送检文献信息

【题名】纯享版100

作　　者：崔傲

检测时间：2022-03-25 04:55:32

18.35%
总相似比

检测范围：

- [x] 中国学术期刊数据库
- [x] 优先出版论文数据库
- [x] 国内外重要学术会议论文数据库
- [x] 中国博士学位论文全文数据库
- [x] 中国优秀硕士学位论文全文数据库
- [x] 中国优秀报纸全文数据库
- [x] 互联网学术资源数据库
- [x] 学术网络文献数据库
- [x] 中国专利文献全文数据库
- [x] 特色英文文摘数据库
- [x] 中国标准全文数据库

详细检测结果

原文总字符数	检测字符数	参考文献相似比	辅助排除参考文献相似比	可能自引相似比	辅助排除可能自引相似比
20100	19486	0.00%	18.35%	5.03%	13.32%

相似文献列表（仅列举前10条）

序号	相似比(相似字符)	相似文献	类型	是否引用
1	6.59% 1284字符	RISC-V指令集架构研究综述 刘畅，武延军，吴敬征，赵琛；《软件学报》；2021-06-28	期刊	否
2	5.03% 980字符	基于RISC-V架构的强化学习容器化方法研究 徐子晨，崔傲，王玉皞，刘韬；《计算机工程与科学》；2020-05-03	期刊	否
3	2.78% 541字符	多核处理器并行程序的确定性重放研究? 高岚，王锐，钱德沛；《软件学报》；2012-07-01	期刊	否
4	1.01% 197字符	基于国产多核处理器的应用程序确定性在线重放技术研究 苟臻凡（导师：李振洲）；哈尔滨工业大学，硕士（专业：计算机科学与技术）；2015	学位	否
5	0.98% 190字符	面向知识图谱的多源数据图表征及检索优化方法研究 程婕（导师：徐子晨;卜象平）；南昌大学，硕士（专业：软件工程）；2021	学位	否
6	0.84% 163字符	可伸缩的确定性重放技术研究 陈宇飞（导师：臧斌宇）；复旦大学，博士（专业：计算机系统结构）；2014	学位	否
7	0.44% 85字符	基于系统调用的软件行为验证方法研究与实现 王焱济（导师：杨力）；西安电子科技大学，硕士（专业：计算机技术）；2019	学位	否
8	0.24% 47字符	[转载]GEM5 模拟器简介 李玉祥；http://www.sciencenet.cn/（网址：http://blog.sciencenet.cn/blog-875277-724855.html）；2013-09-14	学术网文	否
9	0.20% 38字符	面向二进制漏洞分析的软件执行迹提取与分析技术研究 夏源（导师：唐朝京）；国防科技大学，硕士（专业：信息与通信工程）；2018	学位	否
10	0.08% 16字符	基于数据流处理器结构的调度优化研究 李易（导师：范东睿）；中国科学院大学，博士（专业：计算机系统结构）；2021	学位	否

原文标注

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

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鞲下。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ptrace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

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 0.2ARM-32 2736 895032 79 1.9X86-32 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

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（x 寄存器），和一个非特权寄存器（pc 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，x0 寄存器所有位被强制硬件布线为0，也被称作零寄存器（zero）；pc 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x1 ~ x4 被用作保存程序运行相关指针，共4个；x5~x7 和x28 ~ x31用作临时寄存器，共7个；x8 ~ x9 及x18 ~ x27 由被调用者使用，共12个；x10 ~ x17 用于保存调用的参数，共8个。

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*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，将可用的整数寄存器数目从32减少到16，即只使用x0 ~ x15 和pc 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*W指令，新增了用于操作低64位的*D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（load） 和存储（store） 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

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 监视模式 1.0 正式批准

1. 用户模式

用户模式（User Mode，U模式）是RISC-V特权系统中最低级别的权限模式，通常用于执行来自用户等外部环境的不可信操作，通过对其操作范围的限制来保护系统内的各种资源不受侵害。

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）具有比用户级更高的操作权限，可以用于操作一台机器中的敏感资源。RISC-V指令架构中的S模式需要与M模式和U模式共同实现，因此，不能出现系统中只存在S模式而不存在U模式的情况。

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. 监视模式

监视模式（Hypervisor Mode，M模式）可用于管理跨机器的资源，或者将机器整体作为组件承担更高级别的任务。H 模式可以协助实现一台机器系统的虚拟化操作。

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getpid、获取计时器值getitimer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。

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

现有的确定性重放方法的实现方法按照载体可分为基于软件的方法和基于硬件的方法。

基于硬件的确定性重放方法多采用虚拟化方案来记录和重放不确定性事件，如[24]利用处理器的恢复寄存器（Recovery Register）来限制中断只能在程序执行的特定点注入到用户系统，简化了中断的处理。利用程序计数器（Program Counter）和分支计数器（Branch Counter）标记程序执行时的指令顺序。基于硬件实现确定性重放的方法的挑战在于记录共享内存访问顺序，如通过监听缓存一致协议，在缓存一致消息中添加指令计数器来记录共享内存顺序。

基于软件的确定性重放方法通过记录库函数和系统调用的效果来实现重放，但是由于不确定性因素来源较广，此种方法不具有普遍性，难以处理所有类型的应用程序。

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ptrace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：

3.1 可重现方法工作流程

大多数低开销的记录和回放系统都依赖于观察到 CPU 大多是确定性的。我们确定了围绕状态和计算的边界，记录边界内的所有非确定性来源和所有进入边界的输入，并通过重放非确定性和输入来重新执行边界内的计算。如果确实捕获了所有输入和不确定性，则重放期间边界内的状态和计算将与记录期间的状态和计算相匹配。

为了启用任意 Linux 应用程序的记录和回放，而不需要内核修改或虚拟机，RR 记录和回放一组进程的用户空间执行。为了简化不变量，并使重放尽可能忠实，重放几乎保留了用户空间执行的每一个细节。特别是，用户空间内存和寄存器值被准确地保留了下来，本文后面会提到一些例外情况。这意味着记录和回放之间的 CPU 级控制流是相同的，内存布局也是如此。

虽然重放保留了用户空间状态和执行，但在重放期间只复制了最少量的内核状态。例如，不打开文件描述符，不安装信号处理程序，不执行文件系统操作。相反，这些操作的记录的用户空间可见效果以及未来相关的操作会被重放。我们确实为每个记录的线程创建一个重播线程（不是绝对必要的），并且我们为每个记录的地址空间创建一个重播地址空间（即进程），以及匹配的内存映射。

通过这种设计，我们的记录边界是用户空间和内核之间的接口。非确定性的输入和来源主要是系统调用的结果和异步事件的时序。

3.1.1 避免数据竞争

由于线程在多个内核上运行，不同线程对同一内存位置的竞速读写访问将是不确定性的来源。 因此，我们采用通用方法 [17, 28, 1, 15] 一次只运行一个线程。RR 抢先调度这些线程， 因此上下文切换时间是必须记录的不确定性。 如果上下文切换发生在执行中的正确点，仍然可以观察到数据竞争错误（尽管无法观察到由于弱内存模型导致的错误）。

这种方法比替代方案 [7, 18, 34, 39, 29] 更简单，更易于部署，避免假设程序是无竞争的 [14, 29]，并且对于低并行度工作负载有效 。对于具有持续高度并行性的工作负载，速度会大幅下降；然而，即使对于可能高度并行的应用程序，用户也经常应用 RR 来测试具有相对较小数据集的工作负载，因此并行性有限。

3.2.2 拦截系统调用

系统调用通过修改寄存器和内存将数据返回到用户空间，这些变化必须记录下来。ptrace 系统调用允许进程监督其他“tracee”进程和线程的执行，并在tracee 线程进入或退出系统调用时得到同步通知。当跟踪线程进入内核进行系统调用时，它会被挂起并通知 RR。当 RR 选择再次运行该线程时，系统调用将完成，再次通知 RR，使其有机会记录系统调用结果。RR 包含大多数 Li nux 系统调用的模型，描述了他们可以修改的用户空间内存，给定系统调用输入参数和结果。

如上所述，RR 通常通过一次只调度一个线程来避免竞争。但是，如果内核中的系统调用阻塞， RR 必须尝试调度其他应用程序线程在阻塞系统调用完成时运行。正在运行的线程有可能（尽管不太可能）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，我们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”，否则应用程序将不会使用它。 当我们收到一个阻塞系统调用完成的 ptrace 事件时，RR 将暂存缓冲区内容复制到真正的用户空间目标，而没有其他线程在运行，从而消除了竞争。

图 1 说明了记录一个简单的读取系统调用。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被拦截的系统调用时，我们在系统调用指令的地址处设置一个临时断点（记录在trace中）。我们使用 p trace 运行 tracee 线程，直到它遇到断点， 移除断点，将程序计数器推进到系统调用指令之后，并应用记录的寄存器和内存更改。这种方法很简单，并且最大限度地减少了 RR 和跟踪线程之间的上下文切换次数。（有时它是不安全的 ，我们会退回到更复杂的机制。）

一些系统调用操作线程或地址空格并在重播期间需要特殊处理。例如，使用 MAP FIXED 重播记录的 mmap，以确保在正确的地址创建映射。

3.2.3 异步事件

我们需要支持两种异步事件：抢占式上下文切换和信号。我们将前者视为后者的特例，通过向正在运行的跟踪线程发送信号来强制进行上下文切换。我们需要确保在回放期间，当节目处于与录制期间传递信号时完全相同的状态时传递信号。 与之前的工作 [17, 33, 10] 一样，我们使用 CPU 硬件性能计数器测量应用程序进度。理想情况下，我们会在记录期间计算导致异步事件的已退出指令，并且在重播程序期间，CPU 在许多指令已退出后触发中断——但这种方法需要修改才能在实践中工作。

与之前的工作 [17, 33, 10] 一样，我们使用 CPU 硬件性能计数器测量应用程序进度。理想情况下，我们会在记录期间计算导致异步事件的已退出指令，并且在重播程序期间，CPU 在许多指令已退出后触发中断——但这种方法需要修改才能在实践中工作。

我们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）。这个属性 （通常被描述为“确定性”[40]）在实践中并不适用于大多数 CPU 性能计数器 [17, 40]。例如 ，它不适用于任何已知 x86 CPU 模型上的任何“指令退休”计数器（例如，因为触发页面错误的指令被重新启动并计数两次）。

幸运的是，现代 Intel CPU 有一个确定的性能计数器：“退休条件分支” （“RCB”），所以我们使用它。我们不能仅在录制期间计算 RCB 的数量，并在重放期间执行该数量的 RCB 后传递信号，因为 RCB 计数并不能唯一地确定传递信号的执行点。因此，我们将 RCB 计数与通用寄存器（包括程序计数器）的完整状态配对以识别执行点。

通常，它仍然不能唯一标识执行点（例如，考虑无限循环标签：inc [global var]； jmp la bel；）。但是，在实践中，我们发现它可以可靠地工作。没有中间条件分支返回到同一条指令的代码必须非常少见，并且只有在这样的指令上发生异步事件时才对 RR 很重要——在这种情况下， 重放可能会发散并失败。

另一个主要问题是，尽管 CPU 可以编程为在观察到指定数量的性能事件后触发中断，但中断不会立即触发。在实践中，我们经常观察到它在数十条指令退出后触发。为了弥补这一点，在重放期间，我们将中断编程为比我们预期的实际 R CB 计数更早触发一些事件。然后我们在程序计数器值处为我们试图达到的状态设置一个临时断点，并重复运行到断点，直到 RCB 计数和通用寄存器值匹配它们记录的值。

3.2.4 共享内存

通过一次只调度一个线程，只要共享内存仅由跟踪线程写入，RR 就避免了共享内存上的竞争问题。记录的进程可以与其他进程甚至内核设备驱动程序共享内存，其中未记录的代码可以执行与跟踪线程访问竞争的写入。幸运的是，这对于在常见 Linux 桌面环境中运行的应用程序来说很少见，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，和 vdso 系统调用。我们通过自动禁用 PulseAudio 和 X 共享内存的使用（在两种情况下都回退到套接字传输）和禁用从应用程序直接访问 GPU 来避免前三个问题 。

vdso 系统调用是一种 Linux 优化，它完全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。我们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。

应用程序仍然可以以有问题的方式与未记录的进程共享内存，尽管这在实践中很少见，并且通常可以通过扩大 RR 记录的进程组的范围来解决。

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ptrace 监视并拦截用户系统容器中程序执行的所有系统调用。ptrace 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）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。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等调试工具中，但是ptrace 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）。

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 进程内系统调用拦截

上一节中描述的方法有效，但开销高得令人失望（参见图 5 低）。核心问题是，对于每个 tracee 系统调用，如图 1 所示，tracee 执行四个上下文切换：两个阻塞 ptrace 通知，每个都需要从 tracee 到 RR 和返回的上下文切换。对于常见的系统调用，例如 gettimeofday 或从缓存文件中读取，即使是单个上下文切换的成本也会使系统调用本身的成本相形见绌。为了显着减少开销，我们必须在处理这些常见的系统调用时避免上下文切换到 RR。

因此，我们将一个库注入到记录的进程中，该库拦截常见的系统调用，执行系统调用而不触发 ptrace 陷阱，并将结果记录到与 RR 共享的专用缓冲区。RR 定期将缓冲区刷新到其跟踪。这个概念很简单，但有一些问题需要克服。

3.2.1 拦截系统调用

在进程中拦截系统调用的常用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，我们发现这种方法是不够的，因为应用程序直接进行系统调用，并且由于 C 库的变化和需要自己预加载的应用程序 [37, 3]） 而脆弱。

相反，当被跟踪者进行系统调用时，RR 通过 ptrace 陷阱得到通知，并尝试重写系统调用指令以调用我们的拦截库。这很棘手，因为在 x 86 上，系统调用指令有两个字节长，但我们需要用一个 5 字节的调用指令来替换它。在实践中，经常执行的系统调用指令后面跟着一些已知的、固定的指令序列。例如，许多系统调用指令后跟一个 cmpl $0xfffff001,%eax 指令测试系统调用结果。我们在拦截中添加了五个手写存根库在返回修补代码之前执行系统调用后指令。 收到 ptrace 系统调用通知后，RR 将系统调用指令及其后续指令替换为对相应存根的调用。

我们（尝试）将所有系统调用指令重定向到拦截库，但为简单起见，它只包含最常见系统调用的包装器，对于其他系统调用，它回退到执行常规的 ptrace 捕获系统调用。

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，但是我们的拦截库需要避免针对选定系统调用的陷阱。幸运的是，现代 Linux 内核支持选择性地生成 ptrace 陷阱：seccomp-bpf。seccom p-bpf 主要用于沙盒。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；然后，对于目标进程执行的每个系统调用，内核运行过滤器，传入传入的用户空间寄存器值，包括程序计数器。过滤器的结果指示内核要么允许系统调用，要么以给定的 errno 失败，要么终止目标进程，要么触发 ptrace 陷阱。过滤器执行的开销可以忽略不计，因为过滤器直接在内核中运行，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。 RR 将一个特殊的内存页注入到每个 tracee 进程在固定地址（紧接在 execve 之后）。该页面包含一个系统调用指令——“未跟踪指令”。R R 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——除非程序计数器位于未跟踪指令处，在这种情况下允许调用。每当拦截库需要进行未跟踪的系统调用时，它都会使用该指令。

3.2.3 检查系统调用阻塞

一些常见的系统调用有时会阻塞（例如在空管道上读取）。因为 RR 一次运行一个跟踪线程，如果一个线程在没有通知 RR 的情况下进入阻塞系统调用，它将挂起并可能导致整个记录死锁（例如，如果另一个跟踪线程即将写入管道）。每当未跟踪的系统调用阻塞时，我们需要内核通知 RR 并挂起跟踪线程，以确保我们可以调度不同的跟踪线程。

我们使用 Linux perf 事件系统来监控 PE RF COUNT SW CONTEXT SWITCHES。内核每次从 CPU 内核调度线程时都会引发这些事件之一。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 。这些信号触发对 RR 的 ptrace 通知，同时阻止线程进一步执行。为了避免虚假信号（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虚假 S WITCHES。我们通过仔细检查跟踪状态来处理这些边缘情况。

图 3 说明了使用系统调用拦截记录阻塞读取系统调用。内核调度线程，触发性能事件，该事件向线程发送信号，重新调度它，中断系统调用，并向记录器发送 ptrace 通知。记录器记下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进展到系统调用退出并生成 ptrace 通知。在这个例子中 T2 有完成了一个（未拦截的）阻塞 futex 系统调用，所以我们继续执行 T2。 恢复被信号中断的拦截系统调用（例如图 3 中的 T 的读取调用）更加复杂。解释这需要了解 Linux 系统调用重启语义，这太复杂了，无法在此处的可用空间中解释。

3.3 基于用户空间的容器设计

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。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。

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。

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。

3.3.1 进程标识符

由于我们的进程命名空间，我们容器内的进程接收到独立于容器外部世界的唯一 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是确定性的，并且 Linux 在每个命名空间中顺序分配 PID，容器内的 PID 自然是确定性的。

3.3.2 文件和目录

由于复杂的 API 和广泛的元数据，文件和目录是不可再现性的丰富来源。我们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 系统调用完成。DetTrace 也可以嵌套在 Dock er 等标准容器中，以提供与主机更强的文件系统隔离。文件和目录的所有权和权限是 DetTrace 计算的输入（图 1）。Linux命名空间控制着命名空间内部的uid/gid到宿主机上的uid/gid的映射； 此映射也是 DetTrace 输入的一部分。默认情况下，我们将当前用户帐户映射到容器内的 root，并将所有其他用户帐户映射到nobody/nogrou p。返回目录条目的顺序由文件系统实现控制。为了使 getdents 系统调用可重现，DetTrace 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 系统调用具有不可重现的语义，因为它们可能读取/写入的字节数可能比请求的字节数少。虽然在实践中我们从未见过对常规文件进行这样的“部分”操作，但它们确实在访问管道时经常出现。为了使这些系统调用在所有情况下都可重现，DetTrace 会自动重试部分读取和写入，直到它们处理请求的字节数，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，告诉当前读取在前一次读取结束的地方继续。 索引节点是文件系统挂载中文件或目录的唯一标识符。stat 系列系统调用向用户进程报告 ino de，并且仅仅报告一个固定值是不够的，因为许多用户进程比较 inode 值以快速识别相同的文件 。相反，DetTrace 维护一个来自真实（不可重现） 的映射 inode 到可重现的虚拟 inode。需要特别注意确定何时创建新文件，因为操作系统可能会为 回收真正的 inode，但 DetTrace 必须分配新的虚拟 inode 以保持可重复性 （参见文件时间戳讨论，下一个）。 文件时间戳为用户进程提供了一个时间概念， 未经过滤，可用于重建不可重现的时钟。因此， DetTrace 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 DetTrace 中，我们总是将 ati me 和 ctime 报告为 0。但是，我们发现在许多程序中总是返回一个固定的 mtime 值会违反健全性检查。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。 DetTrace 实现了真实 inode 和虚拟 mtime 之间的映射，允许来自报告 mtime 的系统调用 （如 stat）的可重现但合理的响应。每当用户进程打开文件时，在 open 调用到达内核之前，我们会检查指定路径中是否存在文件。在 ope n 调用返回容器中的进程之前，我们通过检查 /proc 文件系统来识别底层的真实 inode，以获取新创建的文件描述符的路径和真实 inod e。通过在打开调用到达操作系统之前和之后检查路径，我们可以可靠地识别何时创建新文件。 如果文件是新创建的，我们将其 mtime 分配为当前虚拟 mtime，并递增当前虚拟 mtime。否则，该文件存在于初始容器映像中，我们为其分配虚拟 mtime 0。写入文件当前不会更新其虚拟 mtime，因为我们在工作负载中没有发现这有必要，但是可以很容易地添加它以提供更逼真的虚拟时代。

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 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基于二进制翻译的可移植虚拟化方案

4.1.1 基于命名空间的容器架构

4.1.2 基于QEMU的模拟器

4.1.3 动态二进制指令翻译

4.2 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。

4.2.1 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。

4.2.2 RISC-V目标系统构建

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（）、gettimeofday（）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

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

Celio C, Chiu PF, Asanovi 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.

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.

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.

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.

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.

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.

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.

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.

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.

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

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.

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.

A. Waterman and Y. Lee. Spike - RISC-V ISA Simulator[OL]. https://github.com/riscv/ riscv-isa-sim.

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

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.

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.

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.

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.

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.

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.

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).

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.

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.

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.

报告指标说明

- 原文总字符数：即送检文献的总字符数，包含文字字符、标点符号、阿拉伯数字（不计入空格）
- 检测字符数：送检文献经过系统程序处理，排除已识别的参考文献等不作为相似性比对内容的部分后，剩余全部参与相似性检测匹配的文本字符数
- 总相似比：送检文献与其他文献的相似文本内容在原文中所占比例
- 参考文献相似比：送检文献与其标明引用的参考文献的相似文本内容在原文中所占比例
- 可能自引相似比：送检文献与其作者本人的其他已公开或发表文献的相似文本内容在原文中所占比例
- 单篇最大相似比：送检文献的相似文献中贡献相似比最高一篇的相似比值
- 是否引用：该相似文献是否被送检文献标注为其参考文献引用，作者本人的可能自引文献也应标注为参考文献后方能认定为“引用”

检测报告由万方数据文献相似性检测系统算法生成，仅对您所选择的检测范围内检验结果负责，结果仅供参考
万方检测官方网站：https://check.wanfangdata.com.cn/　检测报告真伪验证官方网站：https://truth.wanfangdata.com.cn/
北京万方数据股份有限公司出品