**中图分类号：TP311**

**论文编号：10006SY1306320**



硕士学位论文

**面向移动设备的cache攻击关键技术研究**

作者姓名 李勃

学科专业 计算机系统结构

指导教师 姜博副教授

培养院系 计算机学院

**TODO**

**A Dissertation Submitted for the Degree of Master**

**Candidate：Li Bo**

**Supervisor：****Prof. Long Xiang**

School of Computer Science and Engineering

Beihang University, Beijing, China

**中图分类号： TP311**

**论文编号：10006SY1306320**

硕 士 学 位 论 文

**面向移动设备的cache攻击关键技术研究**

作者姓名 李勃 申请学位级别 工学硕士

指导教师姓名 姜博 职 称 副教授

学科专业 计算机系统结构 研究方向 程序调试技术

学习时间自 2015 年 9 月 10 日 起至 2018年 3 月 日止

论文提交日期 2018 年 月 日 论文答辩日期 2018年 3 月 日

学位授予单位 北京航空航天大学 学位授予日期 年 月 日

关于学位论文的独创性声明

本人郑重声明：所呈交的论文是本人在指导教师指导下独立进行研究工作所取得的成果，论文中有关资料和数据是实事求是的。尽我所知，除文中已经加以标注和致谢外，本论文不包含其他人已经发表或撰写的研究成果，也不包含本人或他人为获得北京航空航天大学或其它教育机构的学位或学历证书而使用过的材料。与我一同工作的同志对研究所做的任何贡献均已在论文中作出了明确的说明。

若有不实之处，本人愿意承担相关法律责任。

学位论文作者签名：        日期： 年 月 日

学位论文使用授权书

本人完全同意北京航空航天大学有权使用本学位论文（包括但不限于其印刷版和电子版），使用方式包括但不限于：保留学位论文，按规定向国家有关部门（机构）送交学位论文，以学术交流为目的赠送和交换学位论文，允许学位论文被查阅、借阅和复印，将学位论文的全部或部分内容编入有关数据库进行检索，采用影印、缩印或其他复制手段保存学位论文。

保密学位论文在解密后的使用授权同上。

学位论文作者签名： 日期： 年 月 日

指导教师签名： 日期： 年 月 日

摘 要

多核技术已经成为当今处理器技术发展的主流方向，编写基于多核环境的并行程序也随之成为发掘处理器并行性的重要技术策略。然而多核系统中并行程序执行过程的不确定性给程序调试、容错处理、入侵检测等应用带来了很大的困难。执行重放技术是解决并行程序执行不确定性的有效手段。执行重放技术是指程序首次或某次执行时记录程序的执行信息，然后在程序的二次执行中，利用这些信息指导、控制程序执行，确保程序的重放执行与其在记录阶段执行相同。目前，国内外已经有一些相关研究，然而，这些研究主要面对的是常用操作系统下应用程序的执行重放，对于多核实时操作系统下应用程序重放的研究还很少，还不存在直接可应用于多核实时平台下应用程序重放的方案，这成为多核实时系统应用开发中急需解决的问题。因此，本文致力于研究多核实时系统下应用程序的执行重放技术，然而，由于实时系统对实时性的严格要求，而采用软件方式实现对多核之间共享内存访问的记录开销太大，无法满足实时系统的要求，因此本文对于多核之间共享内存的不确定性不做深入研究。

本文以国产多核实时操作系统ACoreOs作为研究对象，设计实现了一种基于现有ACoreOs开发环境的ACoreOs应用执行重放方法。首先，阐述了执行重放的基本概念、背景以及重要性，总结了近年来对执行重放的研究成果，从纯软件方式和硬件支持方式对已有的执行重放技术进行了分析和对比，并在此基础上分析了多核实时系统上应用程序执行重放的特殊性。其次，分析了目标多核系统自身设计实现的特点，在此基础上给出了执行重放所必须解决的数个问题，并针对这些问题提出了相应的解决方案。接着，给出了基于宿主机—目标机开发环境的多核实时系统下应用程序的执行重放方案，该方案通过应用在目标机执行阶段收集程序执行信息并上传到宿主机，在宿主机上进行日志的分析过滤，并根据分析结果控制目标端应用二次执行来实现对目标应用的执行重放。方案中包括三个功能模块：信息记录、日志解析过滤和重放模块。随后，使用程序插桩、交叉调试以及自旋锁相关技术实现了该执行重放方案，最后使用该重放方案成功对数个目标应用进行了重放，验证了本方案对于几类目标程序的有效性，同时也量化了记录执行阶段所引入的插桩探针对系统的整体影响，测试的结果表明插桩探针对于系统性能造成了一定的影响。

**关键词：** 程序插桩，执行重放，多核实时，并行程序，程序调试

**Abstract**

As multi-core becomes the mainstream of processor technology development, writing parallel programs based on multi-core systems becomes an important means to explore the parallel performance of multi-core processors. However, as the execution of parallel in multi-core systems is non-deterministic, which makes program debugging, fault tolerance, intrusion detection and other applications quite difficult. Execution replay technology is the solution to the non-deterministic of parallel program execution. Execution replay is a general term for a set of methods to record program behaviors during execution and to use these recordings to reproduce the same behaviors during debugging session. Currently, there are already some researches at home and abroad, these studies are mainly focused on performing reproduction under common operating system applications. However, researches about the application replay for multicore real-time systems are extremely few and there does not exists the approach that can directly be used at multicore real-time operating systems. It is quite necessary to solve this problem for application development in multicore real-time operating system. Therefore, this dissertation is aimed at the execution replay of programs in multicore real-time operating systems. In addition, due to the stringent requirements of real-time in real-time systems, it may not meet the requirements of real-time systems to achieve recording the shared memory access between multicore in software way, so the dissertation does not do in-depth research in shared memory access between multicore.

In this dissertation, we use the domestic multicore real-time operating system named ACoreOs as research subject, and design and implement an execution replay method based on existing development environment. Firstly, we describes the basic concepts, background and importance of execution replay of application programs, summarizes recent research on the execution replay, comparatively analyses the existing execution replay technology implemented in two diffident ways: pure software implemented and with hardware support, and analyses the particularity of execution replay in multicore real-time operating system. Secondly, after analysing the characters of target system in design and implementation, we give the number of issues which must be resolved to perform replay and propose solutions corresponding to these issues. Then, we propose a host/target development environment based execution replay method of multicore real-time operation applications. This method achieves the execution replay of target applications by the replay module, which used the logs that has been analyzed to control the application re-execution. The logs are collected and uploaded to the host when the applications running on the target machine in its recording stage. This method includes three modules: information recording module, log parsing and filtering module and replay module. Thridly, we implements this method by using instrumentation technique, cross debugging technique and spin-lock technique. Finally, we successfully reproduces several target programs by our method, which verifies the feasibility of the proposed method for these kinds of programs. After that, we quantify the overall impact of the stub probe introduced in phase of recording, and the result shows that the probe causes a certain influence on the performance of the system.

**Key words**: Program Instrumentation, Execution Replay, Multicore Real-Time, Concurrent Program, Program Debugging

**目 录**

第一章 绪论 1

1.1 研究背景及意义 1

1.2 执行重放技术研究现状 2

1.3 研究内容及目标 8

1.4 本文研究限制 10

1.5 本文的组织结构 10

第二章 相关技术以及原理 12

2.1 程序插桩 12

2.2 软件调试技术 12

2.3 ACoreOs操作系统 14

2.3.1 ACoreOs的任务状态转换 16

2.3.2 ACoreOs任务调度 18

2.3.3 任务间通信机制 20

2.3.4 ACoreOs中断处理 22

2.3.5 自旋锁技术 23

2.4 执行重放挑战及问题 23

2.4.1 探针效应 23

2.4.2 事件发生位置标记问题 24

2.4.3 上下文特殊性 25

2.4.4 事件发生顺序 25

2.4.5 系统特殊性 27

2.5 本文研究限制 28

2.6 本章小结 28

第三章 执行重放方案的设计 29

3.1 执行重放方法整体设计 29

3.2 信息记录模块的设计 32

3.2.1 确定程序执行的不确定性 32

3.2.2 记录程序执行不确定性探针设计 33

3.3 日志分析过滤模块设计 35

3.3.1 确定目标应用中的任务 35

3.3.2 确定必需的信息 35

3.3.3 单核日志过滤模块 36

3.3.4 多核日志过滤模块 37

3.4 重放模块设计 38

3.4.1 重放控制流 38

3.4.2 重放同步事件 39

3.4.3 重放异步事件 39

3.4.4 重放数据流 40

3.4.5 重放阶段调度问题 41

3.4.6 单核重放模块设计 41

3.4.7 多核重放模块设计 42

3.5 关键技术解决方案 43

3.5.1 探针效应的解决方案 43

3.5.2 事件发生位置标记的解决方案 44

3.5.3 上下文特殊性的解决方案 46

3.5.4 事件发生顺序的解决方案 47

3.5.5 系统特殊性解决方案 48

3.6 本章小结 48

第四章 执行重放方案的实现 49

4.1 信息记录模块实现 49

4.1.1 探针实现 51

4.1.2 探针插桩实例 55

4.1.3 环形缓冲区实现 57

4.1.4 写入控制模块实现 58

4.1.5 伪消息队列实现 60

4.1.6 缓冲区管理模块实现 61

4.1.7 日志上传模块实现 62

4.2 日志解析过滤模块实现 64

4.2.1 日志解析模块实现 64

4.2.2 日志过滤模块实现 67

4.3 重放模块 71

4.3.1 单核版本重放模块 71

4.3.2 CPU管理模块 72

4.3.3 任务集信息管理模块 75

4.3.4 多核版本执行重放模块 75

4.4 本章小结 76

第五章 系统验证 78

5.1 测试环境部署 78

5.2 实验目标 79

5.3 实验结果 80

5.3.1 同步事件重放 80

5.3.2 数据流重放 87

5.3.3 异步事件重放 91

5.3.4 多核事件顺序重放 94

5.3.5 插桩开销实验 99

5.4 本章小结 100

总结与展望 101

参考文献 104

附 录 109

攻读硕士学位期间得到的学术成果 111

致 谢 112

**图 目**

图 1 记录系统和重放系统在执行重放中扮演的角色 3

图 2 执行方案的分类 4

图 3 SMP-Revirt使用CREW协议实现对页面访问冲突顺序的记录 5

图 4 FDR硬件架构图 7

图 5 ACoreOs调试器工作原理 14

图 6 ACoreOs操作系统组成及各部分关系图 15

图 7 宿主机与目标机的连接关系图 16

图 8 ACoreOs任务状态转换图 17

图 9 不可抢占的优先级调度 18

图 10 可抢占的优先级调度 19

图 11 时间片轮转调度 20

图 12 导致系统出错的执行 24

图 13 相同的执行，加入了软件探针，"躲过"了故障 24

图 14 任务相应执行代码 26

图 15 并行访问信号量时任务A先获取到操作权 27

图 16 并行访问信号量时任务B先获取到操作权 27

图 17 执行重放方案整体设计图 29

图 18 执行重放执行流程 31

图 19 信息记录模块的分离式设计 34

图 20 单核版本日志过滤模块设计框架图 37

图 21 多核版本日志过滤模块设计框架图 38

图 22 单核版本重放模块设计框图 42

图 23 多核版本重放模块设计框图 43

图 24 互斥信号量对多核任务执行影响 44

图 25 循环体中的PC指针无法唯一标识程序执行的当前位置 45

图 26 触发任务切换因素 46

图 27 信息记录模块实现框图 50

图 28 信息记录模块开启执行流程 50

图 29 ACoreOs\_semaphore\_obtain接口探针插桩比对实例 56

图 30 中断信息插桩探针对比实例 57

图 31 rdtsc()函数插桩实现 57

图 32 环形缓冲区结构图 58

图 33 写入控制模块执行流程图 59

图 34 伪消息队列设计框图 61

图 35 缓冲区管理模块执行流程图 61

图 36 日志上传模块执行流程图 63

图 37 日志解析模块工作流程 65

图 38 日志解析模块实现框图 66

图 39 日志过滤模块实现 67

图 40 单核版本的日志过滤模块执行流程 68

图 41 多核版本的日志过滤模块执行流程 70

图 42 单核版本执行重放模块的执行流程 72

图 43 CPU管理模块工作流程 73

图 44 多核版本执行重放模块执行流程 76

图 45 测试环境架构图 79

图 46 同步事件测试程序运行结果 81

图 47 同步事件测试程序执行结果分析结果 82

图 48 同步事件测试程序执行时序图 83

图 49 日志中任务创建事件信息的提取和托管 84

图 50 重放模块根据日志创建任务上下文 84

图 51 重放模块根据同步事件信息设置断点并启动相应任务 85

图 52 同步事件设置的断点命中后上下文比对过程 85

图 53 到达切换点后任务切换过程 86

图 54 生产者消费者程序重放执行输出结果 87

图 55 数据流测试程序运行结果 88

图 56 数据流测试程序记录得到的日志分析结果 89

图 57 重放阶段创建重放任务上下文 90

图 58 任务1重放过程中断点设置 90

图 59 任务1重放中断点命中数据回写 90

图 60 数据流测试程序重放执行输出结果 91

图 61 中断测试程序执行阶段输出结果 92

图 62 中断测试程序记录得到的日志分析结果 92

图 63 中断事件重放过程 93

图 64 中断测试程序的重放输出结果 94

图 65 多核上生产者消费者测试程序执行结果 95

图 66 多核上生产者消费者任务日志分析结果 96

图 67 多核重放模块创建对应任务上下文 97

图 68 多核下事件重放过程 98

图 69 多核上生产者消费者被重放输出结果 98

图 70 插桩开销实验结果 99

**表 目**

表 1 现有执行重放方案对比 7

表 2 软件调试方法分类 13

表 3 任务创建事件信息记录探针 51

表 4 任务启动事件信息记录探针 51

表 5 信号量操作信息记录探针 52

表 6 消息队列事件信息记录探针 52

表 7 上下文切换事件信息记录探针 53

表 8 调度指示信息记录探针 53

表 9 中断打断任务上下文信息记录探针 54

表 10 数据流信息记录探针 54

表 11 CPU索引信息记录探针 54

表 12 缓冲区管理任务对各个消息处理过程 62

表 13 Java语言和目标系统数据结构对应关系 64

表 14 过滤多余事件信息算法伪代码 68

表 15 日志合并算法伪代码 69

表 16 过滤多余异步事件算法伪代码 70

表 17 上下文比对算法伪代码 74

表 18 实验测试程序概述 80

# 绪论

## 研究背景及意义

确定性是计算机系统中一个诱人的特性，它用于保证程序每次在系统上的执行表现行为是一样的[[1](#_ENREF_1)]。正是这个特性极大的加速了不同计算机产品的开发和部署，从而广为计算机设计人员和编程人员所喜爱。例如，循环调试就借助了这个特性，而循环调试是最通用的调试技术，它要求程序不论被执行多少次，每次的执行行为都是可再现的。确定性在芯片测试中也很关键，工程师通过比较芯片的输出数据和确定的黄金数值来发现有瑕疵的芯片。

不幸的是，在现代计算机系统中，确定性经常被无处不在的不确定性因素破坏。一个比较典型的因素是来自程序外部的输入可能是不确定的数值或者跟时间相关性的数值。而且，普遍应用于多核处理器平台上的进程间通信可能也是不确定性的。由于各种各样的不确定因素，导致相同的程序（即使在相同的计算机系统上）不同次的执行有不同的行为，这个现象并不奇怪。

执行重放是一种新兴技术，致力于在存在不确定性因素的情况下保证计算机程序确定性执行。它对于需要确定的程序执行的应用程序是至关重要的（例如，程序调试）。在诊断错误的来源，我们经常需要重新执行程序很多次，希望程序确定性表现出同样的错误的行为，这可以通过执行重放技术来强制程序执行，从而确定性表现出同样的错误的行为。其他潜在的应用包括在线程序分析，芯片调试，容错，性能预测和入侵分析吸引了来自学术界和工业界（例如IBM[[2](#_ENREF_2)], Intel[[3](#_ENREF_3)], Microsoft[[4](#_ENREF_4)]）广泛的兴趣。随着多处理器计算（其中，处理器间通信往往是不确定的）时代的来临，这些应用对有效的执行重放的需求变得更加迫切了。目前，已经有一些研究人员提出各种各样的执行重放方案解决普通桌面系统上应用程序的执行重放，然而几乎还没有直接可用于多核实时操作系统的执行重放方案。

ACoreOs作为中国自主研发的一款类VxWorks的多核实时操作系统，应用前景极广，其高可靠性和强实时性保证其广泛用于航空、军事等对可靠性和实时性要求苛刻的领域。ACoreOs作为一款提供丰富接口的操作系统，其上开发的应用程序也变得越来越复杂，编写不存在问题的应用程序对开发人员提出很大的挑战，为了方便开发人员在开发过程中对应用程序的调试和当应用程序部署到对应平台后运行出现故障对故障的定位，对该环境下的执行重放技术的研究迫在眉睫。

本文重点研究了多核实时操作系统（ACoreOs）单核下同步事件、异步事件、数据输入的不确定性以及多核调度的不确定性。由于多核共享内存记录开销过大，无法应用于多核实时操作系统，不作为本文研究对象。

## 执行重放技术研究现状

给定包含个线程的应用程序P，对于程序P的一次执行，P中的对一个变量的访问结果不仅仅取决于的初始状态，也取决于不确定数值的输入或者具有时间特性的输入，这里不确定数值的输入和具有时间特性的输入对于P都是不确定的因素。执行重放的思路是，在中执行中将所有的或者部分不确定因素作为日志记录下来，进而根据日志信息来重现中对变量的访问结果，有能力重现的执行，记做。通常，原始的执行体被称为记录执行；重现的执行体被称为重放执行。

使用执行重放的方案需要借助记录系统和重放系统。记录执行在记录系统中进行，记录系统借助专用的软件或者特殊硬件记录不确定的因素；很多情况下，程序员的开发系统充当了记录系统，也有些情况下（例如入侵分析），部署在终端用户中的实际系统充当了记录系统。在记录系统中记录得到的日志会反馈给重放系统用来控制程序的重放执行，重放系统既可以是和记录系统相同的也可以是和重放系统不同的专门系统（例如一个特定的计算机系统或者甚至是一个软件模拟器）。重放系统的选择多样性使得执行重放比传统的基于日志回滚的协议[[5](#_ENREF_5)]更具有灵活性，因为后者必须在记录系统中完成重放。

记录系统和重放系统在执行重放中扮演的角色，如图 1所示。

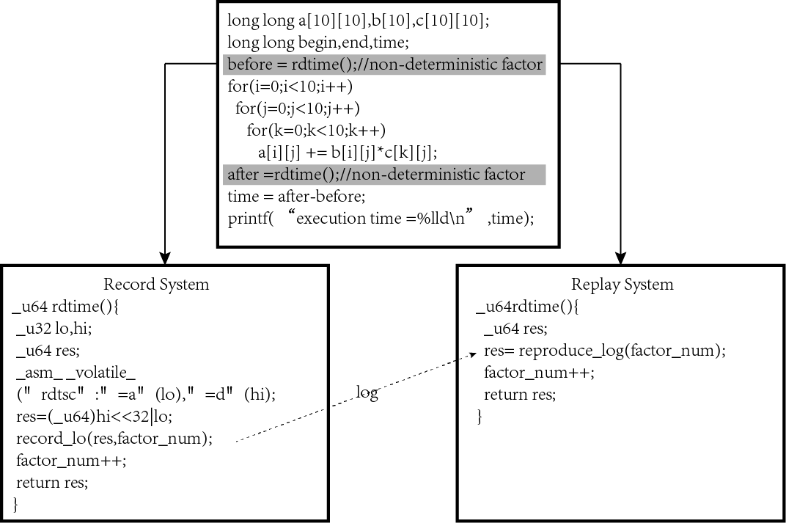


图 1 记录系统和重放系统在执行重放中扮演的角色

从图中可以看出，程序调用rdtime()函数两次用来计算循环的执行时间，很显然，由于不同的执行中调用函数rdtime()将得到不同的输入，从而函数rdtime()是一个不确定因素。在执行重放中为了解决函数rdtime()这种不确定性，记录系统和重放系统中函数rdtime()将有不同的实现体。在记录系统中，函数rdtime()将会调用函数record\_log()将得到的时间（在X86中通过rdtsc指令获得）记录到日志文件中；在重放系统中，函数rdtime()直接从日志文件中读取数值并作为结果而不需要真正的借助执行rdtsc来获取时间。通过这种方式，在重放系统中可以切实的重放记录执行任意多次，并且每次的执行结果都具有和记录执行相同的结果。

本文前面介绍了执行重放的基本概念以及多核平台下应用程序的不确定性，接下来本文将简要介绍几个使用了执行重放的典型应用。这些应用包括程序调试[[6](#_ENREF_6)]、在线程序分析[[7](#_ENREF_7)] [[8](#_ENREF_8)] [[9](#_ENREF_9)] [[10](#_ENREF_10)]、Postsilicon调试[[9](#_ENREF_9); [11-14](#_ENREF_11)]、容错、性能预测和入侵分析。这些应用对执行重放的需求不仅仅在于选择合适的执行重放方案，同时也促进了对执行重放技术进行更深入研究。

执行重放的研究分类方式多种多样，如图 2所示。根据所采用的计算机系统可分为单处理器执行重放方案和多处理器（多核）执行重放方案。在单处理器执行重放方案中，根据应用程序包含的线程数量分为单线程版本和多线程版本；对于单核多线程重放方案，由于实现时的抽象层次不同，可被分成三种情况：基于库接口[[15-17](#_ENREF_15)]级别、基于系统调用[[18](#_ENREF_18); [19](#_ENREF_19)]级别、基于虚拟机[[20](#_ENREF_20); [21](#_ENREF_21)]级别的执行重放方案。在多处理器执行重放版本中，根据程序之间通信方式不同，分为基于消息通信[[22](#_ENREF_22)]应用程序重放方案和基于共享内存[[23-25](#_ENREF_23)]的应用程序重放方案，在基于消息中，由于消息记录形式不同也进一步分为基于消息内容和基于消息方向的重放方案；在基于共享方式中，根据共享内存访问记录方式不同进一步划分为完全、部分[[26-28](#_ENREF_26)]、不需要记录共享内存操作的重放方案，并且每一种方式下，都可以采用纯软和借助专门硬件两种方式来实现。



图 2 执行方案的分类

近年来，国内外的研究人员已经对执行重放有了相当多的研究，提出了许多不同的执行重放技术，不同的方法有不同的策略和不同的权衡。理想的执行重放技术是在通用的硬件平台上，以最小的记录时间空间开销达到最低的开销和最高精度的重放。下面将从纯软件[[26](#_ENREF_26); [29-32](#_ENREF_29)]和有硬件支持[[3](#_ENREF_3); [25](#_ENREF_25); [33-35](#_ENREF_33)]的两种实现方式对现有的国内外的实现方法进行分析和比较。

SMP-Revirt[[31](#_ENREF_31)]和PinPlay[[29](#_ENREF_29)]是典型仅采用软件方式来实现多核下应用程序的执行重放。SMP-Revirt利用已有的硬件页面保护功能来检测多核处理器上虚拟机虚拟的多个处理之间内存页面粒度的竞争。它管理了每个页面的读写权限，并协助采用CREW（并行度、互斥写）协议。从而，如果各个处理器访问相同页面，并且至少其中一个需要修改这个页面内容，那么系统会产生页面错误异常，在处理页面错误异常时，多个处理器对页面的访问顺序就可以被记录下来。

图 3给出了CREW协议如何用来帮助记录页面竞争访问顺序，首先，页面a被设置为并行读的状态，当一个线程T0通过接口write(a)操作写这个页面时，由于T0没有获取到异步写权限而引发页面异常，接着，产生CREW事件i0，该事件剥夺T1的并行读权限而授予T0异步写的权限，在事件i0中，SMP-Revirt在日志中记录i0—> write(a)，在这之后，当线程T1尝试通过read(a)读取相同的页面，因为其没有访问这个页面的权限，也会触发页面异常，同样，产生CREW事件i1，该事件剥夺T0的异步写权限，同时授给T1和T0并行读权限，在事件i1中，SMP-Revirt记录i1—> read(a)到日志文件中。SMP-Revirt可以检测页面粒度级别的内存竞争操作，但也引入了大量的页面粒度的错误共享，从而对记录执行的速度影响还是很大的（甚至可以减慢10倍），同时，当多个处理器需要写相同的页面时，SMP-Revirt采用的CREW协议可能导致频繁的页面异常，这也限制了SMP-Revirt在多个处理器上的扩展性。



图 3 SMP-Revirt使用CREW协议实现对页面访问冲突顺序的记录

PinPlay[[29](#_ENREF_29)]基于动态插桩工具Pin[[36](#_ENREF_36)]跟踪每一条单独的内存操作，取代了基于页面粒度对共享内存访问竞争进行排序。PinPlay实现了软件版本的FDR[[33](#_ENREF_33)]模拟Cache一致性协议获取共享内存操作中所有的读写、写写、写读竞争，并且竞争检测的粒度是可调节的。由于在记录阶段每次共享内存操作都必须被插桩，PinPlay明显造成记录执行速度缓慢。例如，PinPlay在记录4线程的OpenMP基准[[29](#_ENREF_29)]程序中导致程序执行速度下降117倍。

Sundmark提出用于单核处理器实时系统重放的技术[[37](#_ENREF_37)]，该方案利用工业上标准的调试器帮助循环调试实时系统，也提出了多个新的技术：一个新的标示方法[[38](#_ENREF_38)]，用来标示在循环执行块中出现的中断或者任务抢占事件的准确发生位置（标示哪次循环执行中发生了中断或者任务抢占），该标示方法使用PC数值以及堆栈状态唯一标示程序执行过程中的某个位置；一个用来找出预先定义的重放过程开始点的算法以及如何使用标准调试器的条件断点来重放某个实时应用的技术。该重放系统在执行阶段记录典型的系统事件，例如多任务之间的切换以及外部的输入，异步事件发生的时刻以及类型。重放阶段，根据记录阶段得到的数据以离线的方式进行程序的执行重放。

该方案在一个工业机器人控制系统上得到实现[[39](#_ENREF_39)]，该系统包含多个计算控制系统、信号处理系统和I/O单元。该方案主要运用在该系统的控制子系统中，该控制系统是运行在VxWorks实时系统之上，包含大约250万行代码。但该方案是针对具体应用来进行系统代码的修改和记录，针对其他应用，对系统代码的修改也可能会不同，记录的信息也有所不同，重放阶段，需要采取的措施也可能不同，导致该方案没有通用性。

FDR是第一个采用硬件辅助完成记录的执行重放方案[[33](#_ENREF_33)]，用来重放顺序一致的系统。FDR基于cache一致性消息的目录来捕获辅助信息用来记录对共享内存操作顺序。FDR为每个处理器核心提供用来计算最后获取的指令位置的指令计数器（IC）、用来记录编号为b的cache块被编号为i的处理器核访问的cache指令计数器（i: CIC[b]），该计数器来表示在核i上访问编号为b的最后一条指令计数值。对于核i，一旦一条指令被提交，IC数值将被更新，如果被提交的指令同时又是一个访问了cache块b的内存操作，那么i:CIC[b]也做相应的更新。一旦核i发送关于cache块b的一致性回应（数据或者失效响应）给另外一个核（记做j），核i也需要附带发送它的核标识和对应的i:CIC[b]给核j，当核j收到上述描述的来自核i的回复，它将记下接收来自核i的CIC和自身的当前CIC的依赖关系。

图 4给出了FDR需要的硬件支持，每个处理器核心都配置了IC（指令计数器）域，每个cache块都附带CIC项，cache一致性控制器增加了VIC（保存所有核心上的IC数值）域，除此之外，需要一个RLB（竞争日志缓冲区）来记录竞争信息。绝大部分的硬件开销来自为每个cache块配置的CIC域，它几乎占整个cache区域的6.25%，同时，在1GHz运行的核心上，FDR的日志记录速度不能超过40M/s。



图 4 FDR硬件架构图

上文对国内外现有的一些执行重放技术进行了介绍和分析，现有的重放技术各有侧重点，主要有两种分类方式，一种是根据是否借助特殊的硬件支持分为纯软实现方式和借助特殊硬件支持实现方式，另一种是根据重放的级别分为用户级的重放和系统级的重放。本文下面将对已有的重放方案特点进行对比，如表 1所示。

表 1 现有执行重放方案对比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 执行重放方案 | 特点 | | | |
| 重放级别 | 实现方式 | 支持多核 | 支持多核实时系统 |
| InstantReplay[[40](#_ENREF_40)] | 用户级 | 程序插桩 | 否 | 否 |
| PinPlay[[29](#_ENREF_29)] | 用户级 | 动态二进制插桩 | 是 | 否 |
| ODR[[26](#_ENREF_26)] | 用户级 | 插桩运行库 | 是 | 否 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 表 1 现有执行重放方案对比（续） | | | | |
| Multi-stage replay[[20](#_ENREF_20)] | 系统态 | 虚拟机 | 是 | 否 |
| Revirt[[30](#_ENREF_30)] | 系统级 | 虚拟机 | 否 | 否 |
| SMP-Revirt[[31](#_ENREF_31)] | 系统级 | 虚拟机 | 是 | 否 |
| FDR[[33](#_ENREF_33)] | 用户级 | 特定硬件 | 是 | 否 |
| Rerun[[41](#_ENREF_41)] | 用户态 | 特定硬件 | 是 | 否 |
| BugNet[[42](#_ENREF_42)] | 用户态 | 特定硬件 | 是 | 否 |

综上所述，现有的重放方案几乎涵盖了对常见普通系统应用的执行重放，然而，对于多核实时系统上应用程序的重放还没有太多相关的研究，虽然Sundmark等提出了一种面向实时操作系统（VxWorks）的重放方案，但是没有考虑重放方案的通用性，仅仅针对了某些VxWorks上的特殊应用，并且其方案仅能针对VxWorks单核版本应用程序有效，因此，需要研究一种基于多核实时操作系统上应用程序的执行重放方案。

## 研究内容及目标

当前，几乎所有的应用程序都借助了多线程、多任务技术，来最大化利用系统提供的性能。然而，多线程、多任务技术的引入对应用程序的正确性开发和调试都带来了较大的挑战。调试作为找出应用程序执行过程中导致出错的根本原因在程序开发阶段是非常重要的，普通的调试方法对于多线程、多任务程序已经不再适用，而执行重放技术作为可以解决多线程调试问题的一项技术被广泛研究。前文提到，虽然目前已经有非常多样的重放方案，但几乎大部分都是针应对常用操作系统及其应用的执行重放方案，不能够直接应用到多核实时操作中，主要原因是多核实时操作自身设计的特点所决定的，多核实时操作系统面向的是对可靠性、实时性、安全性有很高要求的领域，例如航空、航天等，同时，其开发环境一般是基于宿主机—目标机架构，目标机体系也不同于普通系统所采用的X86体系。因此，需要研究一种可应用于多核实时系统应用程序的执行重放相关技术。本文主要的研究目标是基于多核实时操作系统（以ACoreOs操作系统为例）开发环境研究多核实时系统应用程序的执行重放方法。

围绕本文的研究目标，本文的研究内容包括设计并实现基于ACoreOs多核实时操作系统下应用程序的执行重放方法，主要有以下四个方面构成：

1. ACoreOs下应用程序执行重放方案总体设计

本文研究对象是多核实时系统ACoreOs，其提供的应用开发环境是基于宿主机—目标机架构，应用程序运行需要在目标机端，而开发环境部署在宿主机端。针对这种异构的目标环境，结合执行重放的基本技术，本文提出了基于宿主机—目标机架构的多核实时操作系统应用程序的执行重放方案，该方案主要分为三大模块：运行在目标机端的信息记录模块、运行在宿主机端的日志分析过滤模块和重放模块。

1. 基于插桩的应用程序执行信息记录

在研究目标系统ACoreOs的内部实现基础上，确定了其上执行的应用程序在执行阶段必须记录的信息，并结合ACoreOs操作系统自身特点提出了基于源代码插桩的信息收集探针，探针被插入在任务执行的某些路径上，在被执行时用来记录任务在系统上运行的详细过程。由于应用程序运行在目标端，其执行过程探针收集的信息也存在于目标端，然而日志分析和重放模块都位于宿主机端，必须保证任务执行时的信息被收集到宿主机端，信息记录模块先暂存任务执行中收集的执行信息并在合适时机上传到宿主机端，并且信息记录模块不能对系统的实时性造成较大影响。

1. 应用程序执行信息的分析和处理

信息记录模块在记录任务执行信息过程中，并不知道具体应该收集哪些应用程序的信息，因而，上传的日志信息中存在着其他应用程序运行时产生的日志信息，并且，信息记录模块收集了所有的事件信息，然而这些事件信息仅用来指导日志过滤模块分析出任务的执行流程，对重放不一定有用。在日志信息的分析和处理阶段，会过滤掉其他任务产生的信息，并分析出应用程序中各个任务的切换流程和对应触发切换的事件，以指导重放模块进行应用程序的重放执行。

1. 基于交叉调试器接口的应用程序重放方法

借助开发环境提供的交叉调试接口，本文在宿主机端设计实现了用于控制目标端应用程序执行的重放模块，并根据日志信息分析和处理的结果控制应用程序的二次执行，重放程序的控制流和数据流，以实现目标应用程序的执行重放。

## 本文研究限制

在多核处理器系统中，内存竞争存在于同步操作（同步竞争）或数据访问（数据竞争）中，数据竞争（多个任务之间通过共享内存来进行数据的交互）主要原因是多核上多个任务可以并行运行，任务访问应用中某个内存单元的数据其顺序是不确定性的。然而，由于任务中对内存的访问频率较高，如果采用软件CREW(Concurrent Read Exclusive Write)协议对内存交叉访问顺序进行记录，由于每次任务每次访问某个内存单元，都会触发该协议通过异常来进行相关信息的采集记录，会引入相当大的开销，而这对于实时系统是不可接受的，从而采用软件方式来对内存交叉访问进行记录对于实时系统是不可取的。目前，比较好的解决方案是借助特殊硬件的支持来对内存总线进行一致性监听，动态收集内存访问的顺序和相关性，但是本文的测试环境没有提供所需的硬件支持，故而本文不对共享内存交叉访问这个不确定性重放方案进行深入研究。

同时，由于ACoreOs操作系统中没有提供设置CPU亲和性的接口，而ACoreOs在创建任务时是可以不绑定到固定CPU上，允许任务在运行中动态的从一个核迁移到另一个核，从而本文借助调试服务接口的重放方案无法支持任务在不同的核心之间迁移。解决方案可以通过修改系统某些实现，提供动态设置CPU亲和性的接口，可允许任务动态修改自身CPU亲和性，从而强制任务在下一次切换时，从一个核迁移到另外一个核上运行。不过由于ACoreOs本身特殊性，本文无法对系统源代码进行某些变更使其满足核间迁移的需要，故本文重放模块不支持对发生核间漂移的任务的精确重放，然而本文所设计的日志分析过滤模块是支持对核间迁移任务的日志分析和过滤操作。

## 本文的组织结构

本文共分为五章，具体内容如下：

第一章 绪论。首先介绍了课题的研究背景和意义，接着介绍了执行重放技术的基本概念，原理和应用，并对此分析了近些年国内外对执行重放技术的研究工作，最后给出了本文的研究目标及内容。

第二章 相关技术以及原理。首先分析了插桩技术、软件调试技术，特别应用于本文中的交叉调试技术，接着详细介绍了本文的目标操作系统ACoreOs的内部实现机制，最后给出了执行重放过程中所面对的挑战和问题。

第三章 执行重放方案的设计。首先指出了重放本文目标系统上应用程序存在的问题，然后根据存在的问题具体给出了相应的解决方案，接下来，给出了本文提出的执行重放整体设计框图和各个模块的大致设计思路。最后，本文针对相关挑战给出了相应的解决方案。

第四章 执行重放方案的实现。本章分三部分，分别给出了执行重放方案中三大模块：信息记录模块、日志解析过滤模块、重放模块的实现细节。

第五章 系统验证。详细介绍了对本文提出的执行重放方案的测试和验证。并对引入的插桩开销进行了度量。

总结与展望。首先对本文的工作进行了总结，然后指出来本文工作不足，最后给出了对后续工作的展望。

# 相关技术以及原理

## 程序插桩

程序插桩[[43](#_ENREF_43)]技术最早是由J.C. Huang教授提出的, 它是在保证被测程序原有逻辑不变的基础上在程序中插入一些探针（如在代码中插入printf语句），通过探针的执行获取程序执行到此处的相关信息，通过对得到的信息分析，进而获取程序的控制流和数据流信息。根据探针插入的时间不同分为二进制可执行文件插桩和应用程序源代码插桩。

二进制可执行文件插桩主要是在应用程序编译成可执行文件后，借助某些二进制文件分析工具在可执行文件某些位置插入插桩代码。其优点是不需要应用程序的源代码，插桩工作通过工具自动化完成，插桩速度快，缺点也很明显，插桩点可能过多，并不是每一个符合的位置都需要插入插桩代码，插桩操作盲目，并且需要借助专门的二进制分析工具和插桩工具。

源代码插桩主要是基于源代码的插桩过程，它需要插桩人员熟悉整个程序的执行流程，选择对应的插桩位置，优点是插桩点相对少，对程序执行影响相对较小。

本文提出的方案中，需要对应用程序和操作系统的源代码进行插桩，这就要求非常熟悉操作系统各个模块的设计，能够对操作系统做出正确而尽量小范围的修改，满足重放的目的。对于应用程序的修改，需要满足在不变更其执行逻辑前提下，进行必要的修改，满足重放的目的。

## 软件调试技术

软件调试技术[[44](#_ENREF_44)]是目前应用最广的用于发现软件中错误的技术，它主要通过在程序的多次执行中通过一定手段暂停程序的执行，查看和修改被调试程序当前状态。在一个具体的程序中，指令和数据从来就是一个不可分割的整体。控制程序的执行过程和查看程序执行状态是任何调试工具和调试手段都必须提供的两个功能。

软件调试方法有多种分类方法，如表 2所示。以调试所处的阶段作为分类标准，可分为静态调试和动态调试；按照调试器所处理的指令粒度来分，可分为机器代码级调试和程序源代码级调试；如果按照调试对象运行空间来分，可以分为用户级调试和系统级调试；如果以被调程序是否与调试器位于同一运行环境来分，可以分为本地调试和远程调试等，具体的分类方法多种多样。

表 2 软件调试方法分类

|  |  |  |
| --- | --- | --- |
| 分类方法 | 类别1 | 类别2 |
| 调试所处阶段 | 静态调试 | 动态调试 |
| 处理的指令粒度 | 机器级调试 | 源码级调试 |
| 调试对象所在空间 | 用户级调试 | 内核级调试 |
| 运行环境是否异构 | 本地调试 | 远程调试 |

目前几乎所有的调试器都具备两个基本功能：断点设置和单步运行。为了能够在程序执行过程中动态查看程序的执行状态，程序员需要在程序执行的某些位置临时暂停程序执行。断点为程序员提供了这个能力。在某种意义上，断点可以说是软件调试技术的基础，通过设置断点，调试器才能够控制被调试的程序的整个运行过程。断点可以分为软件断点和硬件断点，也可分为指令断点和数据断点。软件断点是最常用的技术手段，它通常是一种特殊类型的机器指令，比如在X86上是INT 3[[45](#_ENREF_45)]，而在ARM平台上可以是一条非法指令。断点指令被调试器插入到程序的代码中，当处理器执行断点指令时，会触发断点处理，此时调试器就有机会停止被调试程序的执行，将控制权交给用户。硬件断点常常被用作数据监视点，其主要需要借助处理器提供的特殊部件来完成断点命中等相关操作。一般的，指令断点采用软件断点，而硬件断点用作数据监测点。

交叉调试技术[[44](#_ENREF_44); [46](#_ENREF_46); [47](#_ENREF_47)]作为一种调试技术，主要用在嵌入式软件开发环境中，它为用户提供在宿主机端调试运行在远端目标机的应用程序的能力。

调试器自身运行在宿主机端，被调试的应用程序运行在远端目标机上，为了实现调试器对远端应用的调试，宿主机需要通过某些协议与目标机进行通信，从而实现对目标端应用程序的控制。目前广泛采用的交叉调试技术主要分为基于专用硬件和软件方式实现两种。

软件方式实现主要是在目标机端运行特定的监测程序，其接收来自宿主机发送的命令，实现对被调试程序的控制，并将执行的结果反馈给宿主机端。基于软件实现的交叉调试技术，其优点是不需要特殊的硬件支持，扩展性移植性较好。但调试能力也是其一大限制，其无法良好的对某些系统代码提供调试支持。

基于专用硬件[[48](#_ENREF_48)]实现是需要处理器在设计时嵌入额外的硬件用于在处理器调试模式下动态获取处理器内部状态信息同时暂停被调试任务的运行，并通过专用的接口（JTAG）将相关信息发送给调试器，已达到调试的目的。通过该种方式实现的交叉调试器优点是不需要占用系统额外的通信带宽，采用专门的通信接口，调试能力较强，缺点是CPU必须提供相应的支持，通用性扩展性较差。

GDB[[49](#_ENREF_49)]作为GUN工具链中最著名的开源调试器，其稳定的性能、高扩展性、强移植性，在嵌入式系统开发中被广泛使用。GDB在嵌入式调试中由两部分构成：运行在目标机上远程调试代理服务和运行在宿主机上调试模块。运行在目标机端的调试代理简称GDBProxy，负责监听来自宿主机的调试命令，完成具体的调试动作，运行在宿主机上的调试模块简称GDBServer，其为用户屏蔽了远端的GDBProxy，使用户感觉调试远程的程序就如同调试运行在本地机器上的程序。

本文中的环境调试体系如图 5所示，与GDB整体设计是一样的，分为宿主机和目标机，目标机上运行核心调试代理服务、ACoreOs操作系统以及目标应用，核心代理通过网络接收调试服务客户端发送的调试命令，执行完成对应命令后将结果回传给调试客户端，宿主机上运行集成开发环境以及核心代理的远程客户端服务，该服务屏蔽了远端核心代理的实现细节，对开发环境呈现于调试本地应用完全系统的操作和接口。



图 5 ACoreOs调试器工作原理

## ACoreOs操作系统

ACoreOs 机载嵌入式实时操作系统[[51](#_ENREF_51); [52](#_ENREF_52)]（以下简称 ACoreOs）是专门为嵌入式多核实时系统设计开发的操作系统，为应用程序提供高效的实时多任务调度能力、快速中断/异常事件处理以及实时的任务间通信。

ACoreOs 操作系统作为一个针对航空应用需求设计的机载嵌入式多核实时操作系统，能够完全满足机载环境所提出的强实时、高安全、高可靠、高确定、可裁剪性、可升级性等特殊要求。借助于与之配套的集成开发环境 LambdaAE，用户可以快速完成嵌入式应用的开发、调试和部署。

ACoreOs 操作系统在系统中处于硬件层和应用层之间的位置，如图 6所示，其主要功能是高效管理机载计算机软、硬件资源，并以系统接口形式为应用程序提供所需的服务。

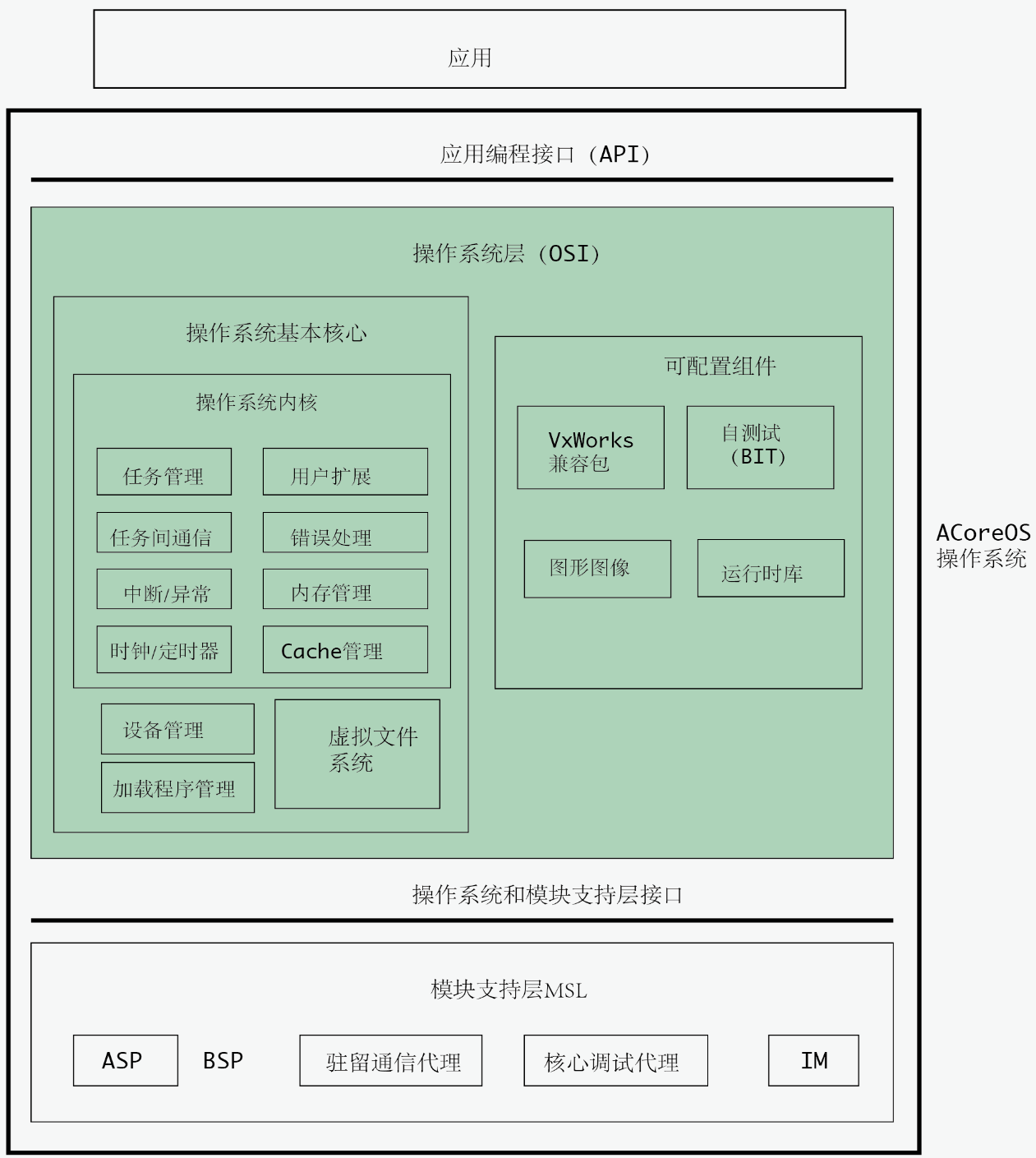


图 6 ACoreOs操作系统组成及各部分关系图

对于目标机硬件，ACoreOS 操作系统提供 MSL(Module Support Layer) 层软件，管理底层硬件层设备；对于应用软件，操作系统负责应用软件的管理，主要实现应用任务的调度、系统资源的分配，为应用软件提供可使用的组件，包括：VxWorks 兼容接口、BIT(Build In Test) 管理、C 运行时库、图形图像支持等功能，应用软件可通过操作系统提供的 API 接口访问操作系统，以支持应用软件的功能。

应用软件开发调试过程中，宿主机与目标机的连接关系如图 7所示。主机上运行的开发环境通过目标机服务器与目标机进行通信，完成应用软件的交叉调试。

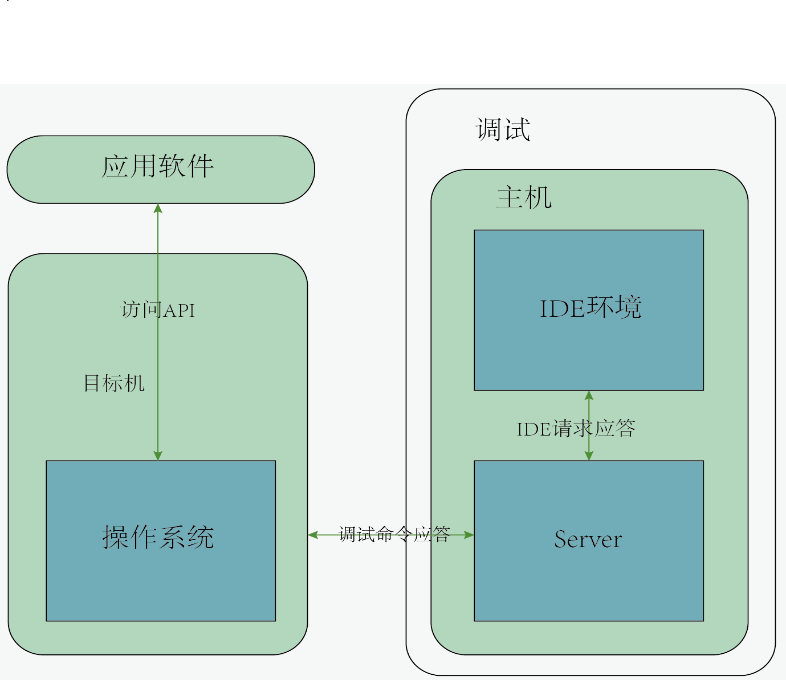


图 7 宿主机与目标机的连接关系图

ACoreOS 内核采用基于对象的方法管理任务、周期任务、用户扩展、消息队列、环缓冲、信号量、标志集和定时器。所有对象自创建起获得内核分配给它的唯一标识符，对象名字可以由用户指定，也可以由系统进行分配，但是内核不允许同类对象的名字相同，对象的名字是不能超过31个可打印字符的字符串。对象的标识符和同类对象名字全局唯一，并且同类对象标识符和名字一一对应。

### ACoreOs的任务状态转换

在 ACoreOS系统中的所有任务，其状态一定处于休眠态、就绪态、运行态、阻塞态四种状态之一。

1. 休眠态：任务已经创建或停止，并且无法接收CPU的调度；
2. 就绪态：任务已经取得除 CPU 以外的资源，可接收 CPU 的调度，一旦获得CPU控制权就进入运行态开始运行；
3. 运行态：就绪的任务成功争夺到 CPU 控制权并正在执行中；
4. 阻塞态：任务被挂起或由于所需资源目前无效而无法继续运行，该类任务不会接受CPU的调度。

任务状态在操作系统运行过程中由于资源的变化或程序指令而发生变化，各状态间的转换如图 8所示。ACoreOs上任务的创建到运行分为两个阶段，任务创建完成后状态并不是就绪态，而是休眠态，任务需要额外通过ACoreOs\_task\_start操作进入就绪态；处于就绪态的任务可被操作系统调度而获取CPU控制权进入运行态，或者因为被其他任务调用ACoreOs\_task\_stop而使其进入休眠态，或者被其他任务调用ACoreOs\_task\_suspend而使其进入阻塞态；任务在运行态可得到真正的运行，执行过程中可主动的调用ACoreOs\_task\_stop停止自身执行而进入休眠态，让出CPU，或者任务调用睡眠操作进入阻塞态，或者调用ACoreOs\_task\_suspend而挂起自身执行而进入阻塞态，或者由于调用资源获取操作但此时资源状态不可用而进入阻塞态，等待资源有效时被唤醒，或者高优先级任务就绪导致被抢占而进入就绪态等待再次得到系统的调度而进入运行态；任务处于阻塞态时，可被其他任务进行多种操作而发生状态迁移，ACoreOs\_task\_suspend操作并不会改变处于阻塞态任务的状态，ACoreOs\_task\_resume操作将唤醒阻塞态的任务，使其状态处于就绪态，或者在资源有效时资源回调函数唤醒因等待该资源而被阻塞的任务，使其进入就绪态，或者某些任务睡眠时间已到需要被唤醒，从阻塞态转换为就绪态。



图 8 ACoreOs任务状态转换图

### ACoreOs任务调度

从就绪状态的任务中，依据一定筛选规则选择一个任务使其取得 CPU 所有权，在处理器上运行，这种过程称为任务调度。与通用多核实时操作系统类似，ACoreOs也支持基于优先级的调度算法和基于时间片轮转的调度算法。

1. 优先级调度

ACoreOs系统内部集成了对基于优先级调度算法的支持，优先级调度算法是保证系统总是优先处理优先级较高的任务，在ACoreOs系统中，每一个任务都拥有一个任务优先级，主要是由程序开发人员在开发阶段手动指定的。当然，任务的优先级也不是一尘不变的，其在运行过程中可通过系统接口动态更改优先级。

ACoreOs 操作系统内核优先级调度又可以按执行过程中是否能够被更高优先级的任务抢占可分为：不可抢占的优先级调度算法和可抢占的优先级调度算法。

1. 不可抢占（non-preemptive）的优先级调度

不可抢占优先级调度算法是指一旦某个高优先级的任务被系统换入运行，会一直运行下去，直到任务执行过程中自主（如任务调用了睡眠操作）放弃CPU的控制权，系统才会重现选择当前就绪态中优先级最高的任务进行运行。当然，任务在执行过程中，可以被中断打断，但即使中断处理函数中使一个更高优先级的任务成为就绪态，在中断返回的时候，被打断的任务也不会被中断中就绪的更高优先级的任务抢占，除非被打断的任务在后续执行中自主放弃CPU控制权，更高优先级的任务才得以调度运行。如图 9所示，高优先级任务Task1获取到CPU控制权得到运行，运行过程中被中断打断，在中



图 9 不可抢占的优先级调度

断处理函数中，唤醒了比Task1拥有更高优先级的任务Task2，然而在中断处理函数退出时，Task2并没有抢占Task1的运行，较低优先级的任务Task1继续运行，直到其因为自身原因主动放弃了CPU控制权，此时更高优先级的Task2才得到系统调度获取到CPU而开始运行。

1. 可抢占（preemptive）的优先级调度

在系统任何执行时刻，都按照优先级高低来进行任务的调度，也就是，处理器上正在运行的任务一定是系统目前处于就绪态的任务中优先级最高的。

就绪态中最高优先级的任务会优先被系统调度运行，在其运行过程中，系统严格保证其对CPU的占用权，目的是尽可能让其运行完成，中途任何较低优先级的任务都不会打断高优先级任务的运行。然而，一旦最高优先的任务执行中，因为某些资源目前无法立刻获取，被阻塞而被系统换出，系统才会从就绪态任务中选择最高优先级的任务运行，但如果某时刻最高优先级任务等待的资源重新有效，最高优先级的任务又重新就绪，那么此时系统会让当前正在执行的较低优先级任务让出CPU，换入最高优先级的任务运行，也就是较高优先级的任务会抢占较低优先级的任务运行。

如图 10所示，高优先级任务Task1运行中遇到等待事件（例如执行一段时间后主动睡眠）而放弃CPU而进入阻塞状态，此时系统中处于就绪态的最高优先级的任务Task2获取到CPU控制权开始运行，运行过程中，之前由于等待事件而阻塞的Task1因等待事件到来而重新就绪，此时由于Task1的优先级高于Task2并且是系统目前就绪态优先级最高的任务，Task1将抢占Task2，获取到CPU控制权开始运行。



图 10 可抢占的优先级调度

1. 时间片轮转调度

在实时系统中，不同的任务可以共享同一个任务优先级，为了保证同优先级下的各个任务调度的公平性，优先级调度算法同时也启用了时间片轮转的策略，称之为基于优先级的时间片轮转调度。在此调度策略下，如果有多个任务共享同一个任务优先级，并且又是系统目前处于就绪态任务中优先级别最高的，那么系统会从该优先级对应的就绪链表中选择表头对应的任务进入运行，待该任务运行一段时间后，系统会重新换入另外一个同优先级的任务进入运行，而将该任务放置到同级优先级链表的链尾，等待后续调度，任务从运行到后来被系统换出换入另一个同优先级的任务运行的这段时间称之为时间片（time slicing）。

如图 11所示，任务Task1、Task2、Task3三个任务共享同一个优先级，任务Task4优先级较前者稍高，同级优先级之间系统采用的是时间片轮转调度算法，Task1消耗完自身时间片后，系统分配同样的时间片给Task运行，Task1置于同级优先级队列的尾端，Task3在Task2消耗本次时间片后得到运行，依次类推，Task1-Task3交替运行，当然在运行过程中也会被高优先级任务Task4抢占，Task4执行结束后如果当前系统中没有更高优先级任务就绪则会继续执行被抢占的任务，直到其时间片消耗完。



图 11 时间片轮转调度

### 任务间通信机制

任务间通信机制是多任务之间相互同步[[52](#_ENREF_52)]与通信的主要手段，主要用来协调各个任务的执行。在多任务的实时系统中，一项工作往往需要借助多个任务共同完成，而这些任务之间或多或少都需要相互协调，相互配合。为了满足系统中各个任务之间通信的需求，ACoreOS 内核提供了消息队列（message queue）、环形缓冲（ringbuf）、异步信号（signal）、信号量（semaphore）、事件（event）和标志集（flagset）等几种机制。

1. 信号量机制

信号量提供任务之间同步与互斥的能力，也是任务之间同步互斥最常用的一种机制，ACoreOs中将信号量细分为三种类型：

1. 二值信号量：用于实现对单个资源的访问，尤其是对临界区资源的互斥访问；其也可以用于两个任务之间的同步，一个任务获取，另外一个任务释放，但是不允许任务嵌套访问。
2. 互斥信号量：与二值信号量类似，既能提供同步功能，也能提供互斥功能，唯一不同是互斥信号量可以被嵌套访问。
3. 计数信号量：提供对使用具有多份资源的互斥机制，实现对两份或多份资源的访问保护。

以上信号量用在互斥时，可能出现优先级反转[[53](#_ENREF_53)]的问题，导致高优先级的任务因为等待某个信号量，而这个信号量需要一个较低优先级的任务释放，而这个较低优先级的任务由于被另外一个稍高优先级任务抢占而不能运行，导致高优先级的任务也无法运行，违背了系统基于优先级调度的法则，在ACoreOs中，使用优先级继承或天花板算法来解决该问题。

1. 消息队列

消息队列是一段内存缓冲区，用来存放任务之间、任务与中断之间等相互通信的数据，简称为消息，其中可以放置的消息个数和大小是可以在创建时候配置的。消息队列自身完成对内存缓冲区的维护，维护目前缓冲区中有多少消息，是否可以继续写入消息，是否可以读取消息等。

消息队列的状态有5个：（1）无消息，消息队列此时为空，里面没有存放任何有效数据；（2）无消息并且有任务等待接收消息，消息队列中没有数据，然而有任务尝试从中获取数据，消息队列可能会阻塞当前任务的运行；（3）消息队列中有消息等待被任务接收，消息队列中存放有效的消息，任务可以接收到有效的消息；（4）消息队列满，消息队列中已达到饱和，如果继续发送消息，任务会被阻塞；（5）消息队列满并且有任务等待发送消息，消息队列中已经充满消息，对于继续发送的任务，需要进行阻塞。通常，消息按照FIFO（先进先出）方式进入消息队列等待被任务接收，对于紧急消息，消息队列会将其放置到队列头部，首先被任务接收处理。

任务可以被阻塞在一个空的消息队列上，等待另外一个任务发送消息，实现两个任务之间的同步操作。任务也可以同时向一个消息队列中写入数据而实现任务之间的互斥。

1. 环缓冲

与消息队列一样，环缓冲也是一种通信机制，在环缓冲数据区域中存放任务间或任务与中断服务程序间通信的内容。

与消息队列不一样的是，环缓冲数据区域没有相互独立的消息，整个环缓冲数据区域都是用来存放通信数据的，环缓冲内部维护一个读指针和一个写指针，分别指示可用数据空间的起始和可用空闲空间的起始。对环缓冲的读和写操作可以由两个访问源同时（并发）进行，即一个访问源在读（写）环缓冲时另一个访问源可以同时写（读）环缓冲。在环缓冲上的任何操作都不会改变任务状态。为了提高效率，内核在进行数据操作时不会屏蔽中断响应和调度响应，这其间的一致性应该由用户来保证。

### ACoreOs中断处理

中断是硬件机制，其是外部设备通知CPU进行相关操作的一种手段。实时系统必须快速地对外部产生的中断请求进行响应，完成与外部环境的实时交互。ACoreOS 向 ISR（Interrupt Service Routine）提供了一些系统调用，以便 ISR 能够使用内核的部分功能。为了减少中断对实时系统性能造成的影响，必须确保关中断的时间尽可能短，因此，ISR 只能完成一些必要的操作，例如接收网卡数据以及读取其他设备输入，对中断后续处理通过代理给任务来完成。

中断发生时，ISR必须保存现场，在ACoreOs中，由于ISR只会修改部分寄存器的数值，因而只需要保存可能要修改的寄存器的数值，待中断处理完，在中断退出时，恢复修改的寄存器。其执行过程一般分为三部分。

1. 中断进入：保存 ISR 中要修改的寄存器（通过通用处理完成）；通知 ACoreOS 进入 ISR；并根据当前状态判断是否发生中断嵌套，对于不是中断嵌套的还需要进行任务堆栈和中断专用栈之间的切换
2. 用户ISR：关调度；使能中断；进入用户注册的 ISR。用户 ISR 中可以调用内核提供的部分系统接口，与任务进行通信与实现同步。如果用户没有对此中断注册相应的处理函数，默认情况下打印中断号
3. 中断退出：禁止中断；通知 ACoreOS 退出ISR；中断嵌套解除时，对于退出的是最后一级中断，完成对栈进行切换，从中断栈切换到先前被打端的任务栈；开调度；此时会触发调度器重新调度，如果系统中有更高优先级的任务处于就绪态，则切换到更高优先级的任务执行；如果当前执行任务有异步信号需要处理，则进行异步信号的处理；恢复 ISR 中修改了的寄存器的值（通用处理完成）；退出到上一级被中断的 ISR 或者被中断的任务

### 自旋锁技术

自旋锁[[50](#_ENREF_50)]是类似于互斥锁、信号量或互斥区等用于进程间或任务间同步机制。自旋锁由于不断的查询某个充当锁的变量可用性而在某些场合表现更高效而经常被用在对称多处理器操作系统内核中。自旋锁比互斥锁、信号量以及互斥区有如下的优点：

1. 任务不会被抢占而导致任务切换
2. 不用维护和修改额外的管理数据结构（目前谁拥有这个同步对象或者是否被锁，拥有该对象的时长，是否超过了设定的超时时长）
3. 不需要因为为维护公平的先进先出机制而执行额外的操作

自旋锁主要通过忙等待实现，在多核平台下，可高效用来保护互斥访问的共享资源，该共享资源临界区执行时间非常短，而且也不会触发任务切换，避免导致可能出现的死锁或者优先级反转问题，如果采用互斥锁机制保护这类资源，可能由于任务切换需要的操作时间远远大于该资源临界区执行所需要的时间，导致系统因为当前任务临时获取不到该资源触发任务切换，待该资源有效时又重新切换进当前任务，从而可能严重降低系统性能。自旋锁让任务在获取不到该类资源时不断在原地轮询资源的状态，而不用阻塞当前任务的执行，待轮询的资源被运行在别的CPU上的任务释放后，该任务可继续执行，期间并不用引入任何的系统切换开销，仅仅只有轮询开销，由于这类资源临界区执行时间很快，不会造成系统性能下降。

## 执行重放挑战及问题

本节给出了本文在提出执行重放方案时所必须面临的挑战和问题，主要包括探针效应、事件发生位置标记问题、上下文特殊性、多核事件顺序、系统特殊性以及数据竞争。

### 探针效应

无论是对目标代码插桩和源代码插桩，都会引入探针。基于软件方式的探针插入到系统中，用来监测系统行为的同时，也可能会改变被监测的系统的行为。这种对系统行为产生影响的效应称为探针效应。其存在于并发程序中并首次被Gait[[43](#_ENREF_43)]提出。

下面这个例子将很好的描述探针效应。有两个系统任务，如图 12所示，任务A和任务B共享一个资源X，这两个任务都是在临界区中对X进行访问，图 12中白色部分表示对X资源进行访问，现假设程序正确执行时访问顺序是：任务B先访问X，之后任务A访问X。根据图 12所示，正确的访问顺序没有得到满足，所以导致了系统故障。由于开发人员很疑惑系统的此次行为，在程序再次执行前加入了一个探针（例如printf语句）到程序中，在图 13中用黑色表示。



图 12 导致系统出错的执行

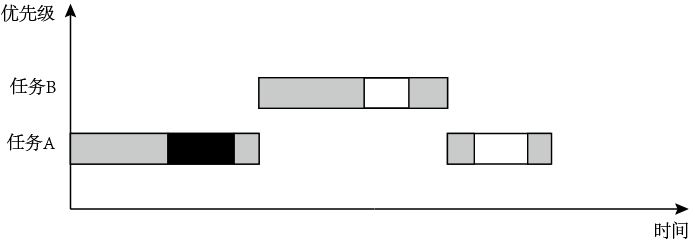


图 13 相同的执行，加入了软件探针，"躲过"了故障

第二次执行时，探针的执行会延长任务A的执行，这会导致在其进入临界区前就会被任务B抢占，这样任务B先于任务A进入临界区对资源X进行访问，而此次的访问顺序刚好满足正确的访问顺序，导致上次的故障不会再现。

由于本方案采用了软件方式的源代码插桩，从而引入了探针效应，如何解决插桩带来的探针效应是本文研究的一大难点。

### 事件发生位置标记问题

在记录执行阶段，不仅需要记录事件发生时任务的相关信息，还需要记录各个事件发生的位置。这里的位置其实指的就是事件发生时，任务执行到达的当前位置。在程序中一般用PC指针来进行标识。对于不包含循环或者递归调用的任务，通过上下文的PC指针就能够唯一确定事件发生的位置，但一旦程序中包含了循环或者递归，由于同一个循环体或者递归函数体被多次访问，从而仅仅依靠PC指针无法精确的指出事件发生的位置，因此必须引入额外的信息辅助PC指针唯一标识事件发生的位置。确定辅助信息也是本文面临的一大挑战。

### 上下文特殊性

ACoreOs是一个典型的嵌入式实时操作系统，其系统调用接口并不是采用被普通桌面操作系统(如Linux、Windows等)广泛采用的软中断机制实现的而是直接采用函数调用实现。在这种设计中，系统并不刻意区分用户态和系统态，可以认为所有的任务都运行在系统态。任务在其整个生命周期中仅存在一份上下文，其中掺杂了任务代码执行路径和系统代码执行路径，这不同于桌面操作系统，在普通桌面操作系统中，应用程序借助系统提供的接口在从用户态切换进入系统态时，操作系统会为其产生一份上下文，称为任务用户态上下文，该上下文反映了该任务在用户态的当前执行状态；一旦进入到系统态，任务将重新拥有新的一份上下文，称作任务系统态上下文，该上下文反映出任务在请求系统服务时，系统正在为其服务的当前执行状态。

ACoreOs任务在执行过程中，一旦发生切换，系统会保存该任务的上下文信息，然而，所有任务的切换都发生在系统的Thread\_dispatch函数中，系统为任务在上下文切换时保存的上下文信息基本是一样的，并且任务执行过程中被系统切换的操作是随机的，不是每次都发生在相同的场合，从而系统在任务执行中产生的一份上下文对重放执行阶段是没有任何意义的。任务整个生命周期仅有一份上下文加大了记录和重放的难度，如何正确的重现任务上下文是本文需要解决的一大挑战。

### 事件发生顺序

在多核处理器系统中，额外的不确定性来源于系统中多个CPU之间产生事件的并行性。主要包括三种情况：同步事件并行、同步和异步事件并行、异步事件和异步事件并行。

同步事件之间的并行也称为同步竞争，是内存竞争中的一种情况，多核环境下，每个核之间执行的代码是不相关的，产生的事件也是相互独立的，从而正在执行该应用中任务的CPU核心之间也会存在相互的竞争。正确的标示各个同步事件相对发生位置也必须的，例如，给定一个应用程序其中包含两个任务，任务A和任务B。其中任务A的任务入口函数是task1，任务B的入口函数是task2，其具体执行代码如图 14所示。任务A被操作系统调度到核心编号为1的CPU上运行，而任务B被调度到核心编号为2的CPU上运行。图 15和图 16分别给出了任务A和任务B可能的执行(假设Semaphore的初始值是0)。任务A和任务B并行竞争同一个信号量，如果两个任务同时访问信号量，哪个任务先获取信号量的操作权，操作系统是无法不保证的。从图 15和图 16中可以看出任务A和任务B并行竞争信号量会影响任务的执行流程，图 15中，由于任务A和任务B同时访问信号量，而此时任务A成功获取到了信号量操作权，那么任务A在执行中会由于获取不到有效的信号量而阻塞，导致操作系统产生了一次任务切换，此时任务A被换出，任务A释放了对信号量的操作权，位于另一个核上运行的任务B稍后获取到了信号量操作权执行了信号量释放操作，此时信号量进入有效状态，任务A重新就绪，进而后续任务A又会被切换进CPU核1继续运行，直到此次时间片用完或者因获取不到后续有效的信号量而被切换。图 16中，在竞争信号量过程中，位于核0上的任务B成功竞争到信号量操作权，对信号量进行释放操作，稍后，任务A才获取信号量的操作权，进行信号量操作，由于之前任务B释放了信号量，任务A可获取到信号量，而不会因为信号量无效而被阻塞触发切换而被系统换出。

图 15和图 16给出的任务A和任务B执行都是可能出现的，但两者结果却存在差异，其根本在于多核环境下应用程序中任务之间同步竞争造成任务对同步事件操作顺序不确定，为了重现应用程序在多核上的执行结果，其任务在多核上重放执行时产生的同步事件的顺序是必须保证与记录执行阶段是完全一致的。

|  |  |
| --- | --- |
| 1 | ACoreOs\_task task1(*void*\* param){ |
| 2 | while (true){ |
| 3 | ACoreOs\_semaphore\_obtain(semaphore); |
| 4 | //do some other things |
| 5 | } |
| 6 | } |
| 7 |  |
| 8 | ACoreOs\_task task2(*void*\* param){ |
| 9 | while (true){ |
| 10 | ACoreOs\_semaphore\_release(semaphore); |
| 11 | //do some other things |
| 12 | } |
| 13 | } |

图 14 任务相应执行代码



图 15 并行访问信号量时任务A先获取到操作权



图 16 并行访问信号量时任务B先获取到操作权

同步事件和异步事件之间、异步事件和异步事件之间发生顺序也具有并行性，例如一个CPU正在进行同步操作，而另外一个CPU产生了中断，或者两个CPU并行进入中断处理函数等，这些产生的事件也会存在类似于上面提到的因为事件发生的顺序不确定性对任务的执行造成影响。不同的是，这两类事件的并行性，操作系统本身并没有引入机制来对事件顺序进行串行化，或者标记出哪一个事件先于另外一个发生。如何正确解决这类事件发生的并行性对程序执行造成的不确定性也是本文研究的一大难点。

### 系统特殊性

目前已有的执行重放方案都是面向于普通桌面操作系统（Linux以及Windows）以及服务器系统（类Unix系统）的。本文的目标系统是ACoreOs，与一般的操作系统不同的是，ACoreOs的应用程序的开发环境是一个基于宿主机—目标机的开发架构。应用程序的开发、编译在宿主机端完成，目标机端负责运行操作系统以及系统之上开发的应用程序。由于程序的执行发生在目标机端，动态产生的信息也发生在目标机端，然而，在重放执行阶段，宿主机端需要读取记录执行阶段产生的日志信息并控制程序的二次执行，因而，产生的日志信息必须被上传到宿主机端。同时，在任务重放执行阶段，运行在宿主机上的重放模块必须能够动态获取目标端任务执行的相关信息和控制目标端任务的执行过程。

## 本文研究限制

在多核处理器系统中，另外一种的内存竞争是数据竞争：多个任务之间通过共享内存来进行数据的交互。正如本文前面提到，由于数据竞争记录开销过大，而本文也没有给出有效的软件方式减小这种开销，使其可使用于实时系统，故本文对数据竞争导致的不确定性重放不作深入研究。

## 本章小结

本章介绍了相关技术及其原理。首先介绍了代码插桩技术的特点，然后阐述了调试技术以及嵌入式开发中广泛使用的交叉调试技术，接着详细介绍了本文的目标操作系统—嵌入式多核实时操作系统ACoreOs，介绍了ACoreOs的任务状态、任务调度、任务间通信机制以及中断处理等基本机制，为后面实现和设计ACoreOs下应用程序执行重放方案奠定了基础，本章最后，指出了本文在提出执行重放方案时面临的挑战和相关问题以及本文研究限制。

# 执行重放方案的设计

本章介绍了ACoreOs下应用程序执行重放方案的设计。首先根据执行重放面临的挑战和问题，结合目标系统开发环境，提出了适用于多核实时系统ACoreOs应用程序执行重放整体设计方案，接着详细介绍了该方案中各个模块的具体设计思路，最后详细阐述了针对重放方案中的各个挑战和问题提出的相应解决方案。

## 执行重放方法整体设计

本文提出的执行重放方案是基于宿主机和目标机的交叉环境架构。目标机是搭载两个Power架构处理器的Freescale MPC8461D开发板，其上运行多核实时操作系统ACoreOs、目标机端调试代理、应用程序、信息记录模块。目标机端调试代理作为目标机操作系统上一个应用程序随系统启动，生命周期与操作系统生命周期相同，其主要监测目标系统上任务的执行情况，接收宿主机端发送的调试命令完成相应的调试动作。目标机端调试代理也提供了上传数据到宿主机端的接口，该接口用于信息记录模块动态上传在记录执行阶段记录的日志信息。应用程序即需要重放执行的目标程序，由开发人员编写。宿主机是普通的桌面机，部署了ACoreOS应用开发环境LambdaAE、宿主机端调试服务模块、日志解析过滤模块以及重放模块。宿主机端调试服务模块提供了与位于目标机端调试代理通信的能力，并提供操作目标机端系统接口以及改变应用程序状态的功能。重放方案的整体架构如图 17所示，其中灰色部分是本文提出的执行重放方法所必须的模块。



图 17 执行重放方案整体设计图

本文提出的方案包含三个部分：目标机端信息记录模块、宿主机端日志分析过滤模块和重放模块。目标机端信息记录模块采用源代码插桩的方式在系统执行路径上插入探针用来在任务记录执行阶段动态的记录其执行过程中相关日志信息，包括任务创建相关信息，任务执行中产生的同步或异步事件信息、任务从外界获取到的数据信息等，并在合适的时机将信息上传到宿主机端供后续分析。宿主机端日志分析过滤模块负责将记录执行阶段得到的日志信息进行解析，并从中提取出重放器需要重放的任务相关信息。重放模块以解析过滤模块得到的数据作为输入，调用宿主机端的调试服务接口控制目标应用的二次执行，在执行中，通过设置断点，动态的控制任务程序的切换顺序，数据的回写等，完成目标任务执行重放。

本方案中程序的执行重放流程如图 18所示，包括记录执行阶段和重放执行阶段。

在记录执行阶段，主要完成对目标应用程序执行信息的记录，包含的步骤有：

1. 给目标机上电，启动目标机，并建立目标机和宿主机间的连接；
2. 通过集成开发环境将开发完成的应用程序和插桩之后的ACoreOs操作系统一起下载到目标机上，初始化和启动目标记录模块并启动目标应用；
3. 目标应用过程中，插桩在操作系统执行路径中的探针会动态记录各个任务的执行状态信息；
4. 信息记录模块为了减少其日志上传过程对任务执行造成的影响，将记录的信息暂时缓存在目标机端；
5. 信息记录模块待缓冲的信息到达设定的阈值，开始调用目标机调试代理接口对信息进行上传；
6. 集成开发环境实时监听来自目标机的数据上传请求，并将上传的信息保存到指定的日志文件中。

在重放执行阶段，根据记录的日志，完成日志数据分析过滤，并依据分析后的结果指导重放模块完成对目标应用的二次执行，包含的步骤是：

1. 由于在执行重放阶段，目标应用仍然需要被下载到目标机上运行，执行重放阶段首先也需要和记录执行阶段一样建立和目标机的连接；
2. 下载目标应用程序和对应的ACoreOs操作系统到目标机；
3. 日志分析过滤模块读取记录执行阶段上传的日志文件，进行日志的解析过滤操作，并将分析后的结果交给重放模块；
4. 重放模块根据分析的结果，进行自身环境的初始化；
5. 重放模块与目标机之间建立连接，并获取目标机的控制权；
6. 重放模块按照获取的日志分析结果，通过调试接口设置断点，在断点处理过程中比对任务上下文信息，进行任务的切换和恢复，控制重放任务的执行，直到任务完成运行。

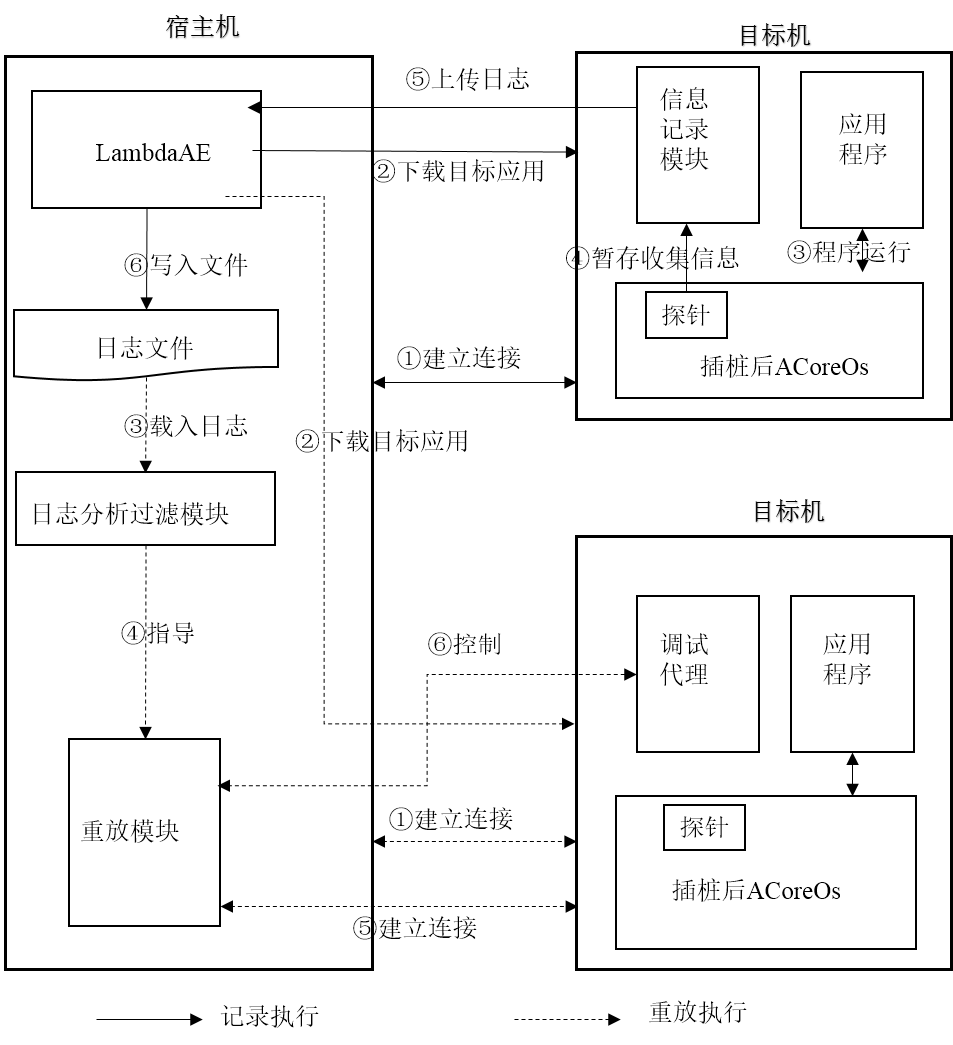


图 18 执行重放执行流程

本文前面介绍了执行重放方案的整体设计和执行流程，设计方案中主要包括了三个模块：信息记录模块、日志分析过滤模块、重放模块。接下来本文将详细介绍各个模块的具体设计。

## 信息记录模块的设计

### 确定程序执行的不确定性

本文中前面介绍执行重放技术时，提到执行重放由记录执行阶段和重放执行阶段两阶段构成。记录执行阶段主要记录应用程序执行过程中的不确定性因素，以指导程序在重放阶段执行，达到程序在重放阶段和记录阶段执行产生相同的效果。确定程序的不确定因素是程序在重放执行阶段和记录执行阶段执行产生完全一样效果的前提，因此确定程序的不确定性因素是信息记录模块必须解决的首要问题。

程序的不确定性因素跟随执行重放级别的不同而不同，执行重放整个操作系统和执行重放其上的应用所应对的不确定性因素是不同的。由于本系统执行重放的对象是操作系统上运行的程序，本文将详细讨论执行重放ACoreOs应用程序所面临的不确定因素。本文中将目标系统应用程序的不确定因素分为两大类，控制流不确定性和数据流不确定性。

控制流指的是应用程序包含的任务在执行中由于可干预应用程序执行路径的事件导致应用程序的任务被操作系统换入换出来回切换而形成的任务执行流。例如一个应用程序创建了3个任务，分别是任务A，任务B和任务C。在记录阶段任务A执行一段时间后，由于任务A时间片耗尽，操作系统换进任务B开始运行，一段时间后，在任务B时间片使用完后，任务C得到了运行。此时应用程序的执行路径就是任务A 🡪 任务B 🡪 任务C。这类可引发任务切换的事件主要包含同步事件（任务显式调用的可触发自身切换的操作系统接口，如信号量获取和释放接口）和异步事件（任务在执行生命周期中，在任何可发生位置随机产生的事件，本文主要指中断）。控制流主要干预应用程序的执行路径和执行步调，确保控制流与记录执行阶段相同是确保应用程序在重放执行阶段和记录执行阶段执行产生的效果相同的必要条件。与已有的研究解决方案不同的是，本文提出在执行重放阶段，必须保证产生同步事件的系统调用接口被执行，这主要是由于ACoreOs系统特殊性所决定的。

数据流指的是应用程序任务执行过程中与外界交互产生的数据。这类数据主要包括任务启动参数、任务间消息通信、执行效果具有不确定性的系统调用接口（如ACoreOs\_task\_get\_pid）以及返回值具有不确定性的某些库函数（如rand）等。数据流的一致性保证任务在重放阶段具有和记录执行阶段相同的输入数据和中间数据。本文此处以及后文中所谓的一致性是指与记录执行阶段产生结果完全相同。数据流中不需要引入任务的输出，因为任务输出的一致性可通过控制流和数据流进行保证。控制流和数据流两者的一致性可保证程序在记录执行阶段和重放执行阶段产生完全相同的效果。

### 记录程序执行不确定性探针设计

信息记录模块主要完成对目标机应用程序执行中不确定性因素的记录。记录模块主要通过在系统执行路径上插入探针来获取任务的执行信息，然而进行插桩必须考虑以下两个问题：

1. 在系统执行路径的哪些位置插入探针；
2. 插入探针而引入的探针效应如何减小。

对于问题（1）主要考虑任务执行过程中哪些信息是重放模块所必须的，应该提供哪些接口，并且这些接口需要插桩在系统的哪些执行路径上。

在任务的记录执行阶段，主要需要记录的信息分为控制流信息和数据流信息。控制流信息包括同步事件、异步事件、任务切换上下文事件、中断打断上下文事件，中断退出上下文事件等信息。数据流主要包括任务执行过程中从外界获取到的输入数据、任务间通信方式下接收数据、不确定的系统接口返回值等。

对于问题（2）需要考虑插桩的探针在执行时具体会给系统产生哪些影响。由于程序执行过程中，不仅仅目标应用会产生这类事件，而且系统中其他的任务也会产生，所以信息记录模块必须高效记录产生的事件信息（后文中也称日志信息）而不能对系统的实时性造成较大的影响。

探针设计的目的在于记录信息，由于ACoreOs开发环境的特殊性，记录的信息不能够被全部放置在目标机内存中，因而探针将记录的日志信息动态上传到宿主机端。为了减少探针在执行时对系统造成的影响，本文提出了先将记录的信息缓冲在目标机内存然后在合适的时机将信息上传到主机端。探针设计时需要考虑几个问题：系统实时性、多核并行记录、探针执行中不可引入新的额外事件。任务在记录执行阶段，需要同时记录控制流和数据流信息。记录开销可通过式子 进行量化。

其中 表示记录阶段执行引入的额外的总的开销，表示第个探针接口被触发的次数，表示第个探针接口收集信息所需要付出的代价，主要指花费的时间，表示信息记录模块提供的探针种类总数， 表示信息记录模块平均记录每条日志需要花费的时间。根据上式，与各个变量之间的关系都是单调递增的。

本文主要采用三种方式来尽量降低日志记录带来的开销。首先，优化探针设计，减小，尽量减少所需要引入的探针种类，这主要通过分析ACoreOs系统自身特点进行优化，例如仅需要记录中断进入事件而不用记录中断退出事件；接着，减少，优化单个探针在记录单条信息的信息量，该优化通过分析记录信息的必然性，去掉对不必要信息或者可以推导出来的信息的收集，同时将需要记录的信息组织成相应的数据结构，保存的时候采用二进制数据存储，从而进一步降低记录信息的大小。最后，减少，优化探针执行时对每条信息记录所需要的开销，主要通过采用写入控制、缓冲区管理、上传三者相分离的设计，其设计框架如图 19所示。



图 19 信息记录模块的分离式设计

写入控制模块供探针调用用来向缓冲区中写入记录的信息，其仅负责从信息记录模块维护的内存缓冲区中分配需要的空间，并将信息保存到分配的内存中。缓冲区管理模块负责缓冲区的维护，其完成的主要工作当缓冲区内存不充足时向系统申请更多的空闲内存补充到缓冲区中，当缓冲区充足时，归还部分空闲内存给操作系统。上传模块负责在缓冲区中驻留的日志信息达到上传阈值时调用目标机端的调试代理接口上传日志，并尽快排空缓冲区数据。

同时，在记录执行阶段，存在多核CPU核心同时触发插入在执行路径中的探针执行，进而同时操作缓冲区，为了保证多核并行访问缓冲区的一致性，本文借助自旋锁对缓冲区的访问进行保护。由于相同类型的事件可以发生在不同的CPU上，为了能够标识出事件发生在哪个CPU上，本文在记录每条日志信息时，会附加记录产生事件的CPU编号（后文称之为CPU索引事件），考虑到尽可能少的减少记录带来的开销，对于某个CPU连续产生多个事件，只有第一个事件需要附带记录CPU编号，后续该CPU上产生的连续事件不需要额外的记录开销，直到别的CPU产生了事件信息。

## 日志分析过滤模块设计

日志分析过滤模块包含两大模块：日志分析模块和日志过滤模块。日志分析模块提供日志解析的能力，将应用程序在记录执行阶段得到的日志信息解析组织成过滤器所需要的数据结构。由于在记录执行阶段，目标系统中不仅运行目标应用程序，而且存在一些其他的任务（例如前面提到的目标端调试代理、网络传输等任务），日志过滤模块一方面需要筛选出目标应用产生的事件，另一方面需要从筛选出的信息中分析保留必要的事件信息，去掉任务重放阶段并不需要的日志信息。

### 确定目标应用中的任务

记录执行阶段，系统中除目标应用运行外，还存在其它为应用程序服务的任务，这些任务也会在执行中触发探针产生相关事件信息，从而导致记录的信息中包含了该类任务的干扰信息，在重放阶段，由于仅仅重放目标应用中的任务，对于这类任务的事件信息必须被过滤。然而，对于同一个应用程序，在其不同次执行中，由于系统自身原因导致每次分配给应用中的各个任务的任务标识符存在不确定性，进而不可以使用任务的标识符作为区分目标应用中的任务和系统中其它任务的标准，必须选择和任务相关的一个特征，无论被执行多少次，通过此特征可确定该任务。本文中采用任务名充当任务上述特征，在ACoreOs中，不同的任务在系统中的任务名也是不同的，而任务名在任务创建时也可被动态指定，只要确保目标应用中的任务名都是不同的，而且不同于系统中其它任务，就很容易使用任务名作为任务的唯一标识。在过滤阶段，很容易根据任务名分析出该任务在创建过程中，系统为其分配的任务标识符，进而可以获取到该任务执行过程中产生的所有事件信息。

### 确定必需的信息

日志分析过滤阶段，必需保留重放模块所必须的所有信息，过滤掉其他的干扰信息。前面提到，本文将不确定因素分为两部分：控制流不确定性和数据流不确定性，所以分析过滤模块需要保留改变控制流和数据流的事件信息。对于多核实时操作系统而言，影响控制流的典型事件是同步、异步操作事件；影响数据流的典型事件是调用返回值具有不确定的函数或者系统接口所产生的数据事件。

本文中，同步事件是程序在执行过程中，程序代码中显式调用操作系统接口产生的一类事件(如信号量获取和释放，消息队列发送和接收等)。这类事件有以下两个共同点。

1. 在程序的二次执行中一定会重现，只不过由于操作系统不确定性的调度产生的顺序可能和记录执行阶段不同(特别是在多任务的应用中)。
2. 可引发任务切换，但不是每次调用都会引发任务切换。
3. 由于中断处理函数中可以调用部分同步接口，从而有些同步事件可以嵌套在中断事件中

正是这类事件的这些特性，必须分析保留必要的同步事件，在重放阶段指导重放模块正确的再现控制流。

异步事件指程序在执行过程中，可能随时产生的打断任务执行的一类事件(如实时钟中断)，这类事件具有发生时位置和时刻的不确定性，在本文中主要指中断。异步事件也可触发任务的切换，与同步事件一样，并不一定每次发生时都触发任务切换。同时，异步事件可以嵌套在同步事件中，异步事件本身也可以相互嵌套，在过滤阶段，必须分析出哪些控制流的改变是异步事件造成的，哪些实质上是同步事件造成的，只不过异步事件嵌套在同步事件中，造成异步事件触发切换的假象。本文主要采取两个手段来辅助分析：（1）在记录执行阶段，标记每次任务切换的触发因素，是同步事件造成还是异步事件造成的；（2）过滤阶段，根据记录的事件信息模拟程序执行过程，梳理程序执行流程，确定触发切换的事件。同时，由于本文使用的交叉调试器并不支持对操作系统自身代码的调试，从而，对于嵌套在同步操作内的触发控制流变更的中断事件，必须采用外层的同步事件来进行替换。

数据流事件不同于前面提到的控制流事件，数据流事件每次产生时并不会导致任务的切换，仅仅影响任务执行中获取的数据。为了保证程序重放执行过程中，从外界获取的数据完全一样，对于目标应用中的数据流事件不能进行任何过滤操作，必须保证数据流信息的完整性。

### 单核日志过滤模块

虽然ACoreOs是多核操作系统，但目标应用程序创建的任务也有可能仅运行在一个处理器核心上，那么此时目标应用就如同运行在单核环境下，为了提高单核环境下重放速度，避免不必要的事件对重放的干扰，本文依据目标应用程序中所有任务运行过程中是否引入了多个处理器划分为单核日志过滤模块和多核日志过滤模块。

单核日志过滤模块设计总体框架如图 20所示。由于在任务执行中，并不是每次同步或者异步事件都能够触发任务的切换，对于单核过滤器，在控制流上，只需要保证任务之间的切换顺序和记录执行阶段完全相同就能够达到执行重放的目的，因此单核过滤器仅需要分析出触发任务切换的事件信息，并保留这些信息。在任务切换过程中，任务切换事件标示了具体任务切换动作：哪个任务被系统换出，同时换入了另外哪一个任务；由于系统中包含其他的任务，这些任务也会参入到目标应用的控制流中，例如目标应用中的某个任务（任务A）切换到系统其它任务上，一段时间后系统其它任务又切换到了目标应用中的另外一个任务（任务B）上，而系统中的其他任务由于记录执行阶段获取不了完整的事件信息，在重放阶段无法被正确重现，对于有其他任务参入的控制流，必须将这些任务的切换从目标应用控制流中剔除，对于上面给出的例子，剔除之后应该是任务A直接切换到任务B。

在数据流上，数据流影响程序执行中的内部状态，为了保证重放阶段任务内部状态信息的一致性，在过滤阶段必须保留所有的数据流事件信息。



图 20 单核版本日志过滤模块设计框架图

### 多核日志过滤模块

多核日志过滤模块的总体设计框图如图 21所示，所考虑的问题与单核版本的大体相同，所不同的是由于目标应用执行中引入了多核，在控制流上，目标应用的任务间执行步调存在相互的制约，因此在过滤日志时必须保留这些潜在的制约因素，同时也需要去掉没有触发任务切换的异步事件和嵌套在同步或者异步事件中的冗余事件；在数据流上，也需要保留所有的数据流事件。同时由于多核的引入，必须保留一些CPU索引事件信息，让重放模块有能力正确获取任务产生事件信息时所处的CPU对应的编号，这是保证重放时确保事件发生在相同CPU上的前提。本文中，将所有的CPU索引事件提取出来，与过滤之后的各个任务事件信息按照时间戳升序排列，保证了CPU索引事件可用来标识之后发生的事件所在的CPU编号，然而，由于提取了所有的CPU索引事件，导致合并得到的日志链表中，存在大量的冗余CPU索引事件，例如CPU1索引事件信息后，紧跟一条CPU0的索引信息，接着又是CPU1索引信息，之后才是有效的任务事件信息，其实最终结果仅需要一个CPU1索引事件，CPU0事件和另外一条CPU1事件是不需要的，而这些冗余的CPU索引信息会对重放模块正确分析重放流程造成干扰，必须过滤掉。



图 21 多核版本日志过滤模块设计框架图

## 重放模块设计

（执行）重放模块的功能是根据过滤后的日志信息创建并控制应用程序任务的二次执行，其必须提供对控制流和数据流信息的正确重放的能力。

### 重放控制流

应用程序在执行重放阶段，必须保证其控制流和记录执行阶段是完全一样的。本文前面提到，控制流的改变主要是任务间切换造成的，保证任务切换顺序和触发切换位置与执行阶段相同也就保证了执行流的一致性。本文中，通过分析记录执行阶段收集的日志信息，得出任务切换的流程以及每次切换前为任务保存的上下文（参见3.5.2）信息，并根据任务的上下文信息指导重放执行模块在目标系统中设置合适的断点，待断点命中并满足任务切换要求时，控制任务之间进行切换。

重放控制流主要包含两个方面：重放同步事件和重放异步事件。

### 重放同步事件

同步事件作为触发任务切换的一个因素必须能够被正确的重放，重放同步事件主要重现其对任务造成的切换效果。对于同步事件，重放阶段主要通过比对引入的同步事件计数器数值来确定任务执行是否到达其在记录执行阶段的切换点。由于产生同步事件的接口非常多，而且重现同步事件存在相似性，下面本文仅以消息队列发送这一同步事件作为代表进行介绍。

消息队列发送操作在消息队列中没有空闲空间时会阻塞当前发送任务，触发任务的切换，因此在记录的事件中会包含该事件信息。重放过程中，日志分析和过滤模块会分析出该事件触发了后续任务切换，将该日志信息提取出交给重放模块，重放模块根据事件信息类型从目标机端获取事件对应函数适当的位置作为断点坐标在目标系统上设置断点注册对应的断点回调函数。任务在重放执行中会命中该断点，触发回调，在回调函数中进行重放执行阶段和记录执行阶段的同步事件计数器数值比对操作。其中，重放阶段的同步事件计数器数值主要通过交叉调试接口动态从任务TCB中获取，记录执行阶段的同步事件计数器数值来源于记录得到的日志信息。如果两者数值相同，说明任务正确执行到切换点，可以进行任务切换操作，否则需要继续当前任务的执行。

对于同步事件的重放，其重放时需要设置断点的坐标为了减少记录开销而没有在记录阶段被记录，从而重放过程中必须获取到对应事件应该设置的断点的位置PC值，本文通过事件类型可以反推出事件发生所在的函数，从而可以根据函数名来获取函数的入口地址，对于像信号量获取、消息队列发送或者接收这样可能阻塞任务运行的操作，在重放过程中仅需要获取到对应函数的入口地址即可作为重放该事件时需要设置的断点坐标，而对于信号量释放由于其必须保证信号量被释放，因此，对于信号量释放事件必须获取信号量释放接口函数结束的PC值作为断点坐标。

### 重放异步事件

异步事件作为触发任务切换的另一个因素也必须能够被正确的重放，重放异步事件与重放同步事件类似，也是重现事件对任务切换造成的影响。本文中，异步事件主要指中断。中断的发生具有不确定性，本文中将中断发生位置分为两大类：一类是发生在操作系统代码中，另一类是发生在用户程序和库函数中。之所以采用这种分类方式，主要由于对应的处理方式不同。

对于发生在操作系统中的，一定是任务执行过程中调用系统提供的同步事件接口，系统在为应用程序服务过程中，中断打断任务执行，此类中断退出时如果产生了任务调度，可认为任务在调用系统的同步接口时产生了后续的任务切换操作，因为两者造成的影响是完全相同的，此时可使用该同步事件替换系统内发生的引起任务切换的中断事件。日志分析过滤模块会分析这类中断并使用相应的同步事件替换，重放该类中断事件其实是重放同步事件（参见3.4.2）。

对于发生在用户程序和库函数代码中的，由于其发生位置与任务的状态是非常相关的，因而必须严格精确的指示出其发生的位置（参见2.4.2）。重放该类中断时，由于在记录执行阶段记下中断发生时任务的上下文信息，因而首先根据上下文中的PC数值在目标系统的相应位置设置断点和注册回调函数，待任务运行命中断点，触发回调，在回调函数处理中，进行重放执行和记录执行阶段任务上下文信息的比对操作。上下文主要分为32个基本寄存器、几个特殊的寄存器以及任务堆栈校验和。其中，重放执行阶段任务上下文通过交叉调试接口动态从目标机获取，而记录执行阶段上下文则通过日志获取。如果两者比对完全相同，说明任务正确执行到切换点，可以进行任务切换操作，否则需要继续当前任务的执行。

### 重放数据流

与控制流类似，应用程序在执行重放阶段，也必须保证其数据流和记录执行阶段是完全一样的。本文在处理数据流时类似于控制流，在信息记录阶段，也会为数据流产生上下文，不过这个上下文仅仅含有产生该数据流时任务的当前PC指针。

在重放执行阶段，根据为每次数据流产生的上下文信息指导重放执行模块在目标系统中设置合适的断点，待断点命中的时候，根据记录执行阶段得到的信息指导重放执行模块回写对应的数据。由于数据流在每次产生时都会记录相应的数据，而每次记录的数据对于重放执行阶段都是必须的，所以不存在断点命中后的信息比对过程。同时，数据流并不会改变任务的执行流程，等待数据回写之后，并不会触发新的任务切换，而是仅仅继续当前被断点打断的任务。

### 重放阶段调度问题

执行重放技术需要程序在记录执行阶段和重放执行阶段被分别执行。在记录执行阶段，程序中任务的创建、调度执行都是由操作系统负责完成，任务状态的改变、迁移也是任务自身和操作系统共同管理和维护，不需要人为因素干预控制。然而，在重放执行阶段，由于要根据记录执行阶段收集的日志信息控制程序中相应任务的执行顺序，因而必须有能力在任务执行过程中干预任务的状态迁移、操作系统对任务的调度顺序，从而保证任务执行顺序的一致性(和记录阶段执行顺序完全一样)。

重放执行阶段为了保证任务切换顺序和记录执行阶段完全相同，重放模块必须干预任务调度。重放阶段，任务调度受控于重放模块和目标操作系统。重放模块主要干预任务的就绪和阻塞，在任务需要运行时，将任务的状态设置为就绪态；而在任务达到切换点，需要换入其他的任务运行时，将当前任务设置为阻塞态。而任务真正运行还需要操作系统进一步干预，重放模块一旦将任务就绪，操作系统在合适的时机分配给该任务时间片，任务得到运行，当然，该任务在执行过程中很可能会被系统中其他任务抢占而被换出，一段时间后重新被系统换入运行，这种由操作系统负责的换入换出操作并不会影响任务重放执行的正确性。

为了防止目标程序中的某个任务在重放执行过程中被该应用中的其他任务抢占发生任务切换，而这个任务切换在记录执行阶段并不存在，重放模块必须保证目标程序仅有一个任务处于就绪态，该任务可被系统多次的换入换出。

### 单核重放模块设计

前面，主要介绍了在设计重放模块必须考虑的问题以及相应的解决方案，下面将具体给出重放模块的设计。

单核重放模块的整体设计如图 22所示。单核日志分析模块负责读取过滤模块过滤后的信息，并分析出任务启动所必需的信息，这些信息托管给任务重放集管理模块进行维护，同时分析出下一个切换进入系统运行的任务和该任务会在执行到何处因为何事件而被系统切换出，这类信息托管给CPU管理模块，并相应更新任务的状态信息。重放任务集管理模块维护重放任务所有的信息，例如该任务执行的入口函数，入口参数，任务创建时的栈起始地址、大小以及任务的当前状态等。



图 22 单核版本重放模块设计框图

CPU管理模块根据单核日志分析模块分析出的需要执行的任务和相关触发该任务后续切换点事件分析出应该设置的断点位置设置断点并启动相应任务执行，待任务执行到设置的断点后，CPU管理模块确定切换点到达后会更新该任务的状态并通知单核日志分析模块进行下一轮的分析，抉择出后续需要调度进入执行的任务以及其切换事件，依次类推，直到日志分析结束或者应用程序完成执行。

### 多核重放模块设计

多核重放模块整体架构如图 23所示。该架构与单核版本的有些相似，但有些模块完成的功能是不同的。多核日志分析模块不同于单核版本负责分析下一个任务和其切换点，而是负责分析出下一个需要发生的事件和触发这个事件的任务以及任务产生事件时所在的CPU编号，这个事件不一定需要触发任务切换，可以是同步、异步或者数据事件。确定待重放事件后，多核日志分析模块将相应信息托管给对应的CPU管理模块，并按需更新任务集管理模块中相应的任务状态等信息。多核重放模块中的重放任务集管理模块和单核版本中拥有相同的功能。在多核重放架构中，CPU管理模块可能有多个，每一个对应目标应用的任务在执行阶段所引入的CPU。每一个CPU管理模块确保事件在自身CPU上被准确重现，所有的CPU管理模块确保所有的事件按照记录执行阶段的顺序被重现。当某个CPU管理模块完成了对应的事件重放，会通知多核日志分析模块继续分析后续的事件，以此类推完成整个重放过程。



图 23 多核版本重放模块设计框图

## 关键技术解决方案

### 探针效应的解决方案

探针效应是本文为了记录任务执行的动态信息而在操作系统源代码中插入记录代码而引入的。本文之前详细阐述了插桩效应对程序执行的影响，为了保证任务执行不会因为插桩效应而被影响，本文采用两种方式来解决所引入的探针效应。

1. 设计记录模块时考虑可能引入的探针效应，并尽量最小化对应的探针效应
2. 优化后信息记录模块的代码一直驻留在系统内

在ACoreOs操作系统中，内存也被划分为堆栈空间，其中堆空间也是供任务执行中动态分配（malloc），然而由于ACoreOs是一个多核多任务操作系统，从堆上分配内存过程中需要获取为了保存内存分配操作的原子性而引入的互斥信号量，从而任务从堆中分配内存会产生信号量事件。该特性对信息记录模块的设计产生了约束，如果信息记录模块提供的探针中调用了malloc函数而动态分配内存，将对调用该探针的任务产生较大的探针效应，没有插入该探针时，任务执行过程中并不会触发探针的执行，从而也不会产生相应的事件，同时，malloc操作也可能由于暂时申请不到内存资源而导致当前正在执行的任务挂起，从而触发了任务的一次额外切换操作，信息记录模块设计中必须回避这个探针效应。在本文进行探针设计时，对记录的数据组织借助任务的栈空间来实现，代替了对系统中堆空间的操作，有效避免了对应的探针效应。

对于信息记录模块，还存在多核互相竞争记录信息的问题。一般的，资源竞争可通过互斥锁完美的解决，但引入互斥锁后探针效应过大，对任务的原本执行流程会造成较大的影响。如图 24所示，如果没有引入互斥锁，两个CPU核心上的任务A和任务B执行流程如图 24左图所示，在执行过程中并没有发生切换，但一旦引入了互斥锁，其执行流程如图 24右图所示，任务的执行变化过大，信息记录模块如果采用互斥信号量解决资源竞争会主动的改变任务原本的执行路径，并且这种改变是非常暴力的，同时频繁的导致任务切换也会严重影响系统的性能，无法满足系统的实时性要求。同时，在对中断进行记录时，由于记录探针插入在中断处理函数中，而对于实时系统是不允许在中断中进行信号量的获取操作的，所以必须采用别的手段来代替信号量实现资源的互斥访问。在本文设计记录器时，采用了内核常使用的自旋锁机制来解决对缓冲区的互斥访问问题。对于信息记录模块采用较复杂的分离技术，同时引入自旋锁来代替互斥信号量，具体参见本文3.2.2一节。



图 24 互斥信号量对多核任务执行影响

### 事件发生位置标记的解决方案

前文中提到，任务执行中PC指针无法唯一的用来作为某些任务当前执行位置的标识。例如，给定如图 25所示的程序，PC数值为0X08048333对应的代码包含在循环体中，由于该循环可能在任务执行过程中，因count数值大于1循环体被执行多次，导致数值为0X08048333的PC会多次出现，从而出现PC指针无法唯一用来标记任务当前执行位置。必须引入额外的信息来辅助PC在任务执行周期内唯一标识执行当前的位置。目前已有的研究方案采用指令计数器作为辅助信息，其中一部分方案采用软件方式对每条执行的指令进行计数，此类方案开销较大；另一些方案借助硬件计数器。采用软件实现优点是可移植性好，无需额外的硬件辅助，但由于需要对执行的每条指令都进行计数，对程序的执行速度影响较大，其副作用也不可忽视。采用硬件辅助方式，对程序的指令执行没有任何干扰，但需要特殊的硬件支持。

|  |  |
| --- | --- |
| 16 | for (*int* i = 0; i < count; ++i) { |
| 17 | value = aa \* i; /\* PC = 0x08048333 \*/ |
| 18 | value += count; |
| 19 | } |

图 25 循环体中的PC指针无法唯一标识程序执行的当前位置

本文执行重放是基于PowerPC架构的处理器，没有硬件辅助方式所需的特殊硬件支持，采用软件方式又会对程序执行的执行速度影响过大，因此本文提出了使用任务执行阶段的上下文信息辅助PC指针来唯一标识任务当前执行位置。这里所需的上下文信息对于同步事件和异步事件是不同的。

对于同步事件（信号量获取、释放等），由于这类事件都是由于任务执行过程中显式的调用了操作系统接口产生的，而这些接口在任务重放执行阶段仍然再次会被调用，因此仅仅需要在产生同步事件时记录其被调用的次数就能够精确指出每一个同步事件产生的具体位置，进而辅助后续分析出导致任务切换的同步事件。

对于异步事件（本文主要指中断），任务在执行过程中不像同步事件显式的在执行到某些位置产生，而是随机发生在任务执行生命周期中的任何可发生的位置（中断被使能）。由于异步事件发生位置的不确定性，需要更多的辅助信息。本文提出采用任务执行时的上下文以及任务堆栈信息作为辅助信息，辅助PC指针唯一标识异步事件发生的位置。对于堆栈空间较大，对整个堆栈空间进行校验和的计算是很花费时间的，而对于常用的应用程序而言，任务执行过程中，其实不用整个堆栈内容就足以配合寄存器组用于对任务当前位置的唯一标示，本文中采用靠近堆栈栈顶的部分任务栈内容进行MD5的校验后与寄存器组一起用于唯一标示异步事件发生的位置。

### 上下文特殊性的解决方案

对于ACoreOs操作系统，任务的上下文仅在任务切换时，操作系统才会为其保存对应的上下文，本文前面阐述了该上下文对指导重放阶段的执行毫无意义，本文必须为任务在某些场合主动的产生上下文，并不是延后到Thread\_dispatch中由系统来帮助生成上下文。

在ACoreOs操作系统中，可触发任务切换的操作分为两大类：同步操作和异步事件，如图 26所示。本文主要通过在同步操作和异步事件发生时主动为相应的任务产生上下文，使用该上下文来直到后续重放执行阶段。



图 26 触发任务切换因素

同步操作主要指供应用程序显式调用的接口，这些接口在完成对应的操作后，会主动检测目前系统是否需要重新调取，如果存在该请求，通过调用Thread\_dispatch来进行具体的任务切换操作。为了减少记录执行阶段的开销，考虑到同步操作自身特点（在代码中显式的被应用程序调用），本文在任务控制块（Task Control Block）中引入了同步操作的上下文（同步操作计数器），在同步操作发生时所需要记录的任务上下文仅仅是该任务对应的同步操作计数器。

异步事件是指随机发生在任务执行中可发生的任何位置，本文中主要指中断。ACoreOs操作系统中，在响应中断时，需要判断是发生了中断嵌套还是打断一个执行体（任务）。对于中断嵌套，由于打断的是一个低优先级的中断，不会对任务造成任何影响，故对于这种类型的中断，本文将不会关注。如果中断打断任务的执行，需要为被打断的任务产生对应的上下文，根据2.4.2一节，上下文必须能够唯一标识该中断发生的位置。

### 事件发生顺序的解决方案

为了正确解决同步事件与同步事件之间、同步事件和异步事件之间、异步事件和异步事件之间发生的并行性对程序执行造成的不确定性影响。本文通过两大模块设计协同来解决：记录执行阶段的记录模块和重放执行阶段的重放模块。

首先，在记录执行阶段，通过记录模块来串行化并行发生的事件。在1978年，Lamport[[54](#_ENREF_54)]提出了一种方法用来获取在分布式系统上发生的事件的整个顺序，Lamport的论文主要解决分布式系统中缺少共有的时基问题。例如，各自拥有本地处理器时钟的两个处理器系统（处理器A和处理器B），由于完美的实现这两个本地时钟的同步几乎是不可能的，这两个时钟将会有一个微小的精度偏差，记做，这样，对发生在处理器A上本地时刻是的事件和发生在处理器B上本地时刻是的事件 排序将会非常困难。在Lamport的论文中，使用逻辑时钟来代替物理时钟，基于每个节点上的逻辑时钟采用分布式算法来解决这个问题。

尽管本文解决的不是分布式系统上应用程序的重放，然而本文重放平台是多核环境，存在与分布式系统上相同的问题：多个CPU之间不存在统一的时基，需要获取多个核上发生的事件的相对顺序。Lamport虽然给出了正确可用的算法来解决分布式多个任务之间发生事件的顺序问题，但是其方案不适用于本文的平台环境，主要原因是：（1）本文环境不同于分布式系统中不存在一个“全局”的实体，而是基于内存共享的多核系统；（2）本文环境中，不仅仅任务之间的同步事件操作会对任务执行造成干扰，异步事件发生也会对任务造成干扰，而在Lamport中没有对不存在交互的任务产生的事件进行排序处理。因此必须给出一种切实可行的方法来确定多核实时系统上事件发生的顺序。

本文中，对于同步事件与同步事件之间的并行性，由于操作系统在实现时，保证了仅有一个CPU核心在某一时刻可以获取到相应资源（例如获取信号量操作），即操作系统对这种类型的并行进行了串行化操作，因此记录模块仅需要记录哪一个CPU核心先成功获取了该资源，哪一个后获取即可。而对于同步事件和异步事件之间以及异步事件和异步事件之间的并行性，操作系统并没有提供机制来进行串行，本文采用多个CPU核心共享同一个逻辑时钟来解决，所有的CPU在事件产生时，都会给该事件打上逻辑时钟时间戳，用于标示该事件发生时的时刻，并对全局逻辑时钟进行递增操作，进而时间戳的大小指示了事件发生的相对顺序。由于多个CPU核心在对产生的事件进行记录时，可并行访问全局逻辑时钟，为了保证访问的一致性，引入了锁的机制。

其次在重放执行阶段，保证事件按照记录执行阶段的顺序被正确重放。在执行重放阶段，为了保证与记录执行阶段事件发生顺序的一致性，重放模块每次仅仅会启动一个处理器运行，待该处理器运行到了对应事件发生位置，如果记录得到的日志指示后续的事件是发生在其他处理器上，那么重放模块会暂停当前处理器的运行，而调度进入下一个事件发生的处理器进行运行。

### 系统特殊性解决方案

由于具体的目标应用程序需要运行在目标机上，并且在记录执行阶段，程序执行中产生的信息也位于目标机，然而，目标机资源有限，无法缓冲整个日志数据，必须将日志上传到宿主机端，这主要通过信息记录模块来解决，信息记录模块会管理一段内存缓冲区，缓冲记录得到的执行信息，并动态的在适当时机将日志上传到宿主机，具体参见3.2。在重放阶段，目标程序也必须在目标机上被重放模块创建和控制，而重放模块运行在宿主机上，本文中采用调试服务提供的调试接口来完成对目标机上任务的创建和控制。在程序重放过程中，重放模块也需要动态从目标机获取任务相关信息，需要通过调试服务提供的接口直接调用目标系统接口来完成，此时调试服务表现的不是调试功能，而仅仅是一个管道，提供了调用目标机上系统接口的能力。

## 本章小结

本章介绍了多核实时操作系统ACoreOs上应用程序执行重放方法的设计细节。在本节中，先给出了本文提出的执行重放方法的整体框架，其包含三个模块：信息记录模块、日志分析过滤模块和重放模块，接着本文给出了各个模块的具体设计思路，最后详细阐述了针对重放方案中的各个挑战和问题提出的相应解决方案。

# 执行重放方案的实现

## 信息记录模块实现

本文前面部分已经给出了信息记录模块所必须完成的工作以及大致的设计思路，下面将具体介绍信息记录模块的实现。

信息记录模块为记录任务执行中的动态信息，为不同的插桩位置提供了相应的探针以完成对相关信息的记录，然而为了提高探针记录信息的速度，减少探针执行时对系统造成的影响，本文实现了探针记录数据时，其调用的信息记录接口具有如下三个特点：

1. 高效快速的支持多核日志写入
2. 动态的缓冲区伸缩管理
3. 实时上传缓冲区日志信息到宿主机端

该记录信息接口以写入控制、缓冲区管理、日志上传三部分相分离并引入环形缓冲区的思想，成功解决探针记录信息时对系统影响过大的问题。下面就详细介绍各个部分详细的实现以及之间的协作。

信息记录模块具体各个部分之间的整体架构如图 27所示。信息记录模块对各个插桩探针提供了信息记录接口LES\_EventLogHandle，该接口支持任意长度日志写入，而不会触发任务切换或者产生新的事件。探针被执行时，组织对应的程序执行信息并通过LES\_Eventloghandle完成对信息的记录。在信息记录模块内部，缓冲区由环形缓冲区来实现，缓冲区管理模块通过运行在目标系统中的高优先级任务实现，日志上传模块也是通过高优先级任务来实现。

LES\_EventLogHandle接口在写入数据时对环形缓冲区当前状态进行相关检查，如果发现缓冲区当前状态未达到要求的状态（例如空闲空间不充足），将以伪消息的方式通知缓冲区管理任务进行相应操作，环形缓冲区接收到操作命令后执行相应的动作，在缓冲区中保存数据超过设定阈值采用信号量机制唤醒日志上传任务，开始日志上传。

信息记录模块必须经过开启操作才能够提供信息记录的功能，其开启操作流程如图 28所示。主要包括初始化环形缓冲区、相关的数据结构，启动所需的任务以及使能探针。只有使能了信息记录模块，探针在执行时才能调用LES\_EventLogHandle向缓冲区中写入收集到的数据，进而任务的执行信息会源源不断的被信息记录模块捕获直到关闭信息记录模块。

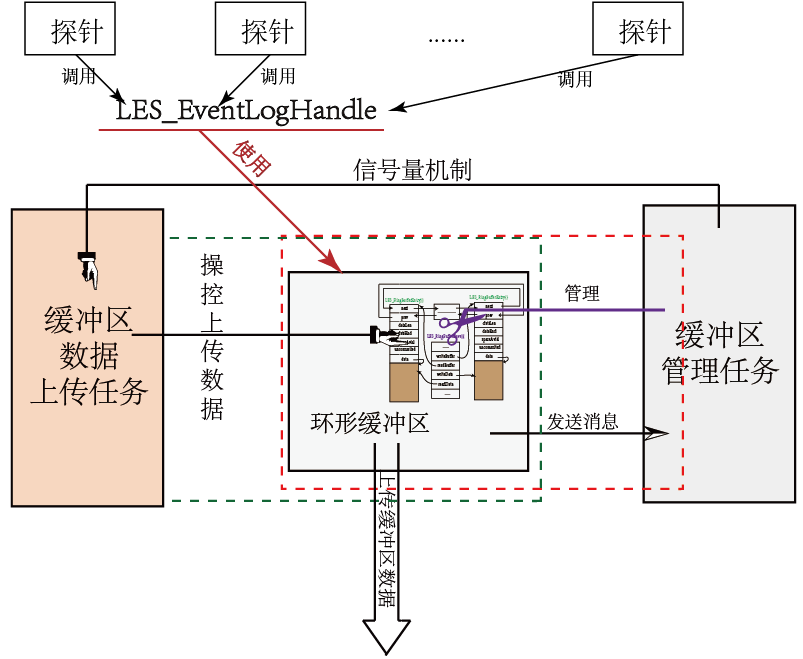


图 27 信息记录模块实现框图



图 28 信息记录模块开启执行流程

前面，本文主要介绍了信息记录模块整体的实现和工作流程，下面，本文将介绍信息记录模块内各模块详细的实现细节。

### 探针实现

信息记录模块主要将提供的探针插入到操作系统执行路径中，进而在系统运行中，通过探针的执行动态记录系统中任务的执行信息。下面将具体分析不同的事件信息所需要插入的探针接口。

1. 任务创建事件信息记录探针

插桩在系统创建任务接口中，在创建任务的时候被调用，以记录任务被创建时任务具体信息，如表 3所示。记录的信息包括任务创建后系统标识该任务唯一的标识符、创建任务时指定的任务名、任务的优先级、任务的额外选项（是否启用时间片轮转、是否允许被抢占等）、任务创建时使用的栈空间大小、任务栈空间起始地址。

表 3 任务创建事件信息记录探针

|  |  |
| --- | --- |
| 接口原型 | 参数说明 |
| LES\_TaskCreateLog(taskId, taskName,taskPriority, options, stackSize,stackBase) | taskId: 任务的Id  taskName: 任务的名称  taskPriority: 任务的优先级  options：任务可选项  stackSize: 任务的栈大小  stackBase: 任务栈基地址 |

1. 任务启动事件信息记录探针

插桩在系统启动任务接口中，在任务启动时被调用，以记录任务启动时的相关信息，如表 4所示。记录的信息包括任务的标识符、任务的入口地址、任务入口参数。

表 4 任务启动事件信息记录探针

|  |  |
| --- | --- |
| 接口原型 | 参数说明 |
| LES\_TaskStartLog(taskId, entryPoint, args) | entryPoint: 任务入口地址  argNums: 任务参数个数  args: 具体的参数 |

1. 信号量事件信息记录探针

插桩在信号量操作接口中，在任务调用信号量接口时被调用，用于记录任务调用信号量操作时具体信息，如表 5所示。记录的信息包括任务的标识符、信号量的标识、以及隐藏在该接口中的获取任务相关上下文的操作。这个操作收集同步事件产生的上下文，主要是同步事件计数器。

表 5 信号量操作信息记录探针

|  |  |
| --- | --- |
| 接口原型 | 参数说明 |
| LES\_SemaphoreTakeLog(semId, taskId, taskTcb) | semId: 信号量的Id值  taskId：调用semTake的任务Id  taskTcb: 调用semTake函数的任务TCB基地址 |
| LES\_SemaphoreGiveLog(semId, taskId, taskTcb) | semId: 信号量的Id值  taskId：调用semGive的任务Id  taskTcb: 调用semGive函数的任务TCB基地址 |

1. 消息队列事件信息记录探针

插桩在消息队列操作接口中，在任务调用消息队列接口时被调用，用于记录任务操作消息队列产生的具体信息，如表 6所示。记录的信息包括任务的标识符、消息队列的标识、发送消息内容、发送消息长度、发送消息附加的选项以及隐藏在该接口中的获取任务相关上下文的操作。这个操作记录同步事件产生的上下文，主要是同步事件计数器。

表 6 消息队列事件信息记录探针

|  |  |
| --- | --- |
| 接口原型 | 参数说明 |
| LES\_MessageQueueSendLog(msgQId, msgStartAddr, msgLen, priority,taskId, taskTcb) | msgQId: 消息队列的Id值  msgStartAddr：消息信息的起始地址  msgLen: 消息的长度  priority: 消息的优先级  taskId: 调用该函数的任务Id  taskTcb: 调用msgQSend函数的任务TCB基地址 |

|  |  |
| --- | --- |
| 表 6 消息队列事件信息记录探针（续） | |
| LES\_MessageQueueRecvLog(msgQId, buffer, bufLen, taskId, taskTcb) | msgQId: 消息队列的Id值  buffer：缓冲区的起始地址  bufLen: 缓冲区的长度  taskId: 调用该函数的任务Id  taskTcb: 调用msgQRecv函数的任务TCB基地址 |

1. 任务上下文切换事件信息记录探针

插桩在操作系统重新调度函数中，用于记录任务的切换信息，如表 7所示。记录的信息包括任务的标识，该接口并不会收集任务的上下文信息。在该接口内部会对将要被切换出去的任务TCB中的同步事件计数器清零。

表 7 上下文切换事件信息记录探针

|  |  |
| --- | --- |
| 接口原型 | 参数说明 |
| LES\_ContextSwitchLog(taskId, taskTcb) | taskId: 任务的Id  taskTcb：任务控制块地址 |
| LES\_FakeContextSwitchLog(taskId, taskTcb) | taskId: 任务Id  taskTcb: 任务TCB地址 |

1. 调度指示信息记录探针

插桩在同步操作或者中断返回完成之后重新使能内核调度的接口中，用于辅助指示后续的调度产生的因素，如表 8所示。LES\_SyncRescheduleLog在同步操作结束后使能调度时被调用，记录的信息包括任务的标识符、当前同步事件嵌套的深入以及是否该同步操作结束后是否需要重新调度任务。LES\_InterruptExitRescheduleLog在中断操作后使能调度时被调用，记录的信息仅包含任务的标识符。

表 8 调度指示信息记录探针

|  |  |  |
| --- | --- | --- |
| 接口原型 | | 参数说明 |
| LES\_SyncRescheduleLog(taskId, nestLevel, needReschedule) | taskId: 任务的Id  nestLevel：同步调用嵌套深度  needReschedule：表明后续是否需要调度 | |
| LES\_InterruptExitRescheduleLog(taskId) | taskId: 任务Id | |

1. 中断打断任务上下文信息记录探针

插入在中断打断上下文处理路径中，用于指示中断打断任务上下文事件发生，如表 9所示。记录中断发生时，被打断的任务上下文信息，包括被打断任务的标识、任务寄存器上下文以及堆栈校验和。

表 9 中断打断任务上下文信息记录探针

|  |  |
| --- | --- |
| 接口原型 | 参数说明 |
| LES\_InterruptEnterContextLog(taskId, contextAddr, stackSum) | taskId：任务Id  contextAddr：任务上下文  stackSum: 堆栈校验和 |

1. 数据流信息记录探针

插入在一些执行结果具有不确定性的函数执行体中，用来收集函数执行的结果，如表 10所示。记录的信息包括存放数据的地址，具体的数据内容、长度以及隐藏在该接口中的获取任务相关上下文的操作。这个操作记录数据流事件产生的上下文，仅仅包含程序执行的当前位置PC数值。

表 10 数据流信息记录探针

|  |  |
| --- | --- |
| 接口原型 | 参数说明 |
| LES\_UserDataLog(dataAddr, dataSize) | dataAddr: 数据地址  dataSize: 数据长度 |

1. CPU索引信息记录探针

放置在每一个事件记录之前，在内部记录模块维护的CPU索引发生变更时，被触发记录，指示后续事件是发生在该CPU上的，具体探针接口如表 11所示。记录的信息仅包含当前CPU编号，同时也会用当前的CPU编号更新系统内部为了减少记录开销设置的当前CPU索引值。

表 11 CPU索引信息记录探针

|  |  |
| --- | --- |
| 接口原型 | 参数说明 |
| LES\_CpuIndexLog(cpuIndex) | cpuIndex: 当前CPU编号 |

以上这些探针被插入在系统执行的不同路径上，在被执行时将程序的动态执行信息组织成对应事件的数据结构，调用写入控制模块接口记录到缓冲区中，以完成探针对信息的记录工作。

### 探针插桩实例

上面，本文介绍了信息记录模块提供的探针以及每个探针所记录的详细信息，下面本文将详细介绍插桩过程，由于每种类型事件的插桩过程存在相似性，本文下面主要拿同步操作、异步操作以及数据流的插桩来介绍探针的插桩过程。

1. 同步操作插桩实例

同步操作在ACoreOs系统提供的接口中所占的比重较大，为了阐明对同步操作插桩的过程而不显得冗余，本文选取了信号量获取操作接口（ACoreOs\_semaphore\_obtain），ACoreOs\_semaphore\_obtain负责获取指定的信号量，在信号量有效时继续当前任务执行，否则阻塞当前任务运行。

ACoreOs\_semaphore\_obtain接口的信息插桩实例如图 29所示，图中灰色的部分是所需要插入的信息记录模块的探针，其在整个执行流程中的位置指示了插入到ACoreOs\_semaphore\_obtain函数实现体的位置。ACoreOs\_semaphore\_obtain接口在未插桩之前，在禁用内核调度之后，直接进行锁状态的判断，并依据当前的锁状态指示当前任务的执行，并适当更新锁状态，在执行这些操作后直接开启内核调度功能完成整个信号量获取操作。ACoreOs\_semaphore\_obtain接口插桩后，在禁用内核调度功能后，需要收集目前信号量和当前任务的相关信息，其中最重要的信息是同步事件计数器，该信息将作为重放模块在命中切换点时进行比对确定切换位置的依据，在开启内核调度功能后，由于系统会检测当前是否需要重新调度，如果发生调度，需要采用插桩以记录信息以指示后续的调度是同步操作触发的。

1. 异步操作插桩实例

本文中的异步操作主要指中断，所以此处用中断处理作为异步插桩实例进行详细讲解相关接口的使用。中断处理的插桩实例如图 30所示，图中灰色部分指示了插桩接口和插桩位置。中断处理中涉及到中断的嵌套，中断栈的切换。在ACoreOs操作系统中，中断处理采用的是专用栈，在未发生中断嵌套时，表明此次中断打断的上下文是某个任务的上下文，信息记录模块采用LES\_InterruptEnterContextLog来获取打断的任务详细信息，由于打断的是任务上下文，在调用具体的中断处理函数前，需要切换当前栈为中断专用栈，待中断服务函数执行完成后，进行中断相应上下文的恢复，并在中断处理完成后采用LES\_InterruptExitRescheduleLog插桩来记录中断退出上下文事件信息以指示后续发生的任务切换与中断有关。



图 29 ACoreOs\_semaphore\_obtain接口探针插桩比对实例

1. 数据流插桩

数据流插桩主要收集程序执行过程中从外界获取的不确定输入数据，此处选择对rdtsc()函数的实现进行插桩作为实例进行说明。如图 31所示，图中灰色区域是插桩引入的代码，用于收集系统源生rdtsc()函数执行得到的结果,收集的信息包括存放结果的地址以及数据对应的长度，为了替换应用程序对于rdtsc函数调用为调用插桩版本LES\_rdtsc()函数，需要借助C语言宏定义，定义rdtsc为LES\_rdtsc，之后应用程序对于rdtsc()的调用其实就是调用插桩版本LES\_rdtsc。



图 30 中断信息插桩探针对比实例

|  |  |
| --- | --- |
| 19 | UINT64 LES\_rdtsc(){ |
| 20 | UINT64 ret = rdtsc(); |
| 21 | LES\_UserDataLog(&ret, sizeof(ret)); |
| 22 | return ret; |
| 23 | } |

图 31 rdtsc()函数插桩实现

### 环形缓冲区实现

环形缓冲区用来解决目标机内存不足，无法将记录的日志全部保存到内存中，而记录信息又无法直接在每次记录的时候就直接上传到宿主机上的问题，环形缓冲区充当一个日志暂存区，其结构如图 32所示，是由多个缓冲区块通过双向指针构成的一个环形链式结构的缓冲区。

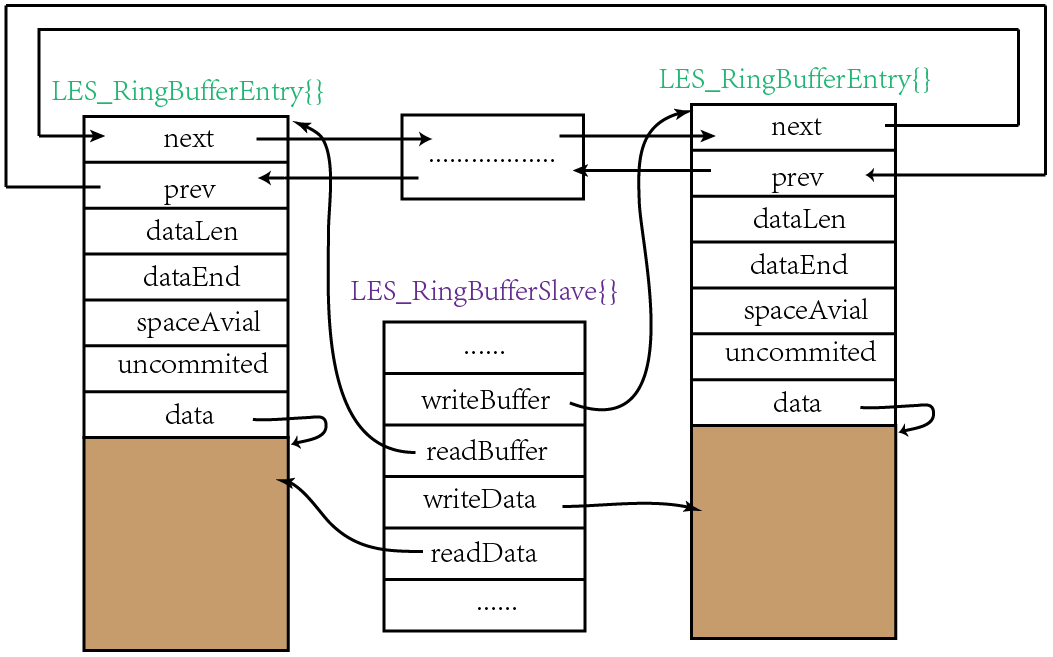


图 32 环形缓冲区结构图

每个缓冲区块（LES\_RingBufferEntry结构）都包含管理自身的信息，例如已经使用的空间，剩余可写入的空间，保存的数据是否有效等。整个环形缓冲区状态指示结构（LES\_RingBufferSlave）用来反映整个环形缓冲区的状态信息，包括下次写入的位置、下次写入的缓冲区块、下次读取的位置、下次读取的缓冲区块以及缓冲区中可读取的数据等。

缓冲区块大小和个数都是可手动配置的。使用前可配置最少的缓冲区块个数(MIN\_NUMS)和最多的缓冲区块个数(MAX\_NUMS)以及缓冲区块的大小(ENTRY\_VOL)。在启动记录模块的时候，根据给定的配置分配MIN\_NUMS个缓冲区块，主要是保证一旦启用了日志记录，现在环形缓冲区中的空闲空间是可以满足临时需求的，而MAX\_NUMS主要用来回收空闲的缓冲区块，保证记录器给系统带来的内存压力会随着记录信息的多少动态调整，不会造成记录信息频率较低时仍占据系统较多的内存而影响系统其它任务的执行。

### 写入控制模块实现

写入控制模块对外提供LES\_EventLogHandle接口，主要解决日志信息如何被记录到环形缓冲区中，其执行流程如图 33所示。

写入阶段仅仅负责从环形缓冲区中分割一块可用的空闲区域，并将数据复制到分配的空间中，以减少写入阶段带来的延时，同时由于在多核平台下，存在多个核并行向环形缓冲区中写入数据的操作，为了保证写入数据的一致性，实现时引入自旋锁来解决多核之间的这种竞争，为了尽可能缩短自旋锁临界区执行的时间，自旋锁临界区中仅仅负责从环形缓冲区中分配空间，而具体数据的写入是在无锁的环境下进行的。



图 33 写入控制模块执行流程图

待日志信息写入到缓冲区后，判断当前缓冲区的状态，并依据其状态给缓冲区管理模板发送相应的命令消息，这些命令消息包括：

1. LES\_MESSAGE\_ALLOC，该消息表示目前环形缓冲区的空闲空间过小，需要分配更多的空间。该消息是一个紧急消息，处理方案有些不同；
2. LES\_MESSAGE\_THRES，该消息表示目前环形缓冲区中的已经存放的数据超过了上传指定的阈值。现在需要进行数据的上传操作，尽可能快的将占用的缓冲区释放出来；
3. LES\_MESSAGE\_FREE，跟LES\_MESSAGE\_ALLOC类似，区别是目前缓冲区的空闲缓冲区块个数过多，可以进行释放一部分缓冲区块；
4. LES\_MESSAGE\_ERROR，该消息主要发生在发出多个LES\_MESSAGE\_ALLOC，但是缓冲区的一直得不到满足或者写入的速度太快，没有空闲缓冲区可用了，这个消息几乎不会让其产生，会有一些措施来保证。

这些消息通过伪消息队列的方式传递给缓冲区管理任务，指导其维护管理缓冲区。

### 伪消息队列实现

伪消息队列是写入控制模块和缓冲区管理模块之间通信的桥梁，其设计框架如图 34所示。伪消息队列设计不同于操作系统提供的标准的消息队列，该消息队列使用循环队列来保存消息，为发送方维护一个写位置，为接收方维护一个读位置；读写位置的变更并不是采用锁机制来保护的，而是借助不会产生任务切换以及新事件的原子操作来完成的。而非借助操作系统提供的具有原子操作的消息队列来完成的。这里没有采用操作系统的消息机制，原因在于操作系统提供的消息机制会迫使信息记录模块在记录系统执行信息时引入了新的事件（消息队列操作事件），而该事件在没有使能信息记录模块时却不会产生；再者，系统提供的消息机制可能会由于目前消息队列的状态满足不了要求而阻塞任务的执行，进而触发任务切换，这都造成了由于使能信息记录模块而严重影响了原先任务执行的路径和流程。

伪消息队列中，并不严格保证发送方发送的消息一定被接收方收到，只需要保证紧急消息（例如缓冲区空间不充足消息）被接收方收到即可。

对于紧急消息，一旦发送方发送了该消息，但该消息还没有被接收方处理，后续的非紧急消息是不会真正被发送出去的，这种设计主要是防止紧急消息被发送方发送的非紧急消息覆盖。例如，缓冲区管理任务刚好处理完成所有的消息后进入睡眠，等待下次被操作系统调度运行，此时写入控制记录模块检测出空闲空间不充足，通过伪消息队列向缓冲区管理任务发送一条LES\_MESSAGE\_ALLOC紧急消息，然而由于后续发现缓冲区中的数据已经超过了阈值，会不断的发送LES\_MESSAGE\_THRES消息，由于本设计中维护的是一个环形消息队列，很快就会覆盖之前发送的LES\_MESSAGE\_ALLOC消息，这样就在缓冲区管理任务再次运行的时候不能立即处理这个紧急的消息，从而导致写入控制模块由于检测没有空闲空间而发送致命消息，导致出错。



图 34 伪消息队列设计框图

### 缓冲区管理模块实现

缓冲区管理模块负责动态根据缓冲区的状态信息维护缓冲区，其作为一个高优先级的守护任务运行在目标系统中，执行流程如图 35所示。



图 35 缓冲区管理模块执行流程图

由于环形缓冲区管理任务是一个拥有较高优先级的任务，为了保证该任务执行不会对系统中其它较低优先级任务执行造成过大影响，该任务会周期性的睡眠，以保证系统中较低优先级的任务可获取到CPU控制权得到运行。环形缓冲区管理任务周期性的被唤醒，每次被唤醒执行时，首先检测消息队列中是否有需要处理的消息，如果没有消息，表明目前缓冲区状态是正常的，不需要进行任何的维护操作，该任务将自动进入睡眠，等待下一次被唤醒执行。如果发现有消息等待处理，会逐条进行消息的读取和处理，直到消息队列中没有未处理的消息。

缓冲区管理任务可接收到的消息以及对应处理过程如表 12所示。其中LES\_MESSAGE\_ALLOC是紧急类型的消息，

表 12 缓冲区管理任务对各个消息处理过程

|  |  |
| --- | --- |
| 消息类型 | 处理操作 |
| LES\_MESSAGE\_ALLOC | 向系统申请一块空闲缓冲区块并加入到环形缓冲区中 |
| LES\_MESSAGE\_FREE | 将指定的缓冲区块从环形缓冲区中去掉并归还给系统 |
| LES\_MESSAGE\_THRES | 以信号量机制通知日志上传日志可以上传日志 |
| LES\_MESSAGE\_ERROR | 调用预先设定的错误处理函数 |

### 日志上传模块实现

日志上传模块负责在适当的时机（数据量超过阈值或者所等待的信号量有效）操作缓冲区，上传保存在缓冲区中的数据到宿主机端，与缓冲区管理模块类似，日志上传模块也作为一个高优先级守护任务运行在目标系统中，执行流程如图 36所示。

日志上传守护任务在缓冲区中的日志量超过设定的阈值时，将被告知，紧接着启动日志上传操作完成对环形缓冲区数据的上传工作。由于日志上传模块的另外一个职责是尽可能快的及时的排空环形缓冲区中累积的日志信息，所以日志上传守护任务的优先级非常高，否则可能出现记录的速度一直大于上传的速度，导致应用程序执行不久就会因为缓冲区没有空闲空间而被迫停止记录。为了比较及时的告知该守护任务，设计中引入了信号量机制，一旦缓冲区管理任务接收到阈值消息，会立刻通过信号量通知日志上传守护任务，日志上传守护任务将抢占低优先级的任务立刻得到运行，尽快上传累积在缓冲区中的日志。

日志上传守护任务在上传日志阶段，需要操作环形缓冲区，由于在此间执行其他程序的处理器核心会向缓冲区写入数据，为了解决多核同时操作同一个环形缓冲区带来的一致性问题，需要借助记录阶段引入的自旋锁，本设计中，为了最小化读写同一个环形缓冲区带来的性能影响，利用环形缓冲区中每个缓冲区的大小来解决。在环形缓冲区一节中提到的读写指针，在设计时会保证这两个指针不会指向同一个缓冲区块，这样在上传和写入时，并不需要自旋缓冲区，而仅仅在缓冲区块间移动读写指针才需对环形缓冲区进行自旋操作。当上传日值数据超过设定的阈值后，会重现检测当前环形缓冲区剩余的待上传的数据是否仍然大于阈值，如果仍然有大量的数据等待上传，上传守护任务继续进行日志上传，直到日志数据量没有超过阈值并且此时缓冲区管理任务并没有通知自己上传数据时会被阻塞，等待下次唤醒。



图 36 日志上传模块执行流程图

## 日志解析过滤模块实现

日志解析过滤模块包含两部分内容：日志解析和日志过滤。日志解析负责解析记录执行阶段保存的日志文件，日志过滤模块则根据解析结果从杂乱无章的日志信息中抽取出重放模块所必须的信息。

### 日志解析模块实现

日志解析模块（后文中也称日志解析器）负责将记录阶段上传到宿主机端的日志文件进行解析，并以对应数据结构格式组织到内存中。

为了尽可能减小记录日志的信息量，本设计中日志记录并没有采用序列化后的字符串形式，而是全部采用二进制数据进行存储。从而，在日志记录阶段需要指定每条日志信息的长度以及记录阶段和解析阶段相应数据结构必须完全相对应。特别的，有两方面需要考虑

1. 记录和解析由于采用的是不同的语言，需要弄清楚每个基本结构所占用的字节大小以及相互的对应关系。由于解析采用Java语言实现，因此需要确定目标平台和Java语言对基本数据类型表示的异同。表 13给出日志解析程序中用到的基本数据类型在目标系统和Java语言中的对应关系。

表 13 Java语言和目标系统数据结构对应关系

|  |  |  |  |
| --- | --- | --- | --- |
| 目标系统 | | Java | |
| 数据类型 | **所占字节** | **数据类型** | **所占字节** |
| Char | 1 | **Byte** | 1 |
| Short | 2 | **Short** | 2 |
| Int | 4 | **Int** | 4 |

1. 记录和解析程序运行在不同的平台上，记录器运行在目标系统上，目标系统是基于Power架构的处理器，而解析程序运行在主机端的Java虚拟机中，两者采用的平台不一致，从而在数据存储和表示方面可能采用不同的存储方式，本文的运行环境中，目标系统采用的是大端序存储，Java虚拟机也是按照大端序模式来处理数据的，两者采用的存储方式是相同的，中间不需要额外的转换。

日志解析模块工作流程如图 37所示，日志解析模块需要根据记录模块定义的事件唯一标识符先注册不同日志的唯一标识符和对应的解析动作，之后解析模块拥有解析包含相关事件类型的日志文件的能力，接着读取记录阶段保存的日志文件，按照每条日志格式先读取日志头，其唯一标识了该条日志的类型，并以此找寻注册这类日志的解析操作，从而通过找寻到的解析操作读取具体的事件所记录的数据，构造该条事件日志对应的数据结构，进而完成该条日志的解析操作，接着读取下一条日志的日志头，并以此类推，直到解析完成整个日志文件。解析模块采用ArrayList组织整个解析得到日志数据。解析模块具体实现如图 38所示，所有的具体事件日志结构都继承于抽象类BaseLogStructInfo，每个事件类型仅需要继承BaseLogStructInfo并提供额外的的数据成员和提供相应的实现了ILogOperation接口的解析类，并通过LogParse提供的注册接口根据该事件的唯一标识符注册进解析系统，解析系统就可以很好的解析具有这种类型事件的日志文件。解析模块的这种结构很好的应对在研究重放方案时需要动态增加或者减少对某个事件记录的变动。



图 37 日志解析模块工作流程

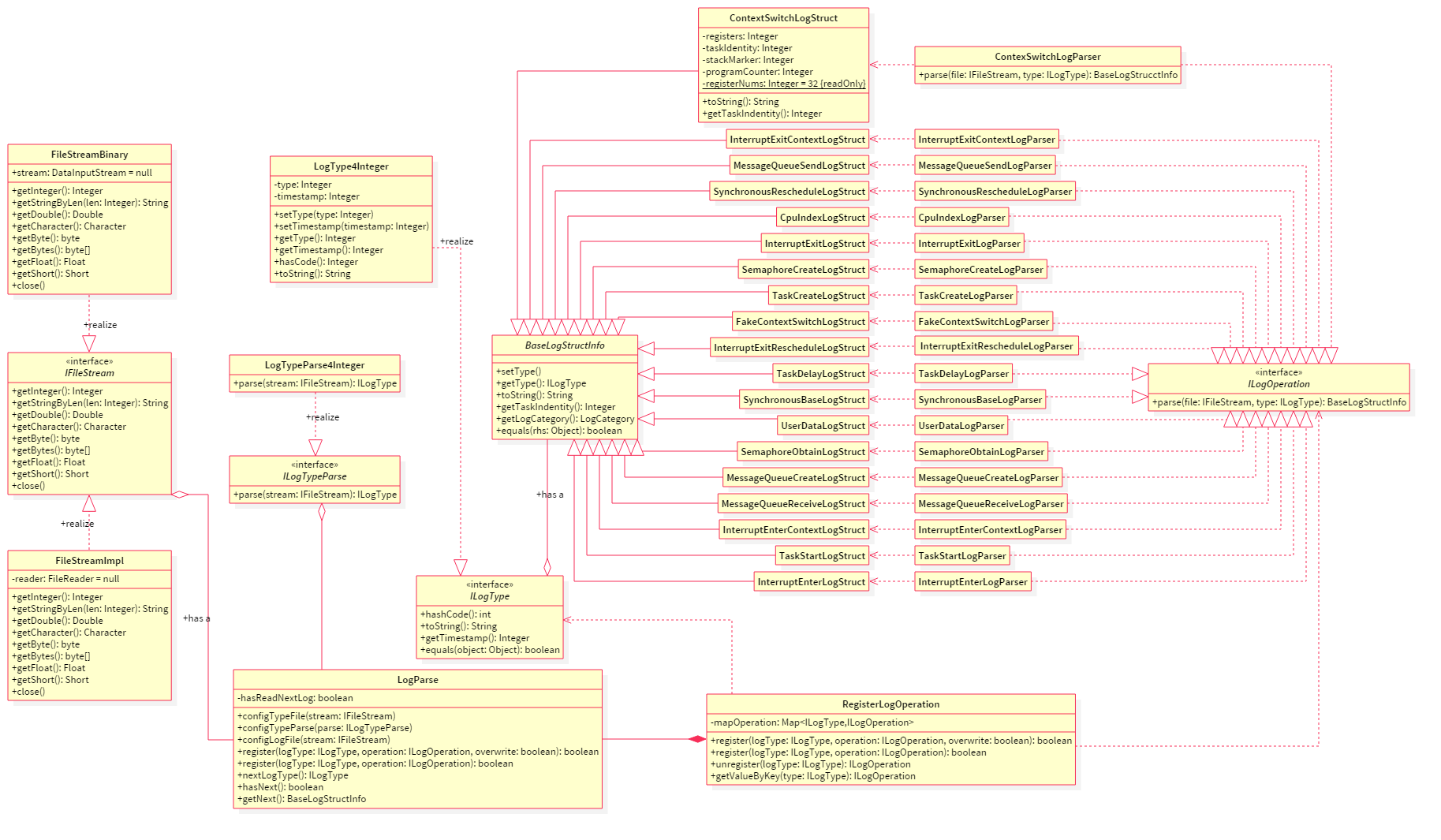


图 38 日志解析模块实现框图

### 日志过滤模块实现

日志过滤模块完成对记录阶段的事件信息过滤操作。其具体实现如图 39所示，过滤部分是通过多个过滤器协同完成，每一个单独的过滤器负责不同的过滤功能，例如LogFilterByCpuIndex过滤器根据指定的CPU索引数值过滤得到该CPU核心上发生的所有索引事件。日志过滤模块根据日志解析模块解析的结果分析目标应用的执行流程，过滤出对执行重放阶段有指导意义的事件和相关数据。

在本文中，由于应用程序不同次的执行中，系统为应用程序中任务分配的任务标识符可能不同，而且任务描述符和任务名称之间没有任何对应关系，因而在过滤阶段不能将任务的标识符直接作为标示目标应用程序中任务的有效手段，在本文中，采用任务名来作为任务唯一的标示，过滤阶段通过任务名来区分目标应用任务和系统自身创建的其他任务。

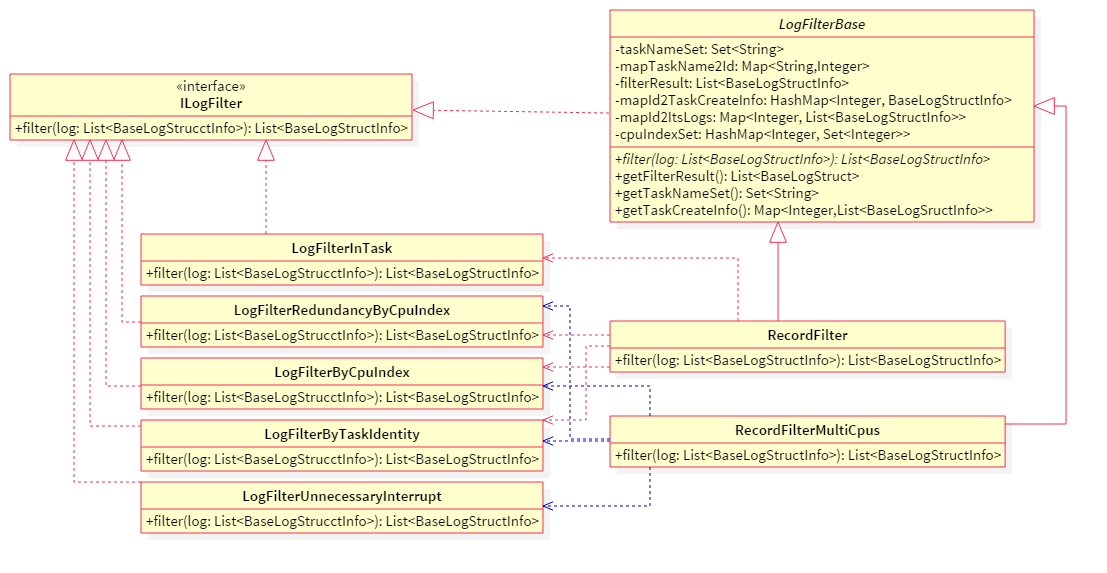


图 39 日志过滤模块实现

日志过滤模块在单核和多核情况下，表现出的过滤行为是不同的。

1. **单核版本过滤模块实现**

在单核版本中，其具体执行流程如图 40所示。过滤模块会首先根据任务名过滤出目标应用创建的任务运行时产生的事件信息，对于每个任务产生的事件信息，提取出该任务创建时的信息，这些信息被单独提取出来供执行重放初始化阶段设置重放任务的相关信息，其他的事件信息会通过任务执行分析过滤器，该过滤器分析任务执行阶段发生的事件嵌套，保留触发任务切换的原始事件，过滤剔除其他无关事件，保留所有的数据流事件；对每个任务日志过滤完成后，汇总CPU索引记录事件按照时间戳升序重新组合成在一起。最后，过滤掉里面不必要的CPU索引事件，最终得到的结果会作为重放模块的输入。



图 40 单核版本的日志过滤模块执行流程

过滤多余事件信息的算法伪代码如表 14所示，主要分析任务执行过程中的事件发生嵌套关系，找出保留最外层触发任务切换的事件。

表 14 过滤多余事件信息算法伪代码

|  |  |
| --- | --- |
| **Algorithm**: filter unnecessary log in task | |
| **Input**: task\_event\_log | |
| **Output**: filted\_log, create\_log | |
| 1 | for event\_log in task\_event\_log do |
| 2 | switch event\_log.type |
| 3 | case ContextSwitch:    task\_is\_running <- ! task\_is\_running |
| 4 | if ! task\_is\_running then |
| 5 | cause\_log <-getCause(cause\_by\_sync,cause\_by\_async) |
| 6 | filted\_logs.add(cause\_log) |
| 7 | cause\_by\_sync.init() cause\_by\_async.init() |
| 8 | in\_sync.init() in\_async.init() |
| 表 14 过滤多余事件信息算法伪代码（续） | |
| 9 | end if |
| 10 | task\_logs[task\_name].add(event\_log) |
| 11 | case SyncEventEnter:    in\_sync['nest'] ++ |
| 12 | update\_sync(cause\_by\_sync, in\_sync, in\_async) |
| 13 | case SyncEventExit:        in\_sync['nest'] -- |
| 14 | update\_sync\_flag(cause\_by\_sync, in\_sync) |
| 15 | case SyncCause:        cause\_by\_sync['flag'] <- true |
| 16 | case AsyncEventEnter:    in\_async['nest'] ++ |
| 17 | update\_async(cause\_by\_async, in\_sync, in\_async) |
| 18 | case AsyncEventExit:    in\_async['nest'] -- |
| 19 | update\_async\_flag(cause\_by\_async, in\_async) |
| 20 | case AsyncCause:        cause\_by\_async['flag'] <- true |
| 21 | case DataEvent:     filted\_log.add(event\_log) |
| 22 | case TaskCreate:     create\_log <- event\_log |
| 23 | end for |
| 24 | return filted\_log, create\_log |

日志合并算法伪代码描述如表 15所示，采用败者树思想对各个任务过滤之后的日志以及CPU事件信息按照时间戳升序合并。

表 15 日志合并算法伪代码

|  |  |
| --- | --- |
| **Algorithm**: sort the task log in ascending order | |
| **Input**: task\_log\_list, one\_cpu\_log | |
| **Output**: sorted\_log | |
| 1 | sorted\_log.add(one\_cpu\_log) |
| 2 | for  index in task\_log\_list.len do |
| 3 | task\_index[index] <- 0 |
| 4 | end for |
| 5 | while (has\_finish(task\_index)) do |
| 6 | min\_task <- getMin(task\_index, task\_log\_list) |
| 7 | sorted\_log.add( |
| 8 | task\_log\_list[min\_task][task\_index[min\_task]]) |
| 9 | task\_index[min\_task] ++ |
| 10 | end while |

1. **多核版本过滤模块实现**

在多核版本中，日志过滤模块执行流程如图 41所示。其大部分和单核处理版本是相同的，然而在对单个任务发生事件信息进行过滤时处理步骤是不同的。多核版本中，需要保留任务执行中产生的所有同步事件，分析任务之间事件发生的先后顺序，对于单个任务执行中产生的异步事件，分析是否触发任务后续切换，对于没有触发任务切换的异步事件直接过滤掉。



图 41 多核版本的日志过滤模块执行流程

任务产生的日志中多余异步事件过滤算法伪代码如表 16所示，采用类似于状态机方式来对嵌套在同步事件中的异步事件进行过滤，对于嵌套在中断处理中的同步事件也会进行过滤。

表 16 过滤多余异步事件算法伪代码

|  |  |
| --- | --- |
| **Algorithm**: filter unnecessary log cause by async event in task | |
| **Input**: task\_event\_log | |
| **Output**: filted\_log, create\_log | |
| 1 | for event\_log in task\_event\_log do |
| 2 | switch event\_log.type |
| 3 | case SyncEventEnter:    in\_sync['nest'] ++ |
| 表 16 过滤多余异步事件算法伪代码（续） | |
| 4 | if in\_async['nest'] not equals 0 then |
| 5 | continue |
| 6 | end if |
| 7 | filted\_log.add(event\_log) |
| 8 | case SyncEventExit:      in\_sync['nest'] -- |
| 9 | case AsyncEventEnter:    in\_async['nest'] ++ |
| 10 | cause\_by\_async['cause'] <- event\_log |
| 11 | case AsyncEventExit:    in\_async['nest'] -- |
| 12 | case AsyncCause: |
| 13 | if in\_sync['nest'] not equals 0 then |
| 14 | continue |
| 15 | end if |
| 16 | filted\_log.add(cause\_by\_async['cause']) |
| 17 | case TaskCreate:     create\_log <- event\_log |
| 18 | case ContextSwitch: filted\_log.add(event\_log)  init\_all() |
| 19 | end for |
| 20 | return filted\_log, create\_log |

对于日志归并，其算法伪代码与单核版本类似，不同的是多核版本下的有很多的CPU索引事件需要参与到归并中，而且归并完成后还需要进一步对多余的CPU索引事件过滤。

## 重放模块

执行重放根据日志过滤模块过滤得到的结果创建应用程序二次执行并在执行中动态控制任务执行步调和执行数据的获取。执行重放模块与日志过滤模块一样也分为单核版本和多核版本。

### 单核版本重放模块

在单核版本中，重放模块的执行流程如图 42所示，重放模块会首先获取过滤模块中分析得到的任务创建信息来设置任务集中相应任务的创建信息，这些信息在后续CPU管理模块中指导任务的创建，接着逐行分析过滤出的事件信息，如果是同步或者异步事件，说明这些事件后续触发了任务的切换，需要通过CPU管理模块来重现对任务造成的切换；如果是数据流事件，说明任务执行到某位置处需要进行数据的回写，保证任务从外界获取数据的一致性，数据回写过程也是通过CPU管理模块来完成。

其他类型的事件，主要用于辅助重放模块正确的判断某个任务的当前状态，以保证任务在重放中的正确性。对于任务启动事件，CPU管理模块并不会立刻创建对应的任务上下文，而仅仅将任务启动信息维护到任务集管理模块中，供CPU管理模块在适当时机创建任务上下文并启动相应任务运行；对于任务切换事件，重放模块也仅影响任务集中相关任务的状态信息和CPU管理模块下次需要切换进入运行的任务信息；对于任务销毁事件，只需要将任务结束执行反应到任务集中相应任务状态上，CPU管理模块并不会真正去杀死任务，原因是任务一旦运行完成，系统会自动销毁任务上下文，不需要CPU管理模块额外销毁操作。



图 42 单核版本执行重放模块的执行流程

### CPU管理模块

CPU管理模块是重放模块中最核心的功能模块，负责与目标机进行通信，控制具体的任务执行完成执行重放。CPU管理模块通过交叉调试器向目标机发送相应的命令，而目标机代理任务通过断点事件触发CPU管理模块注册的事件回调函数来完成对CPU管理模块的通知。

CPU管理模块工作流程如图 43所示，CPU管理模块通过当前的事件信息，分析出需要设置的断点坐标，并调用交叉调试接口在目标系统上设置断点以及注册该断点对应的回调函数，由于在重放阶段任务的创建和启动也通过CPU管理模块来完成，CPU管理模块在开始相应任务执行时，会查询任务集管理模块相应任务的状态信息，判断任务是否已经在相应的CPU运行起来，如果任务还未被创建执行，CPU管理模块此时需要根据CPU管理模块维护的任务创建信息和启动信息创建启动任务的执行。



图 43 CPU管理模块工作流程

启动运行的任务将会在目标机端就绪，等待被系统调度运行，任务在目标机上可能会被多次换入换出最终才执行到前面设置的断点位置，命中断点，在断点异常处理过程中，目标端调试代理会阻塞当前任务的执行，使其进入阻塞状态，并且通知宿主机端交叉调试器目标端系统上发生了断点事件，主机端的调试器触发设置断点时注册的回调函数。在回调函数中，主要完成上下文的比对，确定任务是否已经正确到达记录执行阶段的切换点。上下文比对时，CPU管理模块会通过主机端交叉调试接口调用目标端操作系统接口完成相关上下文信息的收集，并与记录执行阶段的上下文进行比对，如果两者不相同，说明任务还未执行到对应的切换位置，CPU管理模块会继续任务的执行，每次任务命中断点时触发回调函数执行，进行相同的比对操作直到上下文比对完全相同，此时任务执行达到切换位置，CPU管理模块需要进行任务的切换操作，并且重现下一个任务的执行。

上下文比对根据触发事件类型不同具体的比对操作也不同，比对算法伪代码如表 17所示，主要有三类事件会触发上下文比对操作，分别是同步事件、异步事件以及数据流事件。对于异步事件，上下文比对主要是指同步事件计数器的比对，比对操作相对比较简单，如果读取的目标端任务控制块中的同步事件计数器和记录执行阶段记下的同步事件计数器数值完全相等，说明任务执行到达切换点，可以进行任务的切换操作。对于异步事件，需要比对的信息就比较多，除了32个通用寄存器的比对，一些特殊寄存器的比对外，还需要进行堆栈内容的校验比对，只有这三者比对都是完全一样才说明任务执行到达切换点，否则任务还需要继续执行。对于数据流事件，比对操作不会真正进行，因为在执行阶段每次调用不确定性接口时都会产生数据流事件，数据流事件在日志过滤阶段也没有参入过滤，因此重放阶段每次的数据流事件断点都会被命中，而每次命中时的比对操作都返回真，表明执行已经达到数据回写的位置，可以进行数据的回写操作。

表 17 上下文比对算法伪代码

|  |  |  |
| --- | --- | --- |
| **Algorithm**: task context compare | | |
| **Input**: current\_task\_id, context\_log | | |
| **Output**: isArrivedSwitchPoint | | |
| 1 |  | isArrivedSwitchPoint <- false |
| 2 |  | switch context\_log.type |
| 3 |  | case SyncEvent: |
| 4 |  | replay\_sync\_counter <- getSyncFromTarget(current\_task\_id) |
| 5 |  | if replay\_sync\_counter equals context\_log.sync\_counter then |
| 6 |  | isArrivedSwitchPoint <- true |
| 7 |  | end if |
| 8 |  | case AsyncEvent: |
| 9 |  | regs\_context <- getRegsFromTarget(current\_task\_id) |
| 10 |  | if regs\_context equals context\_log.regs then |
| 11 |  | stack\_sum <- getStacksumFromTarget(current\_task\_id) |
| 12 |  | if stack\_sum equals context\_log.stack\_sum then |
| 表 17 上下文比对算法伪代码（续） | | |
| 13 |  | isArrivedSwitchPoint <- true |
| 14 |  | end if |
| 15 |  | end if |
| 16 |  | case DataEvent: |
| 17 |  | isArrivedSwitchPoint <- true |
| 18 |  | return isArrivedSwitchPoint |

### 任务集信息管理模块

任务集信息管理模块用来管理重放阶段任务的所有相关信息，包括任务创建、任务启动、任务当前状态、任务在记录执行阶段的标识符、任务在重放阶段的标识符以及相互映射关系等。

由于应用程序在记录执行阶段和重放执行阶段是两次不同的执行，因此应用程序创建的任务的标识符不一定是相同的，任务集信息管理模块屏蔽了这两者的区别，无论提供给该模块任务在任何执行阶段的任务标识符都可以获取该任务在重放阶段详细的信息。该模块维护的任务信息和运行在目标机上的任务信息是完全一致的。例如，某个任务当前的状态、切换发生的位置等。

### 多核版本执行重放模块

执行重放模块的多核版本接收多核版本过滤器过滤后的日志信息来分析并控制目标应用的二次执行，其执行流程如图 44所示。多核版本与单核版本存在类似的地方，主要不同体现在事件信息分析和CPU管理模块。事件信息分析一方面多出对CPU索引事件分析，另一方面不需要额外判断同步事件和异步事件是否触发了任务后续切换，因为各个CPU上发生的事件之间的相对顺序是必须在执行重放阶段保证的，否则执行重放将无法正确的得到与记录执行阶段相同的程序执行效果。

多核版本执行重放模块中的CPU管理模块具有同时管理多个CPU的能力，本文中称为多CPU管理模块，多CPU管理模块由多个CPU管理模块组成，每个CPU管理模块对应目标平台上一个CPU。在记录执行阶段，多个CPU并行执行目标应用的多个任务，而在重放执行阶段，二次执行的目标应用的任务被多CPU管理模块控制在目标平台对应的CPU上串行的执行。

CPU索引事件会指导多CPU管理模块动态更改当前焦点CPU，之后产生的日志事件信息被焦点CPU管理模块用于设置断点、控制其对应目标端CPU的任务执行，具体操作可参见4.3.2一节，待焦点CPU执行到记录执行阶段指示其当前应该执行的位置后，通知日志分析模块接着进行后续日志的分析。



图 44 多核版本执行重放模块执行流程

## 本章小结

本章重点介绍了执行重放方案中三大模块具体的实现。首先介绍了信息记录模块的实现，其分为两大部分，探针实现和日志记录实现，探针实现主要关注提供哪些探针，插入在系统哪些路径上，用来记录哪些信息；日志记录实现主要降低探针引入所带来的开销。接着本章介绍了日志分析过滤模块实现，日志分析部分采用Java实现日志文件解析和数据组织，日志过滤部分借助多种不同的过滤步骤实现对最终日志信息的提取过滤。最后，本章给出了重放模块的实现细节，并针对重放模块内部某些关键结构实现细节进行了重点介绍。

# 系统验证

在前两章中，本文主要介绍了基于AcoreOs的执行重放方案的设计和详细的实现步骤。针对本文提出的执行重放方案，本章节主要从功能方面对方案的可用性和正确性进行验证。本章先是对提出的方案中各个模块的功能进行正确性验证，最后通过比对目标程序在记录执行阶段和重放执行阶段的结果来验证本方案对于多核特定应用的有效性和正确性。

## 测试环境部署

本文提出的执行重放方案是基于宿主机—目标机的设计，因此测试平台也有两部分构成：宿主机和目标机。宿主机是运行普通桌面操作系统的PC机，其上运行ACoreOs应用集成开发环境LambdaAE，用于提供应用程序的编写和编译功能，其上也运行Eclipse for RCP/Plug-in Developers 4.2.2集成开发环境，用于开发重放模块。宿主机也负责日志的保存和为解析、过滤以及重放模块的执行提供相应环境。目标机是运行ACoreOs操作系统以及目标应用的开发板，其负责应用程序的运行和任务执行信息的记录。图 45给出了测试环境的架构图，根据图 45描述的测试环境架构图，本文选择并搭建了如下的环境用于测试。

采用普通PC机和Freescale MPC8641D开发板来搭建宿主机—目标机测试平台。本文采用的测试平台中，宿主机是普通PC机，硬件配置是Intel® Core™ i5-4430四核处理器，时钟频率是3.00GHz，4G DDR3内存，1TB SATA硬盘。宿主机上安装运行Windows Xp Sp3操作系统，安装的JRE版本是1.7.0-b147，配备了LambdaAE集成开发环境和Eclipse for RCP/Plug-in Developers 4.2.2集成开发环境。目标机为Freescale MPC8641D开发板，其搭载了两个Power架构e600处理器，单核时钟频率是1000MHz~1500MHz，1G DDR2内存。目标机上运行ACoreOs操作系统、目标端调试代理、目标应用程序以及本文提出的重放方案中必须在目标端部署的应用程序。宿主机和目标机通过以太网相互通信，宿主机通过目标服务器控制目标机上任务的执行，目标机端通过UDP协议将记录的数据上传到宿主机端并保存成对应的日志文件。



图 45 测试环境架构图

经过重放测试，本文提出的执行重放方法适用于具有以下特点的目标应用程序：

单核下多任务并发：采用信号量和消息队列机制实现任务通信，任务执行过程中可产生中断以及任务执行中可调用执行结果具有不确定性的函数。

多核下多任务并行：采用信号量和消息队列实现任务通信，任务执行过程中可以产生中断以及调用执行结果具有不确定性的函数，但是正如本文前面提到的，对于多核的交叉内存访问和重放任务的核间迁移不支持。下面将选取具有代表性的几个应用程序对该执行重放方案进行验证。

## 实验目标

此实验的目的是验证本文提出的执行重放方案中各个模块功能的有效性和正确性，也为了测试该方案引入的探针对于系统执行造成的影响。本文执行重放方案包括三个模块：信息记录模块、日志解析过滤模块、重放模块。这三个模块协同工作，共同完成应用程序的执行重放。本文下面会选取编写的数个测试应用程序，如表 18所示，对本文提出的执行重放方案进行测试和量化引入的探针对系统的影响，并最终验证该方案对于此类应用程序的有效性和正确性。

表 18 实验测试程序概述

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 程序名称 | 程序代码行数 | 任务数 | 程序基本功能 | 用处 |
| A1 | 150 | 8 | 实现生产者消费者模型 | 验证同步事件重放 |
| A2 | 100 | 7 | 打印从外界获取的不确定数据 | 验证数据流重放 |
| A3 | 100 | 4 | 模拟耗时的计算，期间产生中断 | 验证中断异步事件重放 |
| A4 | 150 | 8 | 实现多核版本生产者消费者模型 | 验证多核事件顺序 |
| A5 | 500 | - | 4较典型的测试程序 | 测试探针开销 |

样例测试程序代码具体见附录。

## 实验结果

本节设计并完成了4个典型的程序的重放，用来测试重放方案的正确性和有效性，并设计实现了4个测试程序度量了记录执行阶段引入的探针对于系统运行的影响。

### 同步事件重放

为了验证本文提出的执行重放方案对同步事件重放的正确性，本文设计并实现了含有大量同步与互斥操作的生产者消费者模型测试程序，当然该测试程序在执行中也会引入异步（中断）因素，故也会验证对异步事件的重放。目标应用中的各个任务之间通过信号量实现同步互斥，实现典型的生产者消费者模型。目标应用中包含8个任务，其中4个任务扮演生产者的角色，4个充当消费者。生产者任务的入口函数是task\_producer\_proc（具体可参见附录样例代码），消费者任务的入口函数是task\_consumer\_proc，生产者和消费者之间通过共享环形缓冲区来交互数据，data\_buffer（容量为3）定义了该环形缓冲区，为了实现生产者和消费者任务之间的同步互斥，目标应用中定义了数个信号量和互斥锁。

信号量sem\_4\_read被生产者在生产数据后通知消费者取得相应的数据，sem\_4\_write被消费者用来在取得某个数据后通知生产者可以生产更多的数据。互斥锁mutex\_4\_write用来互斥多个生产者向环形缓冲区中添加自己生产得到的数据，mutex\_4\_read用来互斥多个消费者同时从缓冲区中获取数据。整型变量read\_index和write\_index分别用来指示当前环形缓冲区读取和写入位置。应用程序在初始化时，信号量sem\_4\_write和sem\_4\_read初始值分别是缓冲区容量和0，生产者任务首先获取sem\_4\_write，如果获取成功说明目前可以继续生产数据，存放到write\_index指示的位置并释放sem\_4\_read信号量来通知消费者缓冲区中存在可消费数据，否则由于缓冲区被放满生产的数据，生产者任务被阻塞，等消费者消费数据后唤醒。消费者与生产者实现大体相同。下面，本文将使用提出的执行重放方案对该应用程序进行执行重放。

* 1. **记录执行阶段**

记录执行阶段完成对目标程序执行过程中信息的收集。记录执行阶段，首先将该测试程序和ACoreOs操作系统一起编译打包成可执行文件下载到MPC8641D开发板，接下来系统将初始化和启动信息记录模块，紧接着测试程序将被系统引导运行，此时系统运行中的动态信息将被上传到宿主机端，保存到target.svr文件。

图 46是该测试程序在记录阶段的一次运行的部分输出结果，从运行结果上，可分析出程序粗略的控制流信息是任务1（UserTask01） 🡪 5 🡪 1 🡪 2 🡪 3 🡪 5 🡪 6 🡪 7 🡪 4 🡪 1 🡪 2 🡪 8 🡪 5 🡪 6 🡪 3 🡪 5 🡪 4等等。本测试没有从外界获取数据，故不存在数据流信息。下面，将采用记录的日志分析出测试程序具体切换位置和流程。

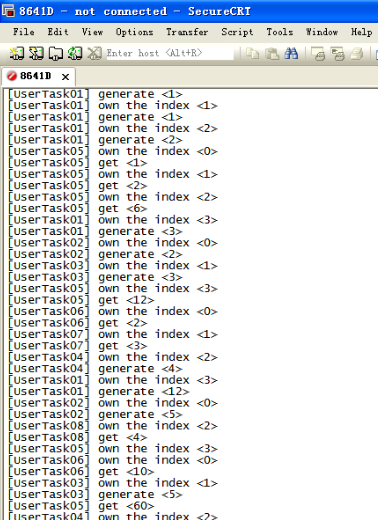


图 46 同步事件测试程序运行结果

* 1. **日志分析过滤阶段**

日志分析过滤阶段根据测试程序在记录执行阶段收集的信息分析出测试程序中各个任务的数据流、切换流程以及触发切换的事件等信息。该阶段解析记录执行阶段保存的target.svr文件，由于target.svr文件中不仅包含测试程序运行的执行信息还有系统自身的守护任务、本文提出的为记录执行阶段服务的几个守护任务等执行所产生的信息。日志过滤模块负责过滤非测试应用创建的任务产生的日志信息并分析出任务切换流程和触发切换的事件。图 47给出了对日志文件过滤分析之后的结果，触发任务切换的事件包含信号量获取和释放事件，由于篇幅限制，图 47中仅显示了日志分析过滤后的部分结果。在重放阶段，重放器将根据该日志信息进行应用程序的重放。



图 47 同步事件测试程序执行结果分析结果

从日志分析的结果，任务1~8按照先后顺序被创建并启动进入就绪态，由于任务1~8拥有相同的优先级，因此ACoreOs在后续的任务调度中，会按照任务就绪的先后顺序调度各个任务进入运行状态。图 48给出了根据日志绘制的测试程序执行部分时序图。

任务1充当生产者首先得到了运行，在生产3个数据将缓冲区填充满后，由于获取不到有效的缓冲区可写信号量（sem\_4\_write）而被阻塞，在任务1阻塞前，任务1已经调用信号量接口操作12次，这一点从记录得到的同步事件计数器上也可以得到。ACoreOs换出任务1换入任务2，任务2刚执行也会因为获取sem\_4\_write操作而被阻塞，同样的，任务3和任务4表现与任务2是一样的。直到操作系统调入任务5开始运行，任务5作为消费者，请求获取的信号量是缓冲区可读信号量（sem\_4\_read），该信号量在任务1生产数据后被释放，任务5可以成功获取到，并在每次消耗一个数据后释放sem\_4\_write来通知生产者可以生产更多的数据。任务在自身的时间片中一次性消耗完缓冲区中所有数据而因为没有后续数据供读取而被信号量sem\_4\_read阻塞，任务5被

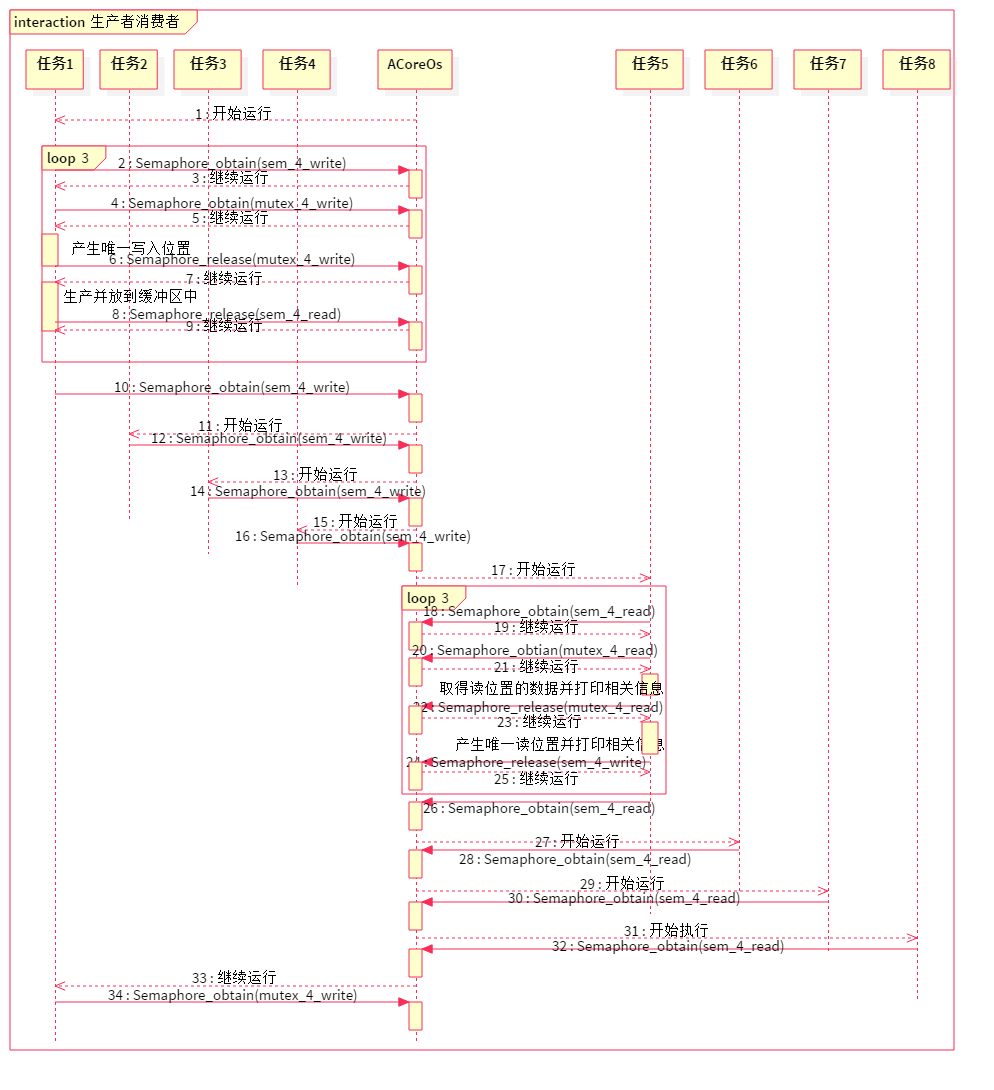


图 48 同步事件测试程序执行时序图

系统换出，后续换入的任务6~8会由于无法获取sem\_4\_read信号量而分别被阻塞，由于生产者被任务5（消费者）唤醒，任务1继续运行，任务1运行一段时间后，被系统换出而换进其他任务继续运行。从日志分析结果可以看出测试程序实际的控制流信息是1（UserTask01） 🡪 2 🡪 3 🡪 4 🡪 5 🡪 6 🡪 7 🡪 8 🡪 1 🡪 2……而不是仅仅从程序运行结果观察出来的1 🡪 5 🡪 1 🡪 2……，其实1 🡪 5的过程中还参入了任务2、3、4的切换与运行，之所以运行结果中无法表现出来，是由于打印语句所在的位置所约束的。任务2/3/4还没执行到打印语句处就因为无法获取信号量而被阻塞。可以看出的是，任务的执行输出只是任务执行过程最粗略的表现，保证应用程序的执行流信息与记录执行阶段的完全相同才保证程序执行效果的相同。下面将对该测试应用程序进行控制流信息的重现操作。

* 1. **执行重放阶段**

前面，已经分析出整个测试程序执行中，任务切换流程和触发切换的事件以及对应的位置。重放模块将根据这些信息来重现应用程序的二次执行。重放模块首先根据日志分析出任务在执行阶段被创建的流程以及对应的参数信息，如图 49所示，并将此信息托管给重放模块中任务集信息管理模块，接着通过交叉调试接口创建对应任务的上下文，创建的顺序与其记录阶段被创建顺序相同，如图 50所示。



图 49 日志中任务创建事件信息的提取和托管

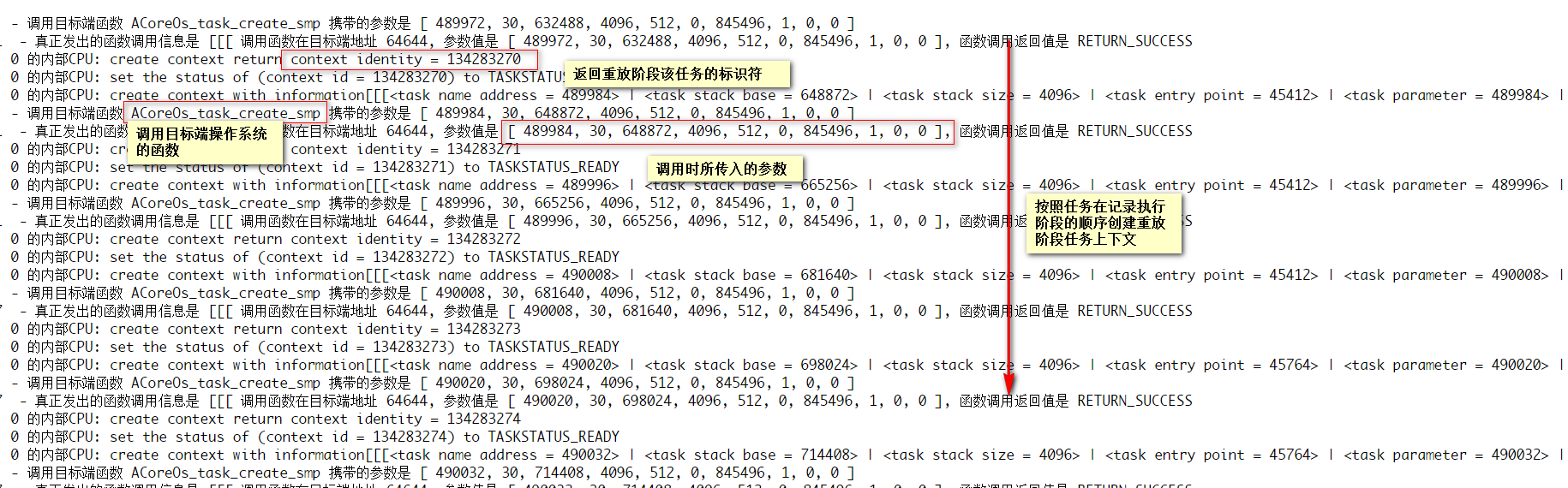


图 50 重放模块根据日志创建任务上下文

重放模块在完成上下文创建后，根据任务在执行阶段的执行顺序启动任务的执行。重放模块分析出应用程序中首先得到运行的任务以及该任务因为何种事件被系统换出，对于本测试程序，任务1首先得到运行，运行一段时间因为无法获取有效的sem\_4\_write而被该信号量阻塞，从而被系统换出。根据日志分析结果可以看出，信号量获取事件导致任务1切换，重放模块在启动任务1运行时，会根据记录得到的信号量获取事件信息中的PC数值设置对应的断点，紧接着启动任务1的运行，如图 51所示，在系统中设置了断点坐标为48912，并将开始运行任务标识符为0x8010006（任务1）的任务。

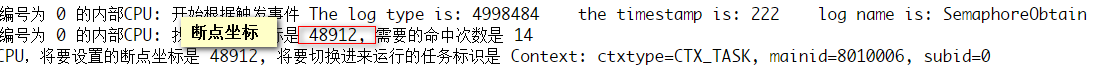


图 51 重放模块根据同步事件信息设置断点并启动相应任务

任务1得到运行后，会因为调用了ACoreOs\_semaphore\_obtain接口而不断的命中断点，每一次命中断点，目标机调试代理通知宿主机调用断点注册的回调函数，在回调函数中，进行任务上下文的比对。由于触发任务1切换的是同步事件，上下文的比对操作仅仅通过比对重放阶段任务的同步事件计数器和记录阶段记录得到的同步事件计数器，如果两者不相同，说明任务1还未执行到达其记录执行阶段的切换位置，需要继续运行，并在再次命中断点时进行相似比对操作，否则说明切换点已经到达，可以进行后续的切换操作。如图 52所示，任务1在记录阶段其切换位置对应的同步事件计数器数值是14，因而在重放执行时，也需要保证该任务同步事件计数器数值等于14。

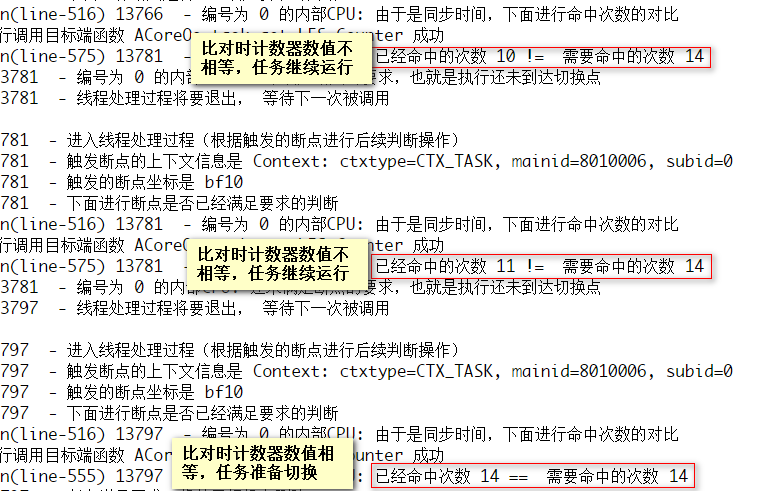


图 52 同步事件设置的断点命中后上下文比对过程

重放模块通过交叉调试接口调用操作系统提供的接口动态获取重放阶段任务的同步事件计数器，在其数值和记录阶段不同时，调用调试接口继续任务运行，在相同时，重放模块需要完成以下操作：

1. 删除断点，调用调试接口删除为了确保当前执行任务的切换位置而设置的断点。
2. 异步事件计数器清零，由于任务在记录执行阶段发生切换操作时，异步事件计时器数值会被清零，因而在重放执行阶段，重放模块必须保证任务执行到达切换点，进行切换时对异步计数器数值清零。重放模块借助内存写入接口对异步事件计数器清零。
3. 根据日志分析出下一个切换进入运行的任务以及触发其切换的事件信息，分析触发切换的事件信息在目标机相应位置设置断点并绑定断点事件回调函数，紧接着开始新任务的运行。

如图 53所示，重放模块分析出下一个需要切换运行的任务描述符是0x810007（十进制134283271，如图 50所示），其是任务2在重放执行阶段操作系统为其分配的一个描述符，任务2运行中，会因为信号量获取操作而被换出，并且到达切换位置时同步事件计数器数值为2。

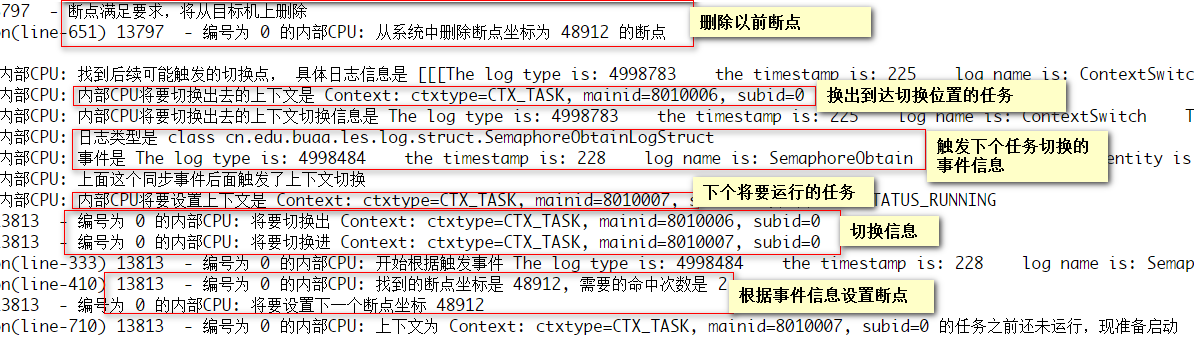


图 53 到达切换点后任务切换过程

重放模块在启动任务运行时，会判断该任务之前是否已经被创建并执行，如果仅被创建，重放模块需要调用操作系统启动任务的接口使任务启动运行，否则，借助调试接口继续任务执行的接口就能够继续被断点打断任务的运行。

重放模块控制测试程序完成测试程序的二次执行，执行结果如图 54所示， 比对图 46和图 54可以看出，程序重放执行输出与程序记录执行的输出是一致的，表明重放模块根据日志分析结果控制程序再次执行时，任务的切换已经不再具有不确定性，验证了记录模块在记录阶段捕获了任务控制流中同步事件造成的不确定性。同时，测试程序在重放执行中，各个任务的切换点是通过同步事件计数器来指示的，并且执行重放阶段的同步事件计数器到达了记录阶段指示的切换位置时，控制重放任务切换，程序的控制流信息就得到了重现，由于该测试程序没有数据流信息，重放模块无需对数据流进行重现，最终程序的输出结果也是完全相同的。通过本测试程序，成功验证了本文提出的重放方案中引入的同步事件计数器对于同步事件触发任务切换位置标示的正确性，也验证了重放方案对同步事件重放的正确性。

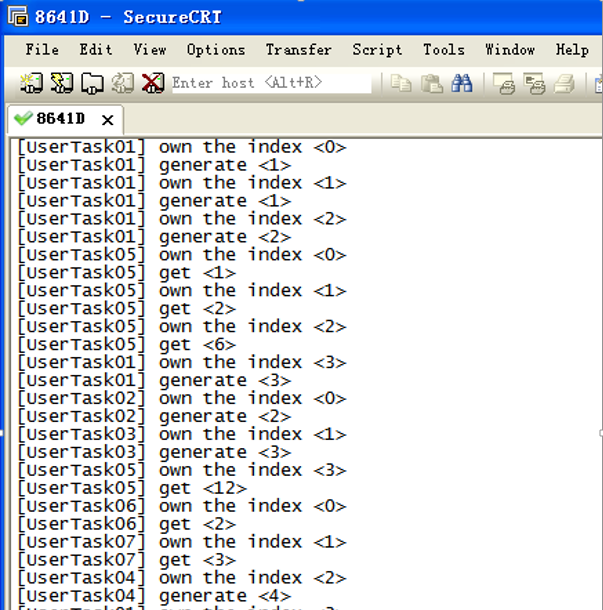


图 54 生产者消费者程序重放执行输出结果

### 数据流重放

为了验证本文提出的执行重放方案对于数据流重放的正确性，本文设计并实现了测试程序中含有对返回值具有不确定性的函数的调用。一个是rdtsc()函数，另外一个是ACoreOs\_msgqueue\_receive()函数。rdtsc()模拟实际中程序从外界传感器获取数据，而ACoreOs\_msgqueue\_receive()模拟不同的应用之间或者应用和中断服务函数之间相互交换数据。测试程序中，一共包含7个任务，其中任务1~3的入口函数是task\_data\_generator，这三个任务循环的通过rdtsc()函数获取系统的时间戳，并将获取的数据结果写入到消息队列中，同时打印输出；任务4~7的入口函数是task\_data\_printer，这四个任务循环从消息队列中接收数据，并将接收的数据打印显示出来。msgq\_4\_connect定义任务通信使用的消息队列，其相应的配置是3个10字节的缓冲区，当消息发送任务向已满的消息队列中写入数据时，该任务会被阻塞；当消息接收任务从空的消息队列中获取消息时也会阻塞，从而msgq\_4\_connect在该测试程序中不仅模拟了应用从外界获取数据也模拟了任务之间的同步过程。下面，本文将使用提出的执行重放方案对该应用程序进行执行重放。

1. **记录执行阶段**

和同步事件重放实验类似，程序执行中动态收集的信息会上传至宿主机端并保存在target.svr文件中，图 55是该测试程序的一次运行的输出结果。从运行结果上，可分析出程序粗略的控制流信息是1（UserTask01）🡪 2 🡪 3 🡪 4 🡪 5 🡪 6 🡪 1 🡪 2 🡪 7 🡪 3 🡪 4 🡪 5 🡪 1 🡪 2 🡪 6 🡪 7 🡪 3 🡪 4。本测试程序的任务存在从外界获取数据，分别是任务1~3采用rdtsc()函数从系统获取具有不确定的数据，以及任务4~7通过消息队列从其他任务获取消息数据。如图 55所示，任务1/2/3通过同一个rdtsc()函数获取的结果是不相同的，任务4/5/6从消息队列中获取的数值也是不同的。

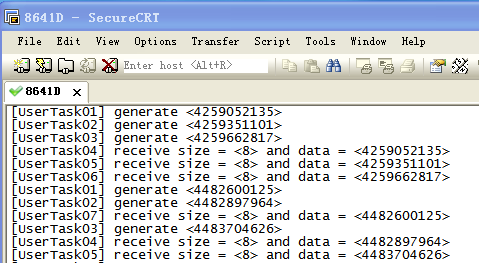


图 55 数据流测试程序运行结果

1. **日志分析过滤阶段**

日志分析过程与实验A1类似，图 56给出了该测试程序记录的日志分析过滤结果，从结果中可以看出任务执行中不仅包含了控制流信息还包含了数据流信息。该测试程序的执行流程如图 56所示，任务1首先被系统换入运行，运行后调用rdtsc()接口导致数据流信息的产生，由于rdtsc()在该目标系统中获取的是两个32位的变量数值，一个保存时间戳的高32位，另一个保存时间戳的低32位，因而在图 55中可以看出数据流产生是成对出现的。任务1运行一段时间后，由于主动睡眠而被系统换出，系统换入任务2运行，任务2执行行为和任务1是一样的，发送数据的任务按照此切换顺序被依次执行，接收数据任务也会按照被创建的顺序被系统调入运行，在换入任务7时，由于任务4-6已经将消息队列中数据消耗完，任务7会因为消息队列的接收操作而被阻塞，此时系统随机选择一个任务继续运行，从分析的结果中可以看出该测试程序的控制流信息是1 🡪 2 🡪 3 🡪 4 🡪 5 🡪 6 🡪 7 🡪 1 🡪 2 🡪 7 🡪 3…不同于仅仅从输出结果上分析出的任务控制流信息。下面将对该测试程序进行执行流和控制流信息的重放操作。



图 56 数据流测试程序记录得到的日志分析结果

1. **执行重放阶段**

重放阶段负责应用程序控制流和数据流信息的重现。重放阶段主要通过调用交叉调试器提供的接口启动测试程序的二次执行，通过分析事件信息设置相关断点，在任务执行过程中暂停任务执行，进行相关上下文比对或数据写回操作。

重放模块会按照测试程序中各个任务的创建顺序创建对应的任务上下文，如图 57所示。创建后根据任务的控制流信息设置断点，由于任务1在记录执行阶段首先被系统换入运行，而且在调用rdtsc()函数产生数据流信息，因而重放模块根据数据流事件信息设置断点，断点设置如图 58所示，重放模块向目标系统中增加了坐标为45196的断点并注册相应的回调函数，接着启动任务1运行。

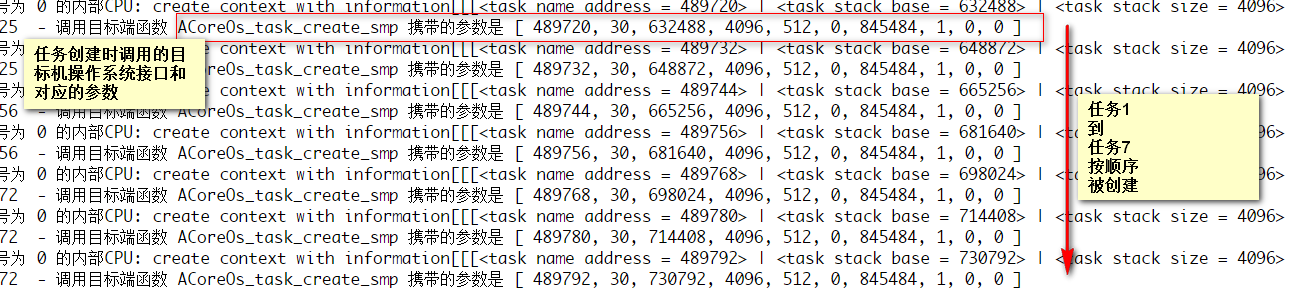


图 57 重放阶段创建重放任务上下文

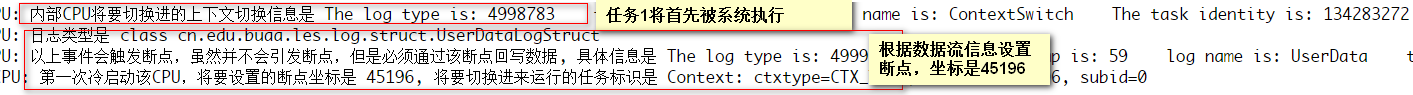


图 58 任务1重放过程中断点设置

待断点被任务1执行命中后，重放模块进行上下文信息的比对，由于断点是根据数据流事件信息设置的，因而不存在上下文比对操作，仅需要正确进行数据的回写操作。由于在任务记录执行阶段，收集并记录了存放任务产生不确定性的数据的地址和产生的具体数据内容，从而在重放阶段，仅仅借助交叉调试器的内存写入接口传入需要保存回写数据在目标系统上的地址以及回写的数据便可以完成数据的回写操作。如图 59所示，回写的数据地址是636400，回写的数据内容长度是4个字节。

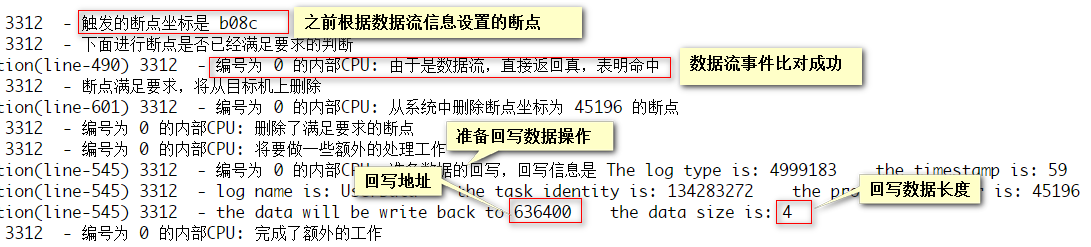


图 59 任务1重放中断点命中数据回写

在对该测试程序重放过程中，也需要对程序的控制流信息进行重放，对于控制流的重现与实验A1是类似的，具体过程可以参看实验A1中重放流程。重放模块按照分析过滤得到的控制流和数据流控制应用程序二次执行，其保证了数据流和控制流信息的一致性，最终测试程序重放执行的输出信息如图 60所示，比对图 55和图 60，测试程序在执行阶段和重放执行阶段通过rdtsc()函数获取到的和通过消息队列接收到的数据都是完全一样的，保证了程序在两次不同的执行中从外界获取的数据都是完全相同的，消除了程序从外界获取数据的不确定性，验证了本文提出的执行重放方案对于数据流记录和重放的正确性。



图 60 数据流测试程序重放执行输出结果

### 异步事件重放

为了验证本文提出的执行重放对于中断等异步事件重放的正确性，本文设计并实现了在执行任务代码中会被中断打断的测试程序。中断分为两种，一种是发生在系统代码中的，另一种是发生在任务代码中。对于发生在系统代码中的，由于ACoreOs系统自身的交叉调试器无法支持对系统级代码的调试，日志分析过滤过程中发现该中断事件导致了任务切换，会采用相应的同步事件替代，所产生的效果是一样的，对于同步事件的重现过程，可参见同步事件重放实验，此实验中并不会涉及。

本实验程序包含4个任务，每个任务完成的工作都是相同的，其入口函数是task\_interrupt\_gen函数，该函数在执行过程中，会调用比较耗时的循环操作，循环体中的do\_something()函数模拟了需要长时间的数据处理操作，从而任务在执行过程中，由于耗完系统此次分配给自身的时间片且没有主动释放CPU控制权，此时CPU会强制该任务暂停运行，换入其他等待调度运行的任务，这个过程主要通过实时钟中断完成的，该中断会打断任务执行，而任务正在执行任务代码，从而产生本实验需要验证的对象：发生在任务代码中的触发任务切换的中断事件。下面，本文将使用提出的执行重放方案对该应用程序进行执行重放。

1. **记录执行阶段**

与上面实验一样，测试程序在记录执行产生的日志信息被保存在宿主机上target.svr文件中。图 61给出了测试程序一次执行的输出结果，从测试程序执行输出结果可以看出，任务1运行一段时间后由于时间片已经被消耗完，操作系统中实时钟中断处理函数中将换出任务1，换入任务2开始运行，任务2执行相同的函数，任务2在执行过程中同样会因为时间片被中断打断而被换出，依次类推，直到应用程序中所有任务都完成运行。下面将使用本文提出的执行重放方案对该测试程序进行重放。

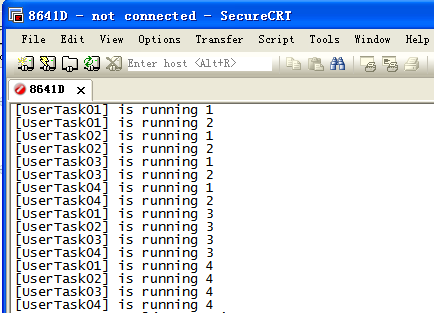


图 61 中断测试程序执行阶段输出结果

1. **日志分析过滤阶段**

记录执行阶段得到的日志文件经过日志分析过滤模块处理后，得到的结果如图 62所示。



图 62 中断测试程序记录得到的日志分析结果

从图 62中可以看出，任务1执行首先被系统换入执行，执行到PC数值等于45532（图 62中srr0寄存器保存的是中断产生时，被打断任务当前执行的PC数值）时，由于时间片被消耗完，被中断打断而被系统换出，紧接着测试程序中任务2被系统换入开始运行，运行一段时间后，同样被中断打断，只不过产生中断的位置不同于任务1，任务2产生中断的位置是PC数值等于48736，通过比对发现，根据记录阶段的日志分析出的触发各个任务切换的事件、执行顺序和测试程序执行输出分析的结果完全相同。

1. **重放执行阶段**

重放阶段，任务的信息维护和上下文创建具体可参见实验1和实验2，该实验的重放阶段重点介绍中断异步事件的重放过程。重放模块首先获取到日志分析过滤模块分析出导致任务切换的中断事件信息，以该事件信息中保存的SRR0寄存器数值作为断点坐标在目标系统上设置对应位置的断点和在宿主机端注册断点处理函数，如图 63所示，任务1在就绪前会先在系统中设置断点坐标为45532的断点，接着任务1被重放模块就绪，等待被操作系统在合适时机换入运行。任务1在其执行中，命中前面设置的断点，触发回调，在回调函数中，重放模块需要进行任务上下文的比对，这里上下文是任务执行被打断时系统为任务生成的上下文以及附加的堆栈校验和。记录执行阶段任务的上下文在中断打断任务时由记录模块收集和记录，同时记录模块也收集记录任务当前使用的堆栈内容的校验和；重放执行阶段，任务执行中一旦命中断点，调试代理会负责保护任务的上下文到任务TCB中，而任务的堆栈校验和需要重放模块借助内存读取接口获取。在具体比对时，首先比对上下文信息中的通用寄存器数值，如果数值不完全相同，则说明任务并没有到达切换点，需要继续运行，一旦通用寄存器数值完全相同，重放模块才会调用调试接口获取当前任务的堆栈校验和，并进行比对。如果上下文信息完全相同，说明任务已经执行到达切换点，重放模块可以根据日志分析结果控制任务进行切换。如图 63，每次进入断点处理函数进行上下文比对时，由于通用寄存器R0的数值不相等，说明任务未执行到切换点，需要继续执行，直到重放执行阶段R0寄存器的数值等于1036，并且其他的寄存器和堆栈校验和都相同。此处的切换操作与实验1中任务执行到同步事件指示的切换点后重放模块进行的切换操作完全相同。最终，任务在重放模块的控制下完成运行，输出结果如图 64所示，比对图 61和图 64，发现两者的输出结果完全相同，并且任务执行中切换位置也完全相同，验证了在记录执行阶段成功捕获异步事件对程序控制流造成的不确定性，并且在重放时，确定性的重现了异步事件对于程序控制流的影响。



图 63 中断事件重放过程

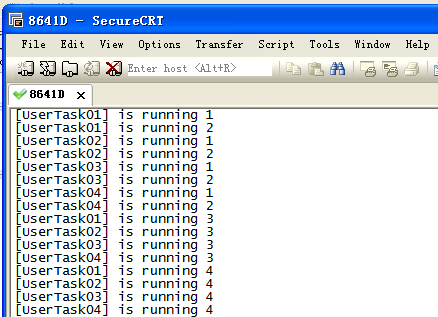


图 64 中断测试程序的重放输出结果

### 多核事件顺序重放

为了验证本文提出的执行重放对于多核事件顺序重放的正确性，本文设计并实现了多核版本的含有大量同步与互斥操作的生产者消费者模型测试程序。本测试程序与实验A1测试程序执行的代码大体相同，不同的是该测试程序一部分任务绑定在0核上运行，另外一些任务绑定在1核上运行。不同的任务之间通过计数型信号量互相同步互斥。任务1、3、5、7被绑定在0核上运行，而剩下的任务2、4、6、8被绑定到1核上运行，任务1、2、3、4的入口函数是task\_producer\_proc，任务5、6、7、8的入口函数是task\_consumer\_proc，为了让程序执行结果可以看出任务具体运行在CPU的哪一个核心上，任务通过sysCoreIndexGet()函数获取自身运行所在的CPU核心，并将获取的结果格式化输出。下面，本文将使用提出的执行重放方案对该应用程序进行执行重放。

1. **记录执行阶段**

记录执行阶段的作用与上面的数个测试程序功能都是一样的，负责收集程序执行的动态信息上传到宿主机端保存为target.svr文件。图 65给出了该测试程序的一次执行输出结果，从输出结果可大致分析出任务在各自处理器核心上执行时大体切换流程。

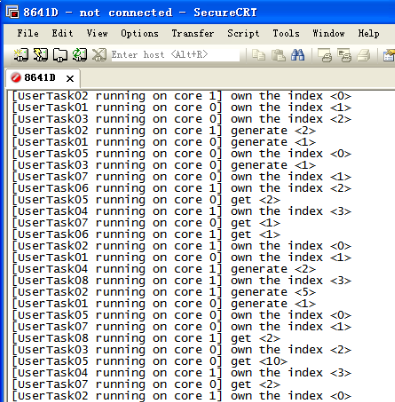


图 65 多核上生产者消费者测试程序执行结果

对于1核，任务切换流程是2（UserTask02） 🡪6🡪4🡪6🡪 2🡪4🡪 8…，对于0核，任务切换流程是1（UserTask01）🡪 3 🡪 1 🡪 5 🡪 3 🡪 7 🡪 5 🡪 7 🡪 1…，同时0核和1核之间存在事件顺序制约，例如，1核上任务2两次输出信息之间必须保证0核上任务1和任务3已经得到执行并输出一部分信息，因而测试程序在多核上运行得到的控制流信息是2 🡪 1 🡪 3 🡪 2 🡪 1 🡪 5 🡪 3 🡪 7 🡪 6 🡪 5 🡪 4 🡪 7 🡪 6…。下面将对该测试程序进行执行重放。

1. **日志分析过滤阶段**

由于该测试程序中任务在多个处理器核心上运行，根据记录的日志信息分析的结果中也会包含CPU核心切换的事件，用于指示后续的事件是发生在某个处理器核心上。图 66中给出了该测试程序日志分析的结果，各个核发生的事件在记录执行阶段被信息记录模块串行化，解决并行程序执行中带来的多核事件发生顺序不确定性。测试程序中任务都是在0核上被创建并就绪，然而会根据绑定的CPU核不同而被放置在不同的CPU就绪队列中，待分配CPU时间片，进入运行状态，任务2被就绪在1核上，并获取到CPU控制权进入运行态，任务2在执行过程中，0核也在并行的创建和就绪测试程序中后续的任务（例如任务4），任务2在1核上运行一段时间后，获取了某个信号量从而产生了信号量获取事件，在该事件发生前，0核已经就绪任务4和任务5，这些不同核之间产生的事件顺序在重放过程中是必须保证的。与单核日志分析过滤模块不同的是，多核日志过滤模块为了确保各个核心上发生事件的相对位置，不可盲目对日志进行过滤操作，仅需要过滤任务执行中产生的嵌套在同步事件中的中断事件。通过对日志的分析，得到测试程序在多核上运行得到的控制流信息是任务（运行的核号）：1(0)🡪2(0)🡪3(0)🡪2(1)🡪4(0)🡪2(1)🡪5(0)🡪2(1)🡪…，下面本文将借助重放模块重现任务在多核上执行产生的事件及其相对顺序。

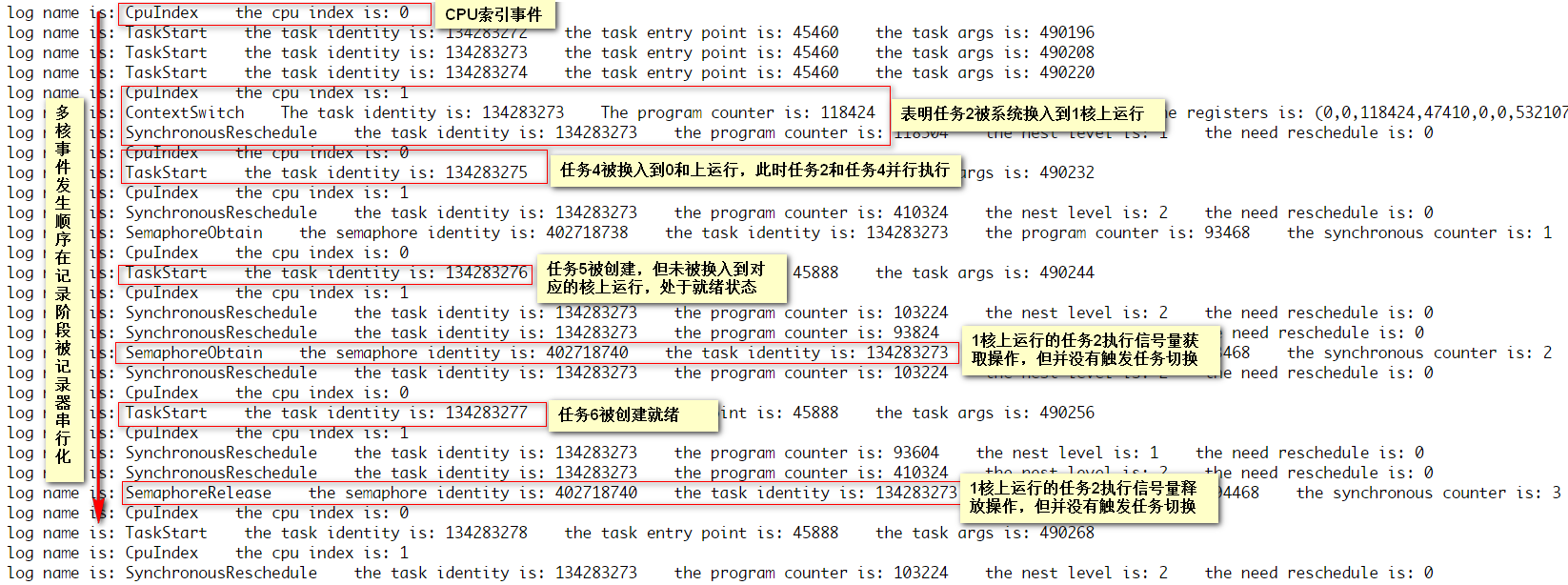


图 66 多核上生产者消费者任务日志分析结果

1. **重放阶段**

本实验重放需要采用多核版本的重放模块（后文简称多核重放模块），该模块首先需要必须保证任务被创建后放置到对应CPU的就绪队列中，通过目标系统提供的创建任务并绑定任务到具体处理器核心可放置任务到指定的CPU任务管理队列中，一旦后续调用任务启动接口，任务将变为就绪态接受CPU调度。

如图 67，根据任务执行阶段被记录的任务创建信息指导多核重放模块在重放执行中创建任务的上下文，例如，记录执行阶段任务标识符是134283272（任务1）的任务在重放阶段被创建获取的任务标示符是0x8010006(134283271)。

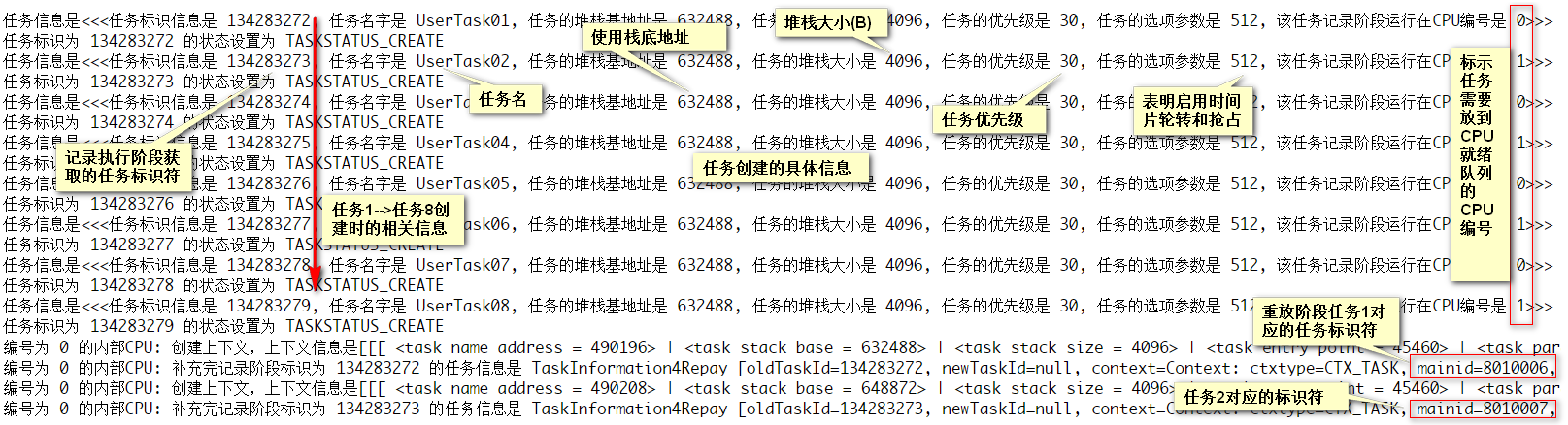


图 67 多核重放模块创建对应任务上下文

多核重放模块和单核重放模块重放操作存在一些相同，多核重放模块对于同步事件、异步事件、数据流事件的重放都借助了断点，都是保证重放程序中仅有一个任务处于就绪状态，然而也存在一些差异，主要体现在以下方面：

1. 对于同步事件重放

在断点回调函数中，不需要比对同步事件计数器的数值，一旦命中断点，该事件就在对应的CPU核上正确重放。

1. 任务切换重放

在单核版本中，不存在与其他CPU核心执行的依赖，任务一旦命中切换点，就可以切入后面将要运行的其他任务，而无需判断其是否运行在当前CPU核心上，对于多核版本，重放事件需要分析该CPU核心是否执行到应该执行到的位置，后续是否还能够继续运行，如果日志表明后续事件是发生在其他CPU核心上，当前CPU需要暂停应用程序中任务的执行，等待其他CPU到达相应的位置。

多核事件重放如图 68所示，1核命中坐标为49128的断点后，探测到接下来0核上任务标识符为0x8010006的任务必须产生信号量获取事件，因而1核完成对应的任务切换操作后，需要通知0核继续重现事件，如此交替，直到按记录阶段的顺序重现所有核上发生的事件。

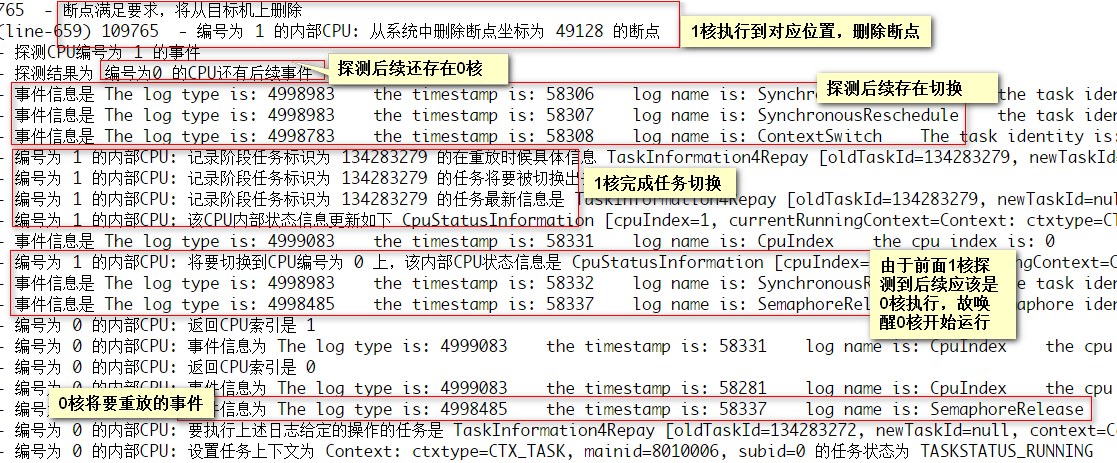


图 68 多核下事件重放过程

多核重放模块控制测试程序二次执行的输出结果如图 69所示。比对图 65（记录执行阶段输出结果）和图 66（日志分析结果），发现日志的分析结果与程序输出相符合，验证了记录阶段对多核事件串行化的正确性。根据日志的分析结果，重放模块对多核事件按顺序进行了重放，比对图 65（记录执行结果）和图 69（重放执行结果），发现两个执行阶段的输出结果一致，说明程序的执行流程是相同的，验证了本问题出的重放方案对于多核事件重放的正确性。



图 69 多核上生产者消费者被重放输出结果

### 插桩开销实验

本文为了测量信息记录模块在记录执行阶段插入的探针对于系统造成的影响，设计并实现了如下4个测试程序，分别是：

1. A1，典型的生产者消费者应用程序（该测试程序会创建8个任务），任务间会频繁采用信号量操作来实现同步和互斥，用于测试信号量的操作带来的探针开销；
2. A2，模拟消息通信的应用程序，任务之间采用消息队列频繁通信，用于测试消息队列操作带来的探针开销；
3. A3，模拟不存在交互的任务，任务之间共享同一个优先级，用于测试中断带来的探针开销；
4. A4，绑定在多核上运行的生产者消费者模型，任务之间会频繁采用信号量操作来实现同步和互斥，并且由于多核可并行产生事件，因此可用来测试对事件串行化所带来的探针开销。

对于每个测试程序，使用rdtsc()函数来计算每次执行所需要的时间。rdtsc()函数读取的是系统时基，其单位是10ns，用来对开销进行衡量，精度已经足够高。

图 70给出了四个测试实验分别运行50次对执行时间取平均后得到的结果，纵坐标表示测试程序执行所花费的时间，以毫秒为单位；横坐标是不同的测试程序，柱上的数据表示插桩前后程序执行时间差。

图 70 插桩开销实验结果

以上测试结果表明，引入的探针的确会给系统带来一定的开销。但这个开销相对于程序的执行所花费的时间是非常小的，并不会对程序执行造成太大干扰。

## 本章小结

本章完成了对提出的执行重放的正确性的验证，并量化了记录执行阶段所引入的插桩探针对系统的影响。首先，本章选择了几个典型的测试样例来分别验证了该执行重放方案对单核上同步、异步、数据流以及多核上控制流事件的正确重放，测试样例的正确重放说明该执行重放方案可应用于此类测试程序的重放。接着，本文通过比对插桩前后程序的运行时间，量化了记录执行阶段引入的探针对系统整体的影响，从测试结果可以看出引入的探针的确会对系统性能造成一定的影响。

# 总结与展望

## 工作与研究总结

当前，几乎所有的应用程序都借助了多线程、多任务技术，来最大化利用系统提供的性能。这些技术的引入给程序开发阶段的调试提出了很大的挑战，循环调试技术作为软件开发过程中的关键技术，可以辅助软件开发人员在软件开发阶段修复很多错误，然而，循环调试方法对于多线程、多任务的应用程序的某些错误的调试不再有效，而执行重放技术作为一项热门技术可以对多线程、多任务应用执行重放，从而辅助调试器对此类应用调试。现有的执行重放技术大多数都是针对于普通操作系统及其应用的执行重放，不能直接应用于多核实时操作应用程序的执行重放。因此研究用于多核实时操作系统下的应用程序执行重放技术是特别有意义的。

本文通过调研国内外现有的重放技术，结合嵌入式多核实时操作系统ACoreOs自身的特点，基于现有的程序开发环境，提出了一种可用于ACoreOs系统上应用程序执行重放方法。本文中选取了数个测试样例对该重放方法进行了测试验证。通过实验分析，证明了本文提出的方法可对该系统下的应用程序进行执行重放。

本文的研究内容和研究成果主要包括以下内容：

1. ACoreOs下应用程序执行重放方案总体设计

本文研究对象所运行的系统是多核实时系统ACoreOs，其提供的应用开发环境是基于宿主机—目标机架构，应用程序运行需要在目标机端，而开发环境部署在宿主机端。针对这种异构的目标环境，结合执行重放的基本技术，本文提出了基于宿主机—目标机架构的多核实时操作系统应用程序的执行重放方案，该方案主要分为三大模块运行在目标机端的信息记录模块、运行在宿主机端的日志分析过滤模块和重放模块。

1. 基于插桩的应用程序执行信息记录

在研究目标系统ACoreOs的内部实现基础上，确定了其上执行的应用程序在执行阶段必须记录的信息，并结合ACoreOs操作系统自身特点提出了基于源代码插桩的信息收集探针，探针被插入在任务执行的某些路径上，在被执行时用来记录任务在系统上运行的详细过程。由于应用程序运行在目标端，其执行过程探针收集的信息也存在于目标端，然而日志分析和重放模块都位于宿主机端，必须保证任务执行时的信息被收集到宿主机端，信息记录模块先暂存任务执行中收集的执行信息并在合适时机上传到宿主机端，并且信息记录模块不能对系统的实时性造成较大影响。

1. 应用程序执行信息的分析和处理

信息记录模块在记录信息过程中，并不知道具体应该收集哪些应用程序的信息，因而，上传的日志信息中存在着其他应用程序运行时产生的日志信息，并且，信息记录模块记录了所有的事件信息，然而这些事件信息仅用来指导日志过滤模块分析出任务的执行流程，对重放不一定有用。在日志信息的分析和处理阶段，会过滤掉其他任务产生的信息，并分析出应用程序中各个任务的切换流程和对应触发切换的事件，以指导重放模块进行应用程序的重放执行。

1. 基于交叉调试器接口的应用程序重放方法

借助开发环境提供的交叉调试接口，本文在宿主机端设计实现了用于控制目标端应用程序执行的重放模块，并根据日志信息分析和处理的结果控制应用程序的二次执行，以实现目标应用程序的执行重放。

## 对未来工作的展望

上述主要讲述了本文的研究内容和取得的成果，但本文提出的执行重放方案并不完整，还存在一些问题需要解决，后续还有一部分工作需要完善和优化。

1. 本文虽然实现了对多核实时系统上应用程序的执行重放，但由于使用软件方式记录多核交叉内存访问顺序开销非常大，对系统的实时性影响特别大，无法满足系统实时性要求，从而本文未解决多核交叉内存访问问题，对于存在多核交叉内存访问的应用程序无法正确重放。
2. 本文提出的重放方案借助了交叉调试环境中的调试器功能，通过在目标系统上设置断点来进行确认任务的切换点，这种方式需要频繁的调用交叉调试器接口从目标机获取任务的相关信息，特别是中断等异步事件的重放，每次获取的信息量较大，导致获取速度慢，而且中断发生位置的不确定性，对于发生在循环体内的中断事件，比对操作将占用大部分时间，重放时间也会大大增加，后续需要找寻相应的解决策略来减少中断事件重放所需要的时间。
3. 目前探针都是源代码级别的，插桩需要花费一定的时间，插桩位置也需要专业的人员才能完成，后续可开发自动化插桩工具，自动化完成对系统提供的编程接口的插桩记录以及对数据流事件的插桩记录。
4. 对于多核交叉内存访问导致的数据竞争，由于记录的开销太大，无法满足系统的要求，本文中没有给出切实有效的解决方案，不过根据已有的研究表明可借助硬件仿真器的支持，完成对交叉内存访问的记录，并完成相应的重放工作，由于本实验环境缺少硬件仿真器支持，并没有验证硬件仿真器对于解决多核内存交叉访问的有效性。

# 参考文献

1. Chen Y., Zhang S., Guo Q., et al. Deterministic Replay: A Survey[J]. ACM Computing Surveys (CSUR), 2015, 48(2): 17-63
2. Konuru R., Srinivasan H., Choi J.D. Deterministic replay of distributed java applications[A]. Proceedings of the 14th International Symposium on Parallel and Distributed Processing [C], 2000: 219-227
3. Pokam G., Pereira C., Dannse K., et al. Architecting a chunk-based memory race recorder in modern CMPs[A]. Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture[C], 2009: 576-585
4. Guo Z., Wang X., Tang J., et al. R2: An application-level kernel for record and replay[A]. Proceedings of the 8th USENIX conference on Operating systems design and implementation[C], 2008: 193-208
5. Elnozahy E.N., Alvisi L., Wang Y.M., et al. A survey of rollback-recovery protocols in message-passing systems[J]. ACM Computing Surveys (CSUR), 2002, 34(3): 375-408
6. Veeraraghavan K., Lee D., Wester B., et al. DoublePlay: parallelizing sequential logging and replay[A]. Proceedings of the 16th international conference on Architectural support for programming languages and operating systems, 2012: 15-26
7. Chen H., Wu X., Yuan L., et al. From speculation to security: Practical and efficient information flow tracking using speculative hardware[A]. Proceedings of the 35th Annual International Symposium on Computer Architecture [C], 2008: 401-412
8. Newsome J., Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software[J], Network & Distributed Systems Security Symposium. 2005:32-51
9. Chow J., Garfinkel T. Decoupling dynamic program analysis from execution in virtual environments[P]. US:20090320009, 2009
10. Goodstein M.L., Vlachos E., Chen S., et al. Butterfly analysis: Adapting dataflow analysis to dynamic parallel monitoring[A]. Proceedings of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems [C], 2010: 257-270
11. Heath M.W., Burleson W.P., Harris I.G. Synchro-tokens: A deterministic GALS methodology for chip-level debug and test[J]. IEEE Transactions on Computers, 2005, 54(12): 1532-1546
12. Sarangi S.R, Greskamp B., Torrellas J. Cadre: Cycle-accurate deterministic replay for hardware debugging[A]. Proceedings of the International Conference on Dependable Systems and Networks [C], 2006: 301-312
13. Su M., Chen Y., Gao X. A general method to make multi-clock system deterministic[A]. Proceedings of the International Conference on Design, Automation and Test in Europe[C], 2010: 1480-1485
14. Chen Y., Li L., Li L., et al. LDet: Determinizing Asynchronous Transfer for Postsilicon Debugging[J]. IEEE Transactions on Computers, 2013, 62(9): 1732-1744
15. Saito Y. Jockey: a user-space library for record-replay debugging[A]. Proceedings of the 6th international symposium on Automated analysis-driven debugging[C], 2005: 69-76
16. Liu X., Lin W., Pan A., et al. WiDS Checker: Combating Bugs in Distributed Systems[A]. Proceedings of the NSENIX Symposim on Networked Systems Design and Implementation[C], 2007: 257-270
17. Geels D.M., Altekar G., Shenker S., et al. Replay debugging for distributed applications[D]. Berkeley. University of California at Berkeley, 2006
18. Russinovich M., Cogswell B. Replay for concurrent non-deterministic shared-memory applications[A]. Proceedings of the ACM SIGPLAN conference on Programming language design and implementation [C], 1996: 258-266
19. Musuvathi M., Qadeer S., Ball T., et al. Finding and Reproducing Heisenbugs in Concurrent Programs[A]. Proceedings of the 8th USENIX conference on Operating systems design and implementation [C], 2008: 267-280
20. Chow J., Lucchetti D., Garfinkel T., et al. Multi-stage replay with crosscut[A]. Proceedings of the 6th ACM SIGPLAN/SIGOPS Internaltional Conference on Virtual Execution Environments, 2010: 13-24
21. King S.T., Dunlap G.W., Chen P.M. Debugging operating systems with time-traveling virtual machines[A]. Proceedings of the annual conference on USENIX Annual Technical Conference[C], 2005: 1-15
22. Xue R., Liu X., Wu M., et al. MPIWiz: Subgroup reproducible replay of MPI applications[J]. ACM Sigplan Notices, 2009, 44(4): 251-260
23. Voung J.W., Jhala R., Lerner S. RELAY: static race detection on millions of lines of code[A]. Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering[C], 2007: 205-214
24. Adve S.V., Boehm H.J. Memory models: a case for rethinking parallel languages and hardware[J]. Communications of the ACM, 2010, 53(8): 90-101
25. Lee D., Said M., Narayanasamy S., et al. Offline symbolic analysis to infer total store order[A]. Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA), 2011: 357-358
26. Altekar G., Stoica I. ODR: output-deterministic replay for multicore debugging[A]. Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles[C], 2009: 193-206
27. Lee D., Said M., Narayanasamy S., et al. Offline symbolic analysis for multi-processor execution replay[A]. Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture[C], 2009: 564-575
28. Park S., Zhou Y., Xiong W., et al. PRES: probabilistic replay with execution sketching on multiprocessors[A]. Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles[C], 2009: 177-192
29. Patil H., Pereira C., Stallcup M., et al. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs[A]. Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization[C], 2010: 2-11
30. Dunlap G.W., King S.T., Cinar S., et al. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay[J]. ACM SIGOPS Operating Systems Review, 2002, 36(SI): 211-224
31. Dunlap G.W., Lucchetti D.G., Fetterman M.A., et al. Execution replay of multiprocessor virtual machines[A]. Proceedings of the 4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments[C], 2008: 121-130
32. Lee D., Chen P.M., Flinn J., et al. Chimera: hybrid program analysis for determinism[A]. Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation [C], 2012: 463-474
33. Xu M., Bodik R., Hill M.D. A" flight data recorder" for enabling full-system multiprocessor deterministic replay[A]. Proceedings of the 30th annual International Symposium on Computer Architecture[C], 2003: 122-133
34. Basu A., Bobba J., Hill M.D. Karma: scalable deterministic record-replay[A]. Proceedings of the international conference on Supercomputing[C], 2011: 359-368
35. Ceze L., Tuck J., Montesinos P., et al. BulkSC: bulk enforcement of sequential consistency[A]. Proceedings of the 34th annual international symposium on Computer architecture [C], 2007: 278-289
36. Luk C.K., Cohn R., Muth R., et al. Pin: building customized program analysis tools with dynamic instrumentation[A]. Proceedings of the 26th annual ACM SIGPLAN conference on Programming Language Design and Implementation[C], 2005: 190-200
37. Thane H., Sundmark D., Huselius J., et al. Replay Debugging of Real-Time Systems Using Time Machines[A]. Proceedings of the 17th International Symposium on Parallel and Distributed Processing, 2003:288-295
38. Sundmark D., Thane H. Pinpointing interrupts in embedded real-time systems using context checksums[A]. Proceedings of the 13rd IEEE International Conference on Emerging Technologies and Factory Automation [C], 2008: 774-781
39. Sundmark D., Thane H., Huselius J., et al. Replay debugging of complex real-time systems: Experiences from two industrial case studies[A]. Proceedings of the 5th International Workshop on Algorithmic and Automated Debugging[C], 2003: 211-222
40. Leblanc T.J., Mellor-Crummey J.M. Debugging parallel programs with instant replay[J]. IEEE Transactions on Computers, 1987, 100(4): 471-482
41. Hower D.R., Hill M.D. Rerun: Exploiting episodes for lightweight memory race recording[A]. Proceedings of the 35th International Symposium on Computer Architecture [C], 2008: 265-276
42. Narayanasamy S., Pokam G., Calder B. Bugnet: Continuously recording program execution for deterministic replay debugging[A]. Proceedings of the 32nd International Symposium on Computer Architecture[C], 2005: 284-295
43. Gait J. A probe effect in concurrent programs[J]. Software: Practice and Experience, 1986, 16(3): 225-233
44. 罗克露. 嵌入式软件调试技术[M]. 北京:电子工业出版社, 2009: 45-47
45. Intel® 64 and IA-32 Architectures Software Developer Manuals [EB/OL]. <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>, 2014
46. 李志丹. 嵌入式软件调试方法研究[J]. 计算机与数字工程, 2012, 40(7): 157-159
47. 张楷, 汤志忠. 一种新的高速嵌入式系统软件调试技术[J]. 计算机工程与应用, 2003, 39(29): 59-62
48. Yang F.M., Ke T., Tu G. Embedded cross debugging software based on JTAG [J]. Computer Engineering & Design, 2005, 10(3): 2817-2819
49. GDB Manuals[EB/OL]. <http://www.sourceware.org/gdb/documentation/>, 2014
50. 彭正文, 徐新爱. 基于 SMP 的 Linux 内核自旋锁分析[J]. 江西教育学院学报, 2005, 26(3): 23-25
51. 薛静. 基于嵌入式实时操作系统的软件开发教学实验的研究与设计[D]. 天津:天津师范大学, 2004: 30-32
52. 王晓春, 刘兴东. 嵌入式实时操作系统任务的同步与互斥机制[J]. 计算机测量与控制, 2005, 12(6): 578-580
53. 林华奇. μC/OS-Ⅱ 系统中优先级反转解决方案的研究与实现[D]. 辽宁:东北大学, 2012: 10-11
54. Lamport L. Time, clocks, and the ordering of events in a distributed system[J]. Communications of the ACM, 1978, 21(7): 558-565

# 附 录

同步事件重放样例代码分为两部分：生产者任务执行代码和消费者任务代码。

生产者任务执行样例代码：

|  |
| --- |
| ACoreOs\_task task\_producer\_proc(const *void*\* param){ |
| const *char*\* task\_name = (const *char*\*)param; |
| UINT32 task\_value = global\_value ++; |
| UINT32 index = 0; |
| UINT32 task\_write\_index = 0; |
| while (stop\_it(index ++)){ |
| ACoreOs\_semaphore\_obtain(sem\_4\_write, |
| ACOREOS\_WAIT, ACOREOS\_FOREVER); |
| ACoreOs\_semaphore\_obtain(mutex\_4\_write, |
| ACOREOS\_WAIT, ACOREOS\_FOREVER); |
| task\_write\_index = write\_index; |
| update\_rw\_index(&write\_index); |
| ACoreOs\_semaphore\_release(mutex\_4\_write); |
| LES\_Printf("[%s] own the index <%u>\n", |
| task\_name, task\_write\_index); |
| data\_buffer[task\_write\_index] = task\_value \* index; |
| ACoreOs\_semaphore\_release(sem\_4\_read); |
| LES\_Printf("[%s] generate <%u>\n", |
| task\_name, task\_value); |
| task\_value = update\_value(task\_value, index); |
| } |
| } |

消费者任务样例代码：

|  |
| --- |
| ACoreOs\_task task\_consumer\_proc(const *void*\* param){ |
| const *char*\* task\_name = (const *char*\*)param; |
| UINT32 task\_value = 0; |
| UINT32 index = 0; |
| UINT32 task\_read\_index = 0; |
| while (stop\_it(index ++)){ |
| ACoreOs\_semaphore\_obtain(sem\_4\_read, |
| ACOREOS\_WAIT, ACOREOS\_FOREVER); |
| ACoreOs\_semaphore\_obtain(mutex\_4\_read, |
| ACOREOS\_WAIT, ACOREOS\_FOREVER); |
| task\_read\_index = read\_index; |
| update\_rw\_index(&read\_index); |
| ACoreOs\_semaphore\_release(mutex\_4\_read); |
| LES\_Printf("[%s] own the index <%u>\n", |
| task\_name, task\_read\_index); |
| task\_value = data\_buffer[task\_read\_index]; |
| ACoreOs\_semaphore\_release(sem\_4\_write); |
| LES\_Printf("[%s] get <%u>\n", |
| task\_name, task\_value); |
| } |
| } |

# 攻读硕士学位期间得到的学术成果

[1] **Chunlei Ma**, Xiang Long, Bo Jiang, Zhenyu Zhang. Replay Debugging of Real-Time Vxworks Applications[A]. International Workshop on Dependable Software and Applications(DSA)[C],IEEE, 2015: 568-573.(Accepted).

# 致 谢

岁月流逝，我在北京航空航天大学两年半的硕士研究生生涯即将结束。回首两年半学习和生活，不禁感触颇多。从2013年刚刚以一个新生的身份踏入北航校园至今临近毕业的两年半时光里，我曾激情万分，对知识充满渴望，对新学校充满好奇，然而也有过迷茫，迷茫是否跟随社会潮流追风逐浪，还是潜心目前的技术研究；迷茫学校里所学的技能是否能通过社会对我们的检验，跻身于技术达人行列；迷茫两年半的付出是否能够得到回报，找到一份满意的工作。两年半的时光虽然即将结束，然而我在这两年半中的收获却跟随自己一生终生受益，在北航里，我学会了如何学习、如何工作、如何与师兄师弟们相处交流，也让我找到了自己的方向和目标，更让我体会到了在技术一流的学府中，应该如何虚心受教，自我鞭策，自我总结，不断进步。为此，我心中充满了对各位老师、同学、家人的感激之情。

首先衷心感谢我的导师龙翔教授，感谢龙老师两年半以来对我的严格要求和耐心指导。在攻读硕士研究生学位期间，我非常荣幸能够成为龙老师的学生，跟随龙老师进行学术研究。龙老师不仅学时渊博、治学严谨，而且为人和蔼，平易近人，热爱生活，热爱技术，是我一生学习的楷模。

此外，要感谢姜博教授，他在我整个硕士研究生生涯中给予我无私的指导和督促。姜老师经验丰富、知识渊博、思维敏锐，是科研项目的带头人。每次科研上遇到疑惑或难点，姜老师总会耐心引导，不厌其烦的为我解答讲解，姜老师的深入讲解不仅帮助我解决了难题而且也完善了我整个的知识体系，进一步加深了对知识的理解，同时，也提高了我发现问题、分析问题、解决问题的动手能力。姜老师对学术的无限执着、对真理的不断追求、对自己的严格要求也是我今后步入社会学习的榜样。

感谢万寒老师一直以来指导我的工作和学习，也感谢万老师在实验室的管理工作，给我营造了一个良好的学习和工作环境。同时，非常佩服她的敬业精神，为整个实验室的良好运营做出了巨大的贡献。

感谢和我一起学习工作的郭兰、李力、马凯，感谢他们在学习和生活上给予的无私鼓励和关怀，正是他们的鼓励才使我坚持下来顺利完成了这个课题！同时，也要感谢实验室的学弟学妹们，他们的到来为实验室注入了新的血液，新的活力，他们乐观、活泼、开朗的个性一直感染我，给我的科研生活平添了很多色彩。研究生阶段我最大的收获是能够进入嵌入式系统实验室，从事自己最感兴趣的方向，大家在科研项目上同舟共济，生活生互帮互助，给实验室营造了非常好的氛围。与大家在一起的美好时光，必将是我人生中的一笔财富，也是我学生时代美好的记忆。在这里，我也祝愿实验室的所有学弟学妹都能顺利毕业，找到满意的工作，跻身于技术达人行列。

特别要感谢的是我的父母，感谢他们的养育之恩，他们虽然没有什么文化，地位卑微，但深知读书是农村孩子唯一好的出路，在养育我25年的漫长岁月里，父母给了我无微不至的关怀，激励我发奋图强，不断进取。他们的付出和期盼是我能迈出农村进入大学最大的动力源。

最后，感谢各位老师在百忙之中抽出宝贵的时间对我的论文进行评阅和审议，谢谢您们。