦截和处理异常（不寻常的运行时事件）的能力。RISC-V 将异常分为两类。一类是同步异常，这类异常在指令执行期间产生，如访问了无效的存储器地址或执行了具有无效操作码的指令时。另一类是中断，它是与指令流异步的外部事件，比如鼠标的单击。RISC-V 中实现精确例外：保证异常之前的所有指令都完整地执行了，而后续的指令都没有开始执行（或等同于没有执行）。图 10.3 列出了触发标准例外的原因。

在 M 模式运行期间可能发生的同步例外有五种：

⚫ 访问错误异常 当物理内存的地址不支持访问类型时发生（例如尝试写入 ROM）。

⚫ 断点异常 在执行 ebreak 指令，或者地址或数据与调试触发器匹配时发生。

⚫ 环境调用异常 在执行 ecall 指令时发生。

⚫ 非法指令异常 在译码阶段发现无效操作码时发生。

⚫ 非对齐地址异常 在有效地址不能被访问大小整除时发生，例如地址为 0x12 的amoadd.w。

有三种标准的中断源：软件、时钟和外部来源。软件中断通过向内存映射寄存器中存数来触发，并通常用于由一个 hart 中断另一个 hart（在其他架构中称为处理器间中断机制）。当 hart 的时间比较器（一个名为 mtimecmp 的内存映射寄存器）大于实时计数器mtime 时，会触发时钟中断。外部中断由平台级中断控制器（大多数外部设备连接到这个中断控制器）引发。不同的硬件平台具有不同的内存映射并且需要中断控制器的不同特性，因此用于发出和消除这些中断的机制因平台而异。

八个控制状态寄存器（CSR）是机器模式下异常处理的必要部分：

⚫ mtvec（Machine Trap Vector）它保存发生异常时处理器需要跳转到的地址。

⚫ mepc（Machine Exception PC）它指向发生异常的指令。

⚫ mcause（Machine Exception Cause）它指示发生异常的种类。

⚫ mie（Machine Interrupt Enable）它指出处理器目前能处理和必须忽略的中断。

⚫ mip（Machine Interrupt Pending）它列出目前正准备处理的中断。

⚫ mtval（Machine Trap Value）它保存了陷入（trap）的附加信息：地址例外中出错

的地址、发生非法指令例外的指令本身，对于其他异常，它的值为 0。

⚫ mscratch（Machine Scratch）它暂时存放一个字大小的数据。

⚫ mstatus（Machine Status）它保存全局中断使能，以及许多其他的状态，如图

10.4 所示。

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）并行程序的共享内存访问；（2）中断、异常等外部信号；（3）部分带有随机性质的系统调用等。确定性重放技术一般通过在第一次执行某个应用程序的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，以期获得与记录阶段一致的执行结果。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。

2.2.1 确定性模型

确定性重放也常被成为记录重放（Record and Replay），即在时间或空间上重复执行一个程序，执行的副本作为原程序的镜像，在相同的输入下应当产生与之相同的输出。数值计算型程序多次执行结果必然相同，但是涉及到系统时间、文件输入、缓冲区等因素影响的程序，执行结果会受到多种因素的影响。因此需要通过记录重放的方法使程序反复执行得出相同的结果。除了并行程序调试以外，确定性重放技术还被应用于并行安全性和可靠性检查[2]、性能预测[3]等领域。

确定性重放实现的关键思想是对不确定性因素的记录。然而，随着多核架构中的并行程序带来更多不确定因素，而且某些因素出现的频率非常高，给确定性重放的实现带来了很多困难。确定性作为一个难以量化评价的概念，研究人员在设计和应用各种确定性重放方案时，对不同程度的确定性做出了定义。

(1) 理想确定性

理想确定性保证重放阶段中所有线程指令执行顺序、线程间指令执行顺序与记录阶段保持严格一致，包括所有进程、线程、访存等重要事件。理想确定性需要记录并重放程序执行阶段的系统状态、中断和异常等信息，由于需要存储大量信息，存储开销较大，也是最理想、最高程度的确定性。

(2) 严格确定性

严格确定性保证重放阶段中所有线程指令执行顺序与记录阶段保持严格一致，且每个线程访问共享内存器返回的结果与记录阶段相同。无需考虑除访存指令之外的其他指令的执行顺序及每条指令的执行时间，而只需保证访存相关指令的执行顺序即可。严格确定性足以支持所有并行程序的确定性重放,满足应用需求。

(3) 非严格确定性

非严格确定性在严格确定性基础上减少时空开销。具体包括 3 种：（1）外部确定性，只保证重放阶段中外部可见的状态与记录阶段相同，要求重放阶段的寄存器文件状态、程序输出、系统调用的顺序及传递的参数应与记录阶段相一致，但并不要求线程间指令交错执行的顺序一致，而只保证程序运行中的外部可见状态相同；（2）部分确定性，保证程序的一部分能够被确定性地重新执行，而对于其他部分则无法实现确定性重放；（3）输出确定性，只保证重放阶段最终的输出结果与记录阶段相同.由于只要求输出结果相同，对某些线程间指令的交错运行顺序以及其他程序状态可以不予考虑和检测。

2.2.2 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，如图所示

**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. 中断与异常**

外部与异常机制是CP对外部信号做出的一种反应，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。

**3. 输入输出操作**

输入输出操作会改变设备状态，由于对硬件设备寄存器的读写不可逆，输入输出操作反复执行程序可能产生不同的结果。

**4. 共享存储器竞争**

多核架构下的并行程序对共享存储器的访问频率很高，来自不同进程的访存指令访问共享存储器时会产生新的不确定性。共享存储器竞争可分为同步（Synchronization）和数据竞争（Data Race）两种，同步操作的确定性重放实现较为简单，但会带来较大的时空开销；数据竞争的确定性重放更为复杂。

**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。

**7. CPU指令**

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。

2.2.3 确定性重放实现方法

确定性重放可分别通过硬件支持方法和纯软件实现。

硬件方法：修改体系结构、设计专用芯片等。成本高、增加功耗、灵活性低。

纯软件方法：

系统级别

进程级别

2.2.4 虚拟化与容器

第3章 容器化可重现方法设计与实现

在x86平台上，设计容器化可重现方法。使用纯软件用户空间的方式隔离容器中的程序，以系统调用getpid（获取进程编号PID）为例，使用ptrace监视并拦截容器中运行中程序（称为用户进程）的系统调用。

DetTrace 将轻量级沙盒容器与系统调用拦截 相结合，以实现任意 Linux 程序的可重复性 强制执行。DetTrace 实现了这一功能，同时满 足了我们的设计目标：纯软件用户空间解决方 案，支持未修改的二进制文件，不需要特权（r oot）访问，并且不需要记录和重放。DetTrac e 使用标准的 Linux 容器功能：用户、PID 和挂载名称、绑定挂载和 chroot。这些机制有 助于将容器中的程序与其外部的程序和文件隔离开来。

3.1 基于ptrace的系统调用拦截

DetTrace 使用 ptrace 拦截容器中运行的代 码进行的所有系统调用。Linux ptrace 机制允 许一个进程（跟踪器）监视另一个进程（跟踪器 ）的执行。跟踪器可以拦截被跟踪者的系统调用 （在它们到达内核之前和返回到被跟踪者之前） 、信号等等。跟踪器还可以读取和写入跟踪内存 和寄存器。由于跟踪器是它自己的进程，因此它 与跟踪器故障隔离良好（反之亦然）。但是，每次 拦截的事件都需要额外的上下文切换才能跳转 到跟踪器。在 DetTrace 中，具有可重现语义的 系统调用

我们使用 ptrace 意味着我们可以看到从容器中进行的所有系统调用，因此没有潜在的关 键系统调用（我们还处理 vDSO 调用，参见第 5.3 节）。如果给定的系统调用是 ir 可重复 性的来源，则有许多潜在的缓解措施：包装系 统调用或完全用确定性对应物（如时间调用） 替换它，将其转换为 nop（如睡眠调用），或不 支持它并引发（可重现的）容器级错误。

定义：

ptrace()系统调用函数提供了一个进程（the “tracer”）监察和控制另一个进程（the “tracee”）的方法。并且可以检查和改变“tracee”进程的内存和寄存器里的数据。它可以用来实现断点调试和系统调用跟踪。

3.2 可重现方法工作流程

通过用户空间隔离进程，使用追踪进程tracer可以拦截被追踪用户进程tracee的系统调用，读取和写入进程B的内存与寄存器。其中可重现的元素被允许通过，不可重现的元素被进行可重现的包装或者被禁止进入容器空间。

3.3 不确定性来源

特权指令通常是不可重现的，但会在 我们的用户级容器中引发异常。一些不可重现的 用户级 x86-64 指令虽然可能，但很难捕获。rd rand 和 rdseed 从硬件熵源返回随机位，并且可 以通过 VT-x 扩展在管理程序级别捕获，但不能 从环 0 捕获。有时可以从用户空间访问像 rdp mc（从性能计数器读取）这样的指令但可以通过适 当的内核设置配置为导致陷阱。

3.3.1 用户进程编号

通过命名空间隔离进程，获取惟一PID等。

3.3.2 随机函数

拦截具有随机性的系统调用函数，用简单伪随机函数替换。

3.3.3 时间

返回时间信息的系统调用，如，替换为用户进程执行的时间调用的计数。

3.3.4 信号

异步信号。

部分信号是天然可重现的：, 和。

3.3.5 文件和目录

通过隔离用户进程拥有的文件系统。

作用于正在运行的用户进程和它的子进程，改变它外显的根目录，设置后的用户进程不能够对这个指定根目录之外的文件进行访问动作，不能读取，也不能更改它的内容。

3.4 容器中的不确定性来源

现有的容器技术（如 Docker）不提供可重复性 ：它们既不是确定性的也不是可移植的，因为主 机操作系统和处理器微架构的许多细节在容器 内是直接可见的。虚拟机提供了更强的硬件抽 象，但缺乏确定性，而且重量也很大。我们相信 ，DetTrace 可重现容器抽象为构建和测试可 重现性至关重要的软件等领域的现有方法提供 了显着优势。

x86-64 指令集和 Linu x 系统调用 API。因为我们对容器中的代码没有 任何限制，它可以包含任意指令并尝试任意系统 调用。受 Popek 和 Goldberg 虚拟化要求 [ 27] 的启发，这些要求定义了提供虚拟机抽象的 要求，我们定义了再现性要求集。我们分析每个 记录在案的 x86-64 ISA 指令 2 和系统调用 ，以查看它是否可能是不可再现性的来源，如果 是，在什么条件下。特别重要的是识别接口的关 键成员——那些允许不可重现但在执行过程中不 能可靠地检测到的成员。任何关键指令或系统调 用都可能默默地引入不可再现性。

第4章 基于RISC-V架构的可重现容器化设计与实现

本章提出了一种在资源受限的RISC-V架构平台上的容器化方法的设计与实现，并将第3章中的可重现容器化方案移植、优化到RISC-V架构平台上。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。

4.1 基于Gem5的RISC-V全系统仿真

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。

4.1.1 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。

4.1.2 RISC-V目标系统构建

4.2 RISC-V容器化方法

可移植性的要求

现有的RISC-V应用部署流程需要将传统软件或模型在RISC-V指令集上重新编译或优化，故如何能快速地在RISC-V体系结构上部署、运行及测试应用程序是一个亟待解决的技术挑战。使用虚拟化技术可以解决跨平台的模型部署及运行问题。但传统的虚拟化技术，例如虚拟机，对原生系统性能要求高、资源占用多，运行响应慢，往往不适用于RISC-V架构的应用场景。因此在本章提出了一种

4.2.1 基于命名空间的容器架构

4.2.2 基于QEMU的模拟器

4.2.3 动态二进制指令翻译

4.3 RISC-V容器化方法中的不确定性来源

4.4 RISC-V特权级切换

第5章 实验设计与分析

5.1 软硬件平台

模拟硬件：Gen5模拟器全系统模式下，模拟多核RISC-V处理器。

软件：Linux 4.12内核版本。

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6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. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. Aviram A, Weng S C, Hu S, et al. Efficient System-Enforced Deterministic Parallelism[C]. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10), 2010.
11. Hunt N, Bergan T, Ceze L, et al. DDOS: taming nondeterminism in distributed systems[J]. ACM SIGPLAN Notices, 2013, 48(4): 499-508.
12. 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.
13. 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.
14. A. Waterman and Y. Lee. Spike - RISC-V ISA Simulator[OL]. https://github.com/riscv/ riscv-isa-sim.
15. Bellard F. QEMU, a fast and portable dynamic translator[C]. USENIX annual technical conference, FREENIX Track. 2005, 41(46): 10.5555.
16. Karandikar S, Biancolin D, Amid A, et al. Using FireSim to Enable Agile End-to-End RISC-V Computer Architecture Research[C]. Workshop on Computer Architecture Research with RISC-V (CARRV). 2019.
17. Clark M, Hoult B. rv8: a high performance RISC-V to x86 binary translator[C]. First Workshop on Computer Architecture Research with RISC-V (CARRV). 2017.
18. 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.
19. 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.
20. Balkind J, Lim K, Gao F, et al. OpenPiton+ Ariane: The first open-source, SMP Linux-booting RISC-V system scaling from one to many cores[C]. Workshop on Computer Architecture Research with RISC-V (CARRV). 2019: 1-6.
21. Vega L, Taylor M B. RV-IOV: Tethering RISC-V Processors via Scalable I/O Virtualization[C]. Workshop on Computer Architecture Research with RISC-V (CARRV). 2017.
22. 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).
23. Ta T, Cheng L, Batten C. Simulating multi-core RISC-V systems in gem5[C]. Workshop on Computer Architecture Research with RISC-V (CARRV). 2018.
24. 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 (CARRV). 2021.
25. Butko A, Garibotti R, Ost L, et al. Accuracy evaluation of gem5 simulator system[C]. 7th International workshop on reconfigurable and communication-centric systems-on-chip (ReCoSoC). IEEE, 2012: 1-7.