基于 Qemu 的龙架构平台高性能插桩工具

项目成员：陈曦 高林萍 穆奕博

指导老师：徐伟 卢建良

学　　校：中国科学技术大学

2024 年全国大学生计算机系统能力赛

操作系统设计赛（全国）

OS 功能挑战赛道

摘　要

本项目为2024年全国大学生计算机系统能力赛-操作系统设计赛-OS功能挑战赛道的333号赛题：基于Qemu的龙架构平台高性能插桩工具。

Intel公司研发的插桩工具Pin是一款功能强大、应用灵活且适用于多种指令架构的动态二进制插桩工具，该软件在不影响原始程序执行结果的前提下，按照用户需求插入分析代码，完成对程序运行过程中行为的监控和分析。Qemu是一款当前主流的多平台开源翻译框架，具有良好的平台可扩展性，同时支持系统级和用户级翻译。为了扩展龙架构上插桩工具的丰富性，推动龙芯生态建设，本团队依托赛题提供的代码框架，在初赛阶段丰富了该插桩框架中的插桩API。

项目在初赛阶段的主要工作如下：

1. 探索Intel Pin对TRACE插桩粒度的定义，完成其在已有框架上的实现。

2. 测试Intel Pin所有TRACE粒度API的功能，对功能进行统计和总结，并在框架上实现这些API。

3. 探索Intel Pin对IMG和RTN插桩粒度的定义，并完成其在已有框架上的实现。

4. 测试Intel Pin所有IMG和RTN粒度API的功能，对功能进行统计和总结，并在框架上实现这些API。、

5. 编写插桩工具测试已实现的API的正确性。

6. 尝试优化框架，包括改善其对INS级以及BBL级的指令翻译，减少了不必要的代码翻译，使其更加契合Intel Pin的API，提高插桩软件在应用时对代码分析的准确性。

7. 开源全部资料。分享源代码、项目文档等资源，便于继续学习和交流。

本团队在初赛阶段工作的主要创新点如下：

1. 修正原有框架存在的不足，重新定义原有框架中的BBL粒度，使其更加符合TRACE粒度的定义和插桩工作。

2. 重新定义TRACE粒度，设计并实现TRACE级插桩粒度的相关API。

3. 定义RTN和IMG粒度，重新编写RTN和IMG获取部分的代码，使代码更加规范易读。添加RTN和IMG两个粒度的通用API。

4. 对于RTN级的插桩，Intel Pin将RTN中所有的指令都存储起来，从而通过对指令的插桩实现对RTN的插桩。本插桩框架选择先记录需要插桩的函数、需要插入的分析函数调用和分析函数调用插入的位置，在程序运行时，通过检测Trace的出入口和RTN的出入口是否相同，进一步在相应位置插桩，从而大大降低了不必要的内存浪费。

目　录

第 1 章 概述

1.1 项目背景与意义

随着软件行业的发展，软件的规模和复杂性持续增长，这为开发人员监控软件运行过程中的行为带来了巨大的挑战。动态二进制插桩（Dynamic Binary Instrumentation，DBI）是一种将额外代码插入应用程序中，以观察其行为的技术。它可以在不影响程序执行结果的前提下，按需求插入分析代码，完成对程序运行过程中的行为的监控和分析。其在代码分析，程序调试，性能分析，逆向工程，代码覆盖率分析，安全性测试和程序优化等诸多场景有重要的应用。在商业软件中，开发者可能会使用二进制插桩来实施反调试和反破解措施。通过在程序中插入特定的指令，可以使调试器无法正常工作，从而增加破解者的难度，以此来达到软件保护的作用。

随着动态二进制插桩的不断发展，各种框架层出不穷，它们拥有各种独特的功能和特点。这些框架不仅支持主流的操作系统和指令集架构，而且针对不同的应用场景提供了专门的解决方案。同时这些框架对底层的二进制代码进行了抽象，通过它们，开发人员可以轻松地在程序运行时向二进制代码中插入额外的指令，无需深入了解其底层操作。这使开发人员可以快速且简单地实现如性能分析、安全检测、逆向工程等各种功能，为软件的开发和维护提供了更多的可能性。

2021 年 Loongarch 自主指令集架构的推出，对芯片国产化有重要的意义，在此之后的龙芯 3A5000 开始的 CPU 将采用 Loongarch 指令集架构[1]。作为新推出的指令集架构，龙芯平台的软件生态系统仍处于相对初级状态。现有的主流二进制插桩框架并不支持龙芯系统，因此，相对于其他平台，目前在龙芯平台上开展动态二进制插桩分析具有一定的技术难度。填补这里的技术空白，开发一款支持龙芯的高效的动态二进制插桩软件，丰富龙芯的软件生态具有重要的意义。

1.2 插桩工具发展历史

二进制插桩技术的历史可以追溯到计算机科学的早期阶段。20 世纪 60 到 80 年代，主要采用汇编级调试器进行程序调试和分析，程序员可能会手动修改程序的汇编代码，插入调试指令或其他额外功能的代码。

到 21 世纪初，英特尔公司推出了轻量级动态二进制插桩工具 Pin[2]。它支持多种平台，包括 Linux、macOS 和 Windows 操作系统以及 IA-32、Intel(R) 64 和 Intel(R) Many Integrated Core 架构的可执行文件。Pin 具有高度的灵活性和可扩

展性，利用丰富的 API，它可以在可执行文件的任意位置插入任意代码。其抽象了底层指令集的特性，并允许将上下文信息（如寄存器内容）作为参数传递给插入的代码。功能强大。

2002 年，HP 和 MIT 联合发布了 DynamoRIO[3]。它起源于惠普公司的 Dynamo 优化系统和麻省理工学院的 Runtime Introspection and Optimization (RIO)研究小组之间的合作，因此得名为"DynamoRIO"。作为一个进程虚拟机，它将程序的执行从原始的二进制代码重定向到该代码的副本。然后，执行所需工具操作的插桩会被添加到这个副本中。原始程序不会进行任何更改，也不需要以任何方式进行特殊准备。DynamoRIO 完全在运行时操作，并处理遗留代码、动态加载的库、动态生成的代码和自修改代码。

除了上述两种外，Valgrind[4]也是一款强大的插桩工具。它是一个用于构建动态分析工具的插桩框架。有一些 Valgrind 工具可以自动检测许多内存管理和线程错误，同时能详细分析目标的程序。它目前包括七种生产质量的工具：内存错误检测器、两种线程错误检测器、缓存和分支预测分析器、生成调用图的缓存和分支预测分析器，以及两种不同的堆分析器。它还包括一个实验性的 SimPoint 基本块向量生成器。

1.3 应用场景和需求

动态二进制插桩可以在不影响程序正常输出的情况下，将分析函数注入到程序中，达到检测程序运行时行为的作用。其在程序分析与调试，安全性检测，调试和故障排查，代码覆盖率分析，逆向工程和程序理解以及软件保护等诸多场景下发挥着重要的作用。

1.3.1 功能需求

插桩接口时插桩框架的使用者与插桩框架之间交互的桥梁，用户编写插桩工具时，通过调用插桩框架提供的插桩接口来描述插桩任务。本团队设计的插桩接口需要在功能上对标 Intel Pin，主要的功能需求是在龙架构上设计出符合 Intel Pin 使用方式的 API。

指令级（Instruction，简称 INS）API：针对单条指令进行操作，API 分为插桩 API 和信息检查 API。其中插桩 API 需要完成用户编写的分析函数在指令粒度上的正确插入，信息检查 API 需要让用户能够检查每一条指令及其相关的信息，例如指令类型、操作数类型、访存地址等，以判断是否为需要插桩的指令。

基本块级（Basic Block，简称 BBL）API：针对单入口、单出口的指令序列

进行操作，API 分为插桩 API 和信息检查 API。其中插桩 API 需要完成用户编写的分析函数在基本块粒度上的正确插入，信息检查 API 需要让用户能够检查基本块的相关信息，如基本块的地址等。

轨迹级（Trace，简称 TRACE）API：针对单入口、多出口的指令序列进行操作，API 分为插桩 API 和信息检查 API。其中插桩 API 需要完成用户编写的分析函数在轨迹粒度上的正确插入，信息检查 API 需要让用户能够检查轨迹的相关信息，如轨迹中的指令数量等。

函数级（Routine，简称 RTN）API：针对函数或例程进行操作，API 分为插桩 API 和信息获取 API。其中插桩 API 需要完成用户编写的分析函数在函数粒度上的正确插入，信息检查 API 需要让用户能够检查函数相关的信息，如函数的地址、指令数等

镜像级（Image，简称 IMG）API：针对可执行程序进行操作，API 为信息检查 API，无需进行插桩，需要让用户能够检查镜像相关的信息，如判断当前镜像是否有效、查询当前镜像名等。

1.3.2 非功能性需求

可靠性需求：

动态二进制插桩技术通过注入分析函数来实现对程序行为的监控，注入新的代码后要保证新注入的代码不影响原本程序的执行，为了达到这一目的，对程序的可靠性提出了要求。本团队需要做到以下几点以保证不影响原本的输出:

1. 插桩前后的程序执行路径不变。
2. 插桩前后程序运行相关寄存器，调用栈等相关状态信息不变。
3. 系统调用应被正确模拟。

性能需求：

动态二进制插桩采用即时编译的方式，会引入额外的运行时开销。因为在程序的执行过程中需要动态生成要执行的代码，程序不断地在代码缓存和插桩框架的上下文之间切换。在满足插桩功能的同时，插桩框架需要尽可能地提高运行效率来保证运行大型代码时能够较为快速地完成插桩工作。在使用 API 时，插桩型 API 对整体性能的影响最大，需要与 Intel Pin 的性能相比较来尽可能提高本插桩框架的性能。基于上述原因，本团队对现阶段及下一阶段的开发工作提出以下的性能要求：

1. 在不进行插桩的情况下，使用本项目中的插桩框架运行应用程序时，效率至少为执行原生程序的 50%。

2. 在进行插桩的情况下，插入分析函数带来的性能下降不得高于 50%。

1.4 本文创新点

在原有框架基础上，本文的做出了许多创新性的工作。

第一，修正了原有框架的不足。本文通过实验测试，更加仔细的重新定义了框架原有的 bbl 粒度，使其符合实际运行的同时适配 TRACE 粒度的定义和插桩工作。

第二，本文从无到有的为一款应用于龙芯平台上的插桩框架设计并实现了一个 TRACE 级插桩粒度。参照英特尔的 Pin，从 x86 平台上开展了一系列测试工作，得到了准确且细致的 TRACE 的定义。

第三，本文设计并实现了对 RTN 和 IMG 粒度的定义，重新编写了 RTN 和 IMG 获取部分的代码，使得本项目代码更加规范易读。

第四，对于 RTN 级的插桩，Intel Pin 将 RTN 中所有的指令都存储起来，从而通过对指令的插桩实现对 RTN 的插桩。本插桩框架选择先记录需要插桩的函数、需要插入的分析函数调用和分析函数调用插入的位置，在程序运行时，通过检测 Trace 的出入口和 RTN 的出入口是否相同，进一步在相应位置插桩，从而大大降低了不必要的内存浪费。

第五，添加了新的插桩接口。对标英特尔公司的插桩工具 Pin，添加了各粒度的插桩接口。

1.5 组织结构

本文的章节安排如下：

第一章 概述，介绍本文的研究背景以及工作的重要意义，介绍二进制插桩技术的发展历史，应用场景以及需求，同时总体概述本文的工作内容和创新点。

第二章 整体框架设计，从总体上概述插桩框架的结构，介绍框架执行与调度流程。

第三章 INS、BBL、TRACE 级 API 的设计与实现，本章介绍了这三个粒度的解析与定义，同时对标 Intel Pin 的 API 的功能，介绍在这个框架下对相关 API 的设计和实现

第四章 IMG、RTN 级 API 设计与实现，本章介绍了这两个粒度的解析与定义，同时对标 Intel Pin 的 API 的功能，介绍在这个框架下对相关 API 的设计和实现。

第五章 插桩工具的设计与实现，本章介绍用户编写的插桩工具的基本结构，同时给出了典型的插桩工具的应用实例。

第六章 系统测试与分析，在实现了插桩框架后，按照需求分析中的功能性

需求和非功能性需求设计测试环境、方案和测试用例，对插桩框架进行测试，确保满足预期的结果。

第七章 总结和展望，对本文的工作进行总结，对插桩框架目前存在的不足和改进空间进行描述分析，展望该插桩框架的未来发展。

第 2 章 整体框架及功能模块设计

2.1 系统整体架构设计

本章将对题目给出的部分代码进行分析，从而引出本文对原始框架的扩充和改进。本章所述内容是团队针对初步代码阅读工作的总结，同时为下一步的开发打下基础。

本文实现的插装框架是基于 Qemu 开发的。Qemu 是一个跨指令集架构的动态二进制翻译器，在本系统中的主要作用是提供虚拟机运行环境。由于动态二进制翻译和动态二进制插装技术有一定程度上的相似性，都采用即使编译（Just In Time，JIT）的方式生成新的二进制代码并代替程序的原始二进制程序，因此 Qemu 中的许多功能模块都可以用在本文的插装框架中。

本系统的整体架构如图 2.1 所示，其主要由虚拟机、代码缓存与插桩接口构成，虚拟机中包含调度器和即时编译器。用户编写的插桩工具通过插桩接口，向插桩框架注册插桩函数。插桩框架先将应用程序加载到内存中完成初始化，随后虚拟机中的调度器负责协调各个部件来运行应用程序。 虚拟机采用即时编译（JIT）的方式运行来应用程序，在截获应用程序的执行流后，对应用程序的将要执行的二进制代码进行解析，得到指令序列并进行插桩，最后生成插桩后的二进制代码，并交给调度器进行执行。插桩后的二进制代码会被保存到代码缓存中，以避免重复的插桩工作。插桩框架拦截应用程序的系统调用，并在插桩框架的控制下模拟执行。

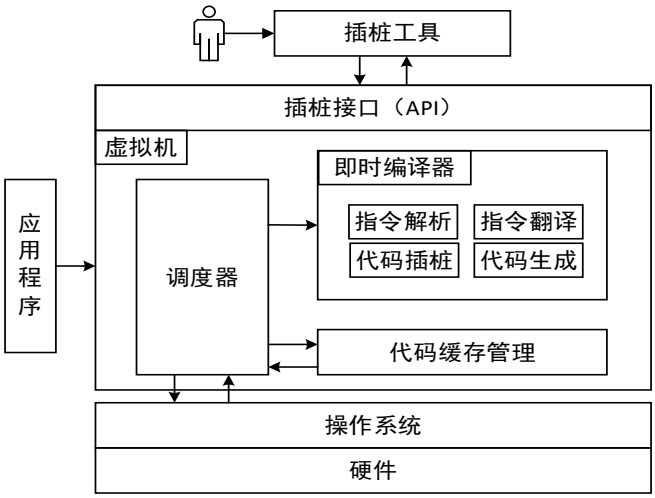

图 2.1 系统整体架构

根据本插装系统的功能，整个系统可以被分为下面的模块：

1. 调度执行模块。负责控制整个系统的运行，通过协调插桩框架中的各个模块，完成 Loongarch 应用程序二进制代码的解析、插桩与执行。调度执行模块的主要功能包括基本块调度、上下文切换、异常处理等。

2. 指令解析模块。负责反汇编原始 Loongarch 指令序列并生成本系统需要使用的相关数据结构。

3. 翻译模块。负责对生成的指令序列进行修改和优化，便于执行。

4. 插桩模块。根据用户编写的插桩工具（Pintool），对翻译后的指令序列进行插桩。

5. 代码生成模块。将翻译及插桩后的指令序列重新汇编，生成可执行的二进制 Loongarch 代码。

6. 代码缓存模块。负责保存和维护插桩后的二进制指令序列，避免重复插桩，同时提高运行效率。

在本文后续中，会对上述模块的详细设计进行介绍。

2.2 调度执行模块

本文介绍的插桩系统基于 Qemu 实现，以基本块为单位进行指令流的翻译、插桩与执行。调度执行模块的主要功能包括基本块调度和上下文切换，图 2.2 展示了该模块协调插桩框架中的不同模块，共同完成不同粒度的插桩和代码执行。

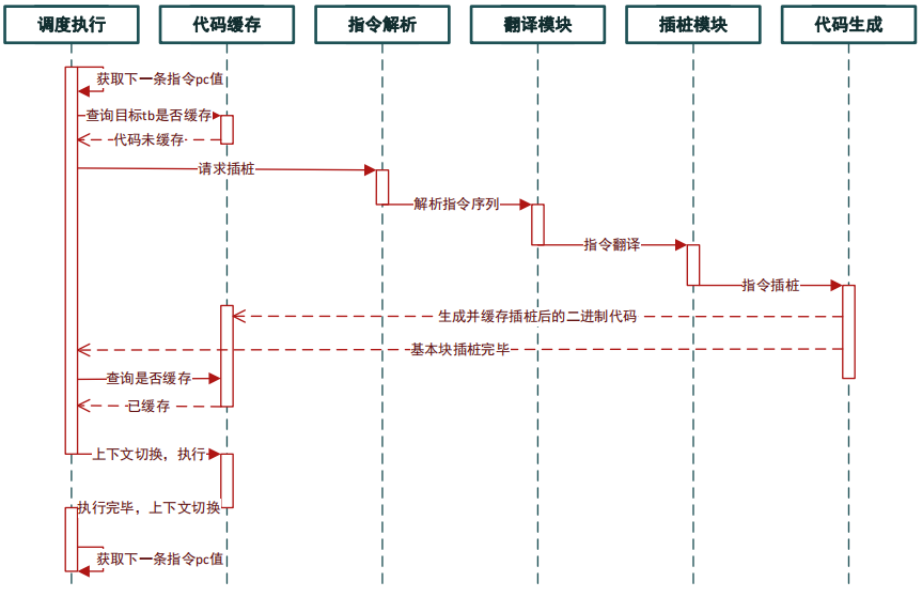

图 2.2 基本块调度时序图

2.2.1 数据结构说明

本文介绍的插桩系统基于 Qemu 实现，以基本块为单位进行指令流的翻译、插桩与执行。调度执行模块的主要功能包括基本块调度和上下文切换，图 2.2 展示了该模块协调插桩框架中的不同模块，共同完成不同粒度的插桩和代码执行。

调度执行模块主要使用 Qemu 原有的部分数据结构完成，需要得到基本块信息和包括寄存器在内的 CPU 状态信息。表 2.1 展示了该模块使用的主要数据结构，结构体 TranslationBlock（TB）保存一个翻译块信息，结构体 CPULoongArchState 保存龙芯 CPU 的信息。

表 2.1 调度执行模块数据结构说明

结构体名	结构体成员
TranslationBlock	target_ulong pc
	target_ulong cs_base
	uint32_t flags
	uint32_t cflags
	uint32_t trace_vcpu_dstate
	uint16_t size
	uint16_t icount
	struct tb_tc tc
	uintptr_t page_next[2]
	tb_page_addr_t page_addr[2]
	QemuSpin jmp_lock
	uint16_t jmp_reset_offset[2]
	uintptr_t jmp_target_arg[2]
	uintptr_t jmp_list_head
	uintptr_t jmp_list_next[2]
	uintptr_t jmp_dest[2]
CPULoongArchState	uint64_t gpr[32]
	uint64_t pc
	uint64_t fpr[32]
	float_status fp_status

续表 2.1 调度执行模块数据结构说明

结构体名	结构体成员
	bool cf[8]
	uint32_t fcsr0
	uint32_t fcsr0_mask
	uint32_t cpucfg[21]
	uint64_t CSR_CRMD
	uint64_t CSR_PRMD
	...

2.2.2 函数说明

调度执行模块的函数可以分为基本块调度和上下文切换两种类型，基本块调度相关函数负责完成 QEMU 翻译块（TB）的翻译、插桩和执行，上下文切换相关函数负责进行寄存器及相关信息的保存和恢复，表 2.2 展示了主要函数。

表 2.2 调度执行模块主要函数及功能

函数名	功能
cpu_exec	qemu 主执行函数
tb_lookup	查找代码缓存中的 TB
tb_gen_code	执行解析、翻译、插桩，生成新的 TB
cpu_loop_exec_tb	执行当前 TB
context_switch_bt_to_native	找到要执行的基本块后的上下文切换
context_switch_native_to_bt	基本块执行完毕后的上下文切换
la_gen_prologue	生成上下文切换序言
la_gen_epilogue	生成上下文切换结尾

2.3 指令解析模块

指令解析模块的主要功能是将应用程序的二进制流解析成 Loongarch 指令组成的基本块，同时构建本插桩框架所需的数据结构，以用于后续的插桩工作。当调度模块请求翻译或插桩一个基本块时，指令解析模块开始执行，基本步骤为：

1. 获取二进制指令，即从当前 PC 值开始，依次读取 4 字节的指令字。
2. 解析操作码。根据 Loongarch64 指令集解析出当前指令的操作码类型。

3. 解析操作数。根据上一步得到的操作码，查询预先定义的指令格式表，从二进制代码中提取出对应数据。解析完全部的操作数之后，便完成了一条指令的解析过程，将这条指令添加到指令链表中。

4. 构建后续插桩工作所需的数据结构。在本文中，主要是 INS、BBL、TRACE、RTN 和 IMG 数据结构的构建，这些会在第 3、4 章中分别介绍。

2.3.1 数据结构说明

表 2.3，表 2.4 和表 2.5 展示了指令解析模块所使用的数据结构。

表 2.3 枚举类型

枚举名	功能
IR2_OPCODE(LA_OPCODE)	指令操作码类型
IR2_OPND_TYPE(LA_OPND_TYPE)	指令操作数类型
GM_OPERAND_TYPE	指令操作数比特域类型
LISA_REG_ACCESS_TYPE	指令寄存器访存类型

表 2.4 结构体类型

结构体名	结构体成员
GM_OPERAND_PLACE_RELATION	GM_OPERAND_TYPE type
	pair bit_range_0
	pair bit_range_1
GM_LA_OPCODE_FORMAT	IR2_OPCODE op
	uint32_t opcode
	GM_OPERAND_TYPE opnd[4]
LISA_REG_ACCESS_FORMAT	IR2_OPCODE op
	LISA_REG_ACCESS_TYPE opnd[4]
	bool valid
LA_OPND	int val
Ins	LA_OPCODE op
	LA_OPND opnd[4]
	int opnd_count
	struct Ins *prev
	struct Ins *next

表 2.5 指令格式表（常量数组）

数组类型	数组名	功能
IR2_OPND_TYPE	ir2_opnd_type_table	操作数格式
GM_OPERAND_PLACE_RELATION	bit_field_table	操作数位域
GM_LA_OPCODE_FORMAT	lisa_format_table	操作数格式
LISA_REG_ACCESS_FORMAT	lisa_reg_access_table	操作数访存

2.3.2 函数说明

表 2.6 展示了指令解析模块的主要函数及功能。

表 2.6 指令解析模块函数说明

函数名	功能
la_decode	解析二进制流、生成不同粒度的数据结构，同时完成翻译和插桩工作
la_disasm	解析龙芯指令的操作码和操作数，保存到指令链表中
get_ins_op	解析龙芯指令的操作码
extract_opnd_val	解析龙芯指令操作数的值

2.4 翻译模块

翻译模块遍历解析完成的基本块，对每一条指令进行处理，包括寄存器的重映射、部分指令的特殊翻译，在这个过程中一条指令的前后可能被插入多条指令。翻译模块需要保证上述操作后产生的执行效果与单条原始指令的效果一致，且对于跳转、系统调用等控制转移指令，翻译模块还需保证指令执行完毕后，程序控制权最终能够回到插桩框架。

在 JIT 过程中，经常需要使用额外的寄存器来保存可能存在的中间值和计算结果，因此在执行应用程序代码时，需将一部分寄存器作为临时寄存器用于保存结果，但这会导致上下文切换时应用程序寄存器无法被全部加载到真实寄存器中。当指令要访问的寄存器未被加载时，翻译模块需要分配临时寄存器用于加载将被访问的寄存器的值，并将原始指令中的寄存器替换为临时寄存器。这个过程被成为寄存器重映射。

部分指令的执行效果不仅取决于指令本身的操作数，如 PCADDI 指令的执

行结果与当前指令的地址有关。在插桩过程中，新生成的指令中的地址与原始指令不同，因此对这种类型的指令需要特殊处理。与此同时，对于控制转移指令，如果直接跳转到原始指令的目标地址，会导致程序运行未插桩处的原始代码，因此这一部分指令也需要进行特殊处理。这个过程被乘坐特殊指令的翻译。

2.4.1 函数说明

表 2.7 展示了翻译模块的相关函数及其功能。

表 2.7 翻译模块函数说明

函数名	功能
translate_rdtime_return_zero	和原生 qemu 对比时，确保 rdtime.d 总是返回 0
translate_pcaddi	pcaddi 指令的翻译
translate_branch	直接跳转指令的翻译
translate_jirl	间接跳转指令的翻译
translate_syscall	系统调用指令的翻译
INS_reg_remapping	寄存器重映射
INS_free_all_itemp	释放未映射的临时通用寄存器
INS_translate	指令翻译入口函数

2.5 插桩模块

2.5.1 插桩工具加载

在本插桩框架的使用过程中，需要将用户编写完成的 Pintool 加载到 Qemu 的主流程中，从而完成分析函数的插入工作。本插桩工具在使用过程中，需要类似 Qemu 的 user-mode 来添加命令行参数指定 Pintool。在 Qemu 的 main 函数中有一系列解析命令行参数的工具，在解析参数的过程中会通过调用 load_pintool 函数来完成 Pintool 的加载工作。

load_pintool 函数会从编译完成的 Pintool 动态链接库中加载该 Pintool 的 main 函数，并执行该 main 函数。插桩工具（Pintool）的加载过程，实际上就是在 main 函数中进行分析函数的注册，目的是将用户编写的分析函数的指针保存到结构体 PIN_STATE 的相应字段中。Pintool 可以使用的分析函数指针类型如表

2.8 所示。在加载过程中，会使用表 2.9 中的相应注册函数，来保存分析函数的指针。

表 2.8 分析函数指针

函数指针名	参数表
INS_INSTRUMENT_CALLBACK	INS ins, VOID* v
TRACE_INSTRUMENT_CALLBACK	TRACE trace, VOID *v
FINI_CALLBACK	INT32 code, VOID *v
SYSCALL_ENTRY_CALLBACK	THREADID threadIndex, CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v
SYSCALL_EXIT_CALLBACK	THREADID threadIndex, CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v
CPU_EXEC_ENTER_CALLBACK	CPUState *cpu, TranslationBlock *tb
CPU_EXEC_EXIT_CALLBACK	CPUState *cpu, TranslationBlock *last_tb, int tb_exit
IMAGECALLBACK	IMG img, VOID *v

表 2.9 注册函数表

函数名	参数
PIN_InitSymbols	void
INS_AddInstrumentFunction	INS_INSTRUMENT_CALLBACK fun, VOID* val
TRACE_AddInstrumentFunction	TRACE_INSTRUMENT_CALLBACK fun, VOID *val
IMG_AddInstrumentFunction	IMAGECALLBACK fun, VOID *val
PIN_AddFiniFunction	FINI_CALLBACK fun, VOID *val
PIN_AddSyscallEntryFunction	SYSCALL_ENTRY_CALLBACK fun, VOID *val
PIN_AddSyscallExitFunction	SYSCALL_EXIT_CALLBACK fun, VOID *val
PIN_AddCpuExecEnterFunction	CPU_EXEC_ENTER_CALLBACK fun, VOID *val
PIN_AddCpuExecExitFunction	CPU_EXEC_EXIT_CALLBACK fun, VOID *val

2.5.2 插桩接口

本文在第 3 章中，将介绍 INS、BBL 和 TRACE 三种粒度的插桩接口，在第 4 章中将介绍 RTN、IMG 两种粒度的插桩接口，因此在此处不做额外赘述。

2.6 代码生成模块

插桩模块完成不同粒度的插桩工作后，会调用代码生成模块，将插桩后的 Loongarch 指令序列重新生成二进制代码。生成的代码被保存在代码缓存中，用于之后被调度执行模块调用。

2.6.1 函数说明

表 2.10 展示了代码生成模块中的函数及功能。

表 2.10 代码生成模块函数说明

函数名	功能
la_assemble	根据 Ins 汇编出龙芯指令二进制码
la_encode	将指令链表转换成二进制码并存储在代码缓存中

2.7 代码缓存模块

代码缓存模块负责保存和管理插桩前后的代码块。当同一代码块被再次访问时，调度执行模块优先向代码缓存中进行查询，若有已经插桩好的代码块，则可以避免对该代码块重新执行解析、翻译和插桩的过程，从而提高插桩框架的运行效率。本框架所使用的代码缓存功能均使用 Qemu 中已经实现的代码缓存。

3 INS、BBL 和 TRACE 级 API 的设计与实现

3.1 INS、BBL 和 TRACE 级 API 的概念

3.1.1 INS 级 API 的定义

INS 级 API 的是以一条汇编指令为插桩对象设计实现的 API。在框架中，一条原始的指令在经过代码解析模块，翻译模块和插桩模块的处理后，会生成与原始指令功能相同的一条或多条指令，每条指令由数据结构 Ins 描述，这些 Ins 共同组成了一个 INS 块。INS 块是整个框架插桩操作的最小单位，也是可插桩的最细粒度。

3.1.2 BBL 级 API 的定义

BBL级API的是以一个基本块为插桩对象设计实现的API。BBL是一个单入口单出口的指令序列，如果一个基本块的出口指向另一个基本块的中间，那么将会产生一个新的基本块。以下三种情况可以作为基本块的入口：

1. 进入程序的第一条指令。

2. 一个基本块结束后，紧跟着基本块的下一条指令。

3. 跳转指令的目标地址处的指令。

条件跳转指令，无条件跳转指令，系统调用等控制执行流的指令可作为基本块的出口。基本块是程序能够连续执行的最小指令序列。

3.1.3 TRACE 级 API 的定义

TRACE 级 API 的是以一个轨迹为插桩对象设计实现的 API。TRACE 是程序能够连续执行的最大指令序列。在本节中将参考英特尔 Pin 对 TRACE 级粒度的定义，描述本团队对项目插桩框架 TRACE 级粒度定义的探索过程。

Intel Pin 官方对 TRACE 的官方定义是：TRACE 通常是一个单入口，多出口的一系列指令序列的集合。它通常开始于跳转指令的目标地址处的指令，并且以一个无条件跳转指令为结束。TRACE 由许多的 BBL 组成，如果有一条跳转指令的目标地址在 TRACE 中间，那么将会在原有的 BBL 基础上产生新的 BBL，同

时也产生新的 TRACE。

TRACE 是以跳转指令的目标为入口，以无条件跳转指令为出口的。无条件跳转指令在 x86 架构下包括 syscall，jmp，call，ret 等。在这个前提条件下编写了一个测试程序 hello.cpp，之后使用 Intel Pin 编写 Pintool 对该程序进行插桩，该 Pintool 的功能是可以打印程序所有的 TRACE 以及它所包括的 BBL，同时还能将 BBL 中每条指令的详细信息打印出来。第一次测试结果的节选如图 3.1 所示。

图 3.1 展示了打印出的 TRACE 的详细信息。如图所示，该 TRACE 的入口符合官方的定义，出口为一条条件跳转指令，与官方定义不符。同时还发现，相邻打印出的两个 TRACE 在地址上并不相邻。出现这种现象，可能是因为整个程序的插桩和执行是动态的，每插桩完一个 TRACE 块，就会执行该 TRACE 块，所以打印出的结果不是代码层面相邻的 TRACE，而是程序运行时相邻执行的两个 TRACE。基于上述原因，才有地址不相邻的输出结果。

```
The address of the last ins in the 3 trace is:140555487211193
The ins of the 1 bbl in the 3 trace is:
140555487211167 : cmp rax, 0x22
140555487211171 : jnbe 0x7fd59fe2ee78
The ins of the 2 bbl in the 3 trace is:
140555487211173 : mov qword ptr [rcx+rax*8], rdx
140555487211177 : mov rax, qword ptr [rdx+0x10]
140555487211181 : add rdx, 0x10
140555487211185 : test rax, rax
140555487211188 : jnz 0x7fd59fe2ee9f
The ins of the 3 bbl in the 3 trace is:
140555487211190 : test r12, r12
140555487211193 : jz 0x7fd59fe2ef3f
```

图 3.1 hello.c 插桩测试结果图

根据 Intel Pin 的描述，TRACE 是能连续执行的最长指令序列，因此结束指令序列连续执行的指令都可作为 TRACE 的结尾。无条件跳转指令一定会结束指令序列的顺序执行，因此无条件跳转指令可以作为 TRACE 的结尾；而当一个条件跳转指令满足跳转条件，也会结束指令序列的顺序执行，因此也可以作为 TRACE 的结尾。由于在 TRACE 块解析生成阶段并没有实际执行 TRACE 块，

不能得知是否满足条件跳转指令的跳转条件，因此在以当前指令为入口开始 TRACE 的生成后，每当遇见一个条件跳转指令，就会以这条指令为结尾生成并缓存一个新的 TRACE，直到遇见一条无条件跳转的控制流指令，以当前指令为入口的 TRACE 才全部生成完毕。这样操作之后会得到许多入口相同，出口不同的 TRACE 块。编写汇编程序 test.asm 进行测试，程序如图 3.2 所示，预期测试结果会打印出起始指令相同但结束指令不同的 5 个 TRACE 块。

测试结果如图 3.3 所示，不满足预期。观察发现每个以条件跳转结束的 TRACE 都有 3 个 BBL，且没有超过 3 个 BBL 的 TRACE 出现。于是提出最终猜想：如果一个正在生成的 TRACE 遇到了无条件跳转指令，则结束 TRACE，但若指令序列中一直没有出现无条件跳转指令，同时 BBL 数量已达到了 3，，此时也结束这个 TRACE 的生成，并以 PC 中的指令为开始，开启下一个 TRACE 的生成。在这个猜想的基础上，编译了新的测试用 Pintool，这次 Pintool 的主要功能是在之前的 Pintool 功能基础上，额外统计每个 TRACE 中的 BBL 数量，如果所有 TRACE 中 BBL 数都小于等于 3，并且拥有小于 3 个 BBL 的 TRACE 都是由无条件跳转指令结尾的，那么就在输出的最后输出“you are right!”。最终结果满足预期。

```
section .data
    target_count dd 3

section .text
    global _start

_start:
    mov eax,1

target_loop:
    add eax,1
    cmp eax,2         ; 比较ax和2
    je target_loop     ; 如果相等，跳回target_loop
    cmp eax,3         ; 比较ax和3
    je target_loop     ; 如果相等，跳回target_loop
    cmp eax,4         ; 比较ax和4
    je target_loop     ; 如果相等，跳回target_loop
    cmp eax,5         ; 比较ax和5
    je target_loop     ; 如果相等，跳回target_loop
```

图 3.2 测试用汇编代码

```
The address of the first ins in the 1 trace is:4198400
The address of the last ins in the 1 trace is:4198421
The ins of the 1 bbl in the 1 trace are:
4198400 : mov eax, 0x1
4198405 : add eax, 0x1
4198408 : cmp eax, 0x2
4198411 : jz 0x401005
The ins of the 2 bbl in the 1 trace are:
4198413 : cmp eax, 0x3
4198416 : jz 0x401005
The ins of the 3 bbl in the 1 trace are:
4198418 : cmp eax, 0x4
4198421 : jz 0x401005

The address of the first ins in the 2 trace is:4198405
The address of the last ins in the 2 trace is:4198421
The ins of the 1 bbl in the 2 trace are:
4198405 : add eax, 0x1
4198408 : cmp eax, 0x2
4198411 : jz 0x401005
The ins of the 2 bbl in the 2 trace are:
4198413 : cmp eax, 0x3
4198416 : jz 0x401005
The ins of the 3 bbl in the 2 trace are:
4198418 : cmp eax, 0x4
4198421 : jz 0x401005

The address of the first ins in the 3 trace is:4198423
The address of the last ins in the 3 trace is:4198435
The ins of the 1 bbl in the 3 trace are:
4198423 : cmp eax, 0x5
4198426 : jz 0x401005
The ins of the 2 bbl in the 3 trace are:
4198428 : mov eax, 0x1
4198433 : xor ebx, ebx
4198435 : int 0x80
```

图 3.3 test asm 测试结果

上述过程完成了对 TRACE 基本结构的定义，而对官方定义的后半部分的理解还存在问题。后半部分描述道，如果一条跳转指令的目标地址指向基本块的中间，则产生一个新的 TRACE，同时产生一个新的 BBL，为方便后面描述，称这个跳转指令目标地址处的指令为 middle 指令。对于产生新的 BBL 的过程，如图 3.6 所示，有两种猜想：一、依据 BBL 的定义，由于 BBL 都是单入口单出口的，于是产生是指分裂产生，如细胞分裂一般，变成两个基本块，一个基本块以原起始指令为开始，以 middle 指令前一条指令为结束的基本块，另一个基本块以 middle 指令为开始，以原基本块最后一条指令为结束的基本块，这两个基本块完全不相交。二、不会引起 BBL 块的分裂，而是从 middle 指令开始产生一个新的基本块，同时以该基本块为起始产生新的 TRACE，两个基本块都以原基本块的最后一条指令结束，这两个 BBL 块虽有重叠，但是两个完全不同的基本块，属于两个不同的 TRACE。在有上述猜想的基础上编写 Pintool 进行测试，测试结果如图 3.7 所示，实验结果支持猜想二。

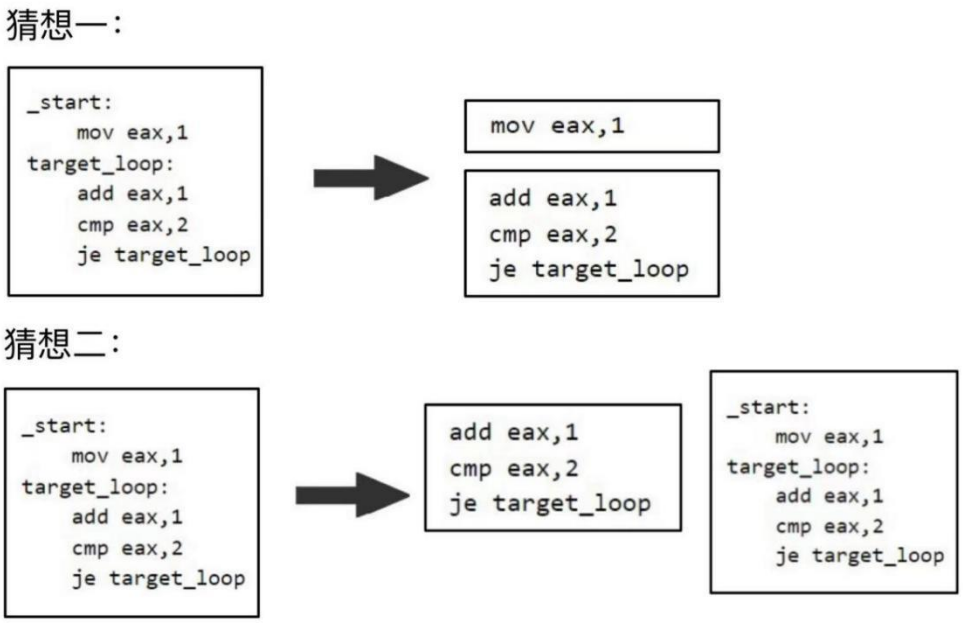

图 3.6 BBL 生成猜想图

经过上述的探索，本团队得到 TRACE 准确且详细的定义：TRACE 通常是一个单入口多出口的指令序列，入口是上一个 TRACE 解析完此时 PC 中的地址，出口有两种情况：1.无条件跳转指令。2.TRACE 中第三个基本块最后的条件跳转指令。一个 TRACE 中的基本块数不大于 3。如果有一条跳转指令的目标地址在 TRACE 中间，那么将会在原有的 BBL 基础上产生新的 BBL，同时也产生新的 TRACE。这个产生过程并不是如细胞分裂一样产生两个完全不相交的 BBL，而是另外产生一个以 middle 指令为第一条指令的基本块。

```
4198400 : mov eax, 0x1
4198405 : add eax, 0x1
4198408 : cmp eax, 0x2
4198411 : jz 0x401005
The ins of the 2 bbl in the 1 trace are:
4198413 : cmp eax, 0x3
4198416 : jz 0x401005
The ins of the 3 bbl in the 1 trace are:
4198418 : cmp eax, 0x4
4198421 : jz 0x401005

The address of the first ins in the 2 trace is:4198405
The address of the last ins in the 2 trace is:4198421
The ins of the 1 bbl in the 2 trace are:
4198405 : add eax, 0x1
4198408 : cmp eax, 0x2
4198411 : jz 0x401005
The ins of the 2 bbl in the 2 trace are:
4198413 : cmp eax, 0x3
4198416 : jz 0x401005
The ins of the 3 bbl in the 2 trace are:
4198418 : cmp eax, 0x4
4198421 : jz 0x401005
```

图 3.7 BBL 生成验证结果

3.1.5 INS、BBL 和 TRACE 的数据结构

根据前文对 TRACE 定义的探索，在 INS 和 BBL 定义的基础上，完成了对 TRACE 的定义，结构体如图 3.8 所示。INS 的属性包括指向下一个 INS 的指针、指向上一个 INS 的指针、地址、操作码、原始指令、第一条指令、最后一条指令、指令条数、插入 INS 块前的分析函数指针链表和插入 INS 块后的分析函数指针链表。

BBL 的属性包括地址、包含的 INS 个数、第一个 INS、最后一个 INS、指向下一个 BBL 的指针和指向上一个 BBL 的指针。

TRACE 的属性包括地址、包含的 BBL 个数、包含的 INS 个数、所属函数、第一个 BBL 和最后一个 BBL。

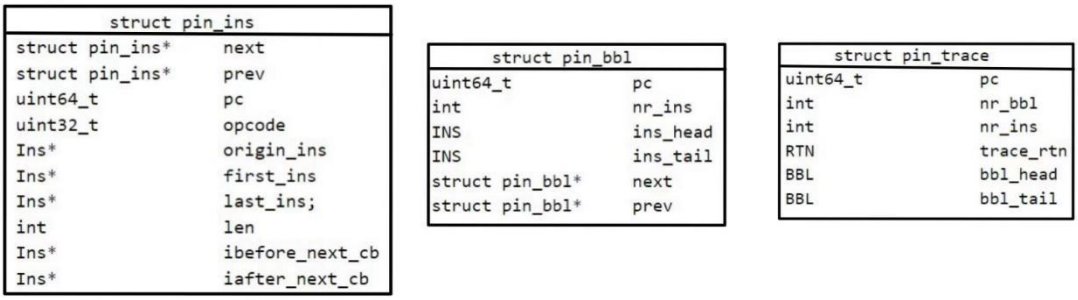

图 3.8 INS、BBL 和 TRACE 结构体定义图

3.1.6 INS、BBL 和 TRACE 的解析

如图 3.9 所示，首先根据 PC 查找代码缓存模块，若目标 TRACE 块已经在代码缓存中，则直接从代码缓存中取得代码块执行；如果不在则从当前 PC 开始逐条进行指令的解析和翻译，一条指令经过翻译后会得到多条指令，这些指令组成一个 INS。从第一个 INS 开始 BBL 的生成，解析到跳转指令或者系统调用时结束一个 BBL。将 INS 和 BBL 添加到当前 TRACE 中。在翻译的过程中如果遇见无条件跳转指令则结束当前 TRACE 的生成，如果是条件跳转指令，则判断当前 TRACE 中 BBL 的数量，如果小于 3 则结束只当前 BBL 的生成，继续解析后续指令，如果 TRACE 中的 BBL 数是 3 则也结束当前 TRACE 的生成。对得到

的 TRACE 进行指令插桩，将插桩后的 TRACE 存入代码缓存，从而完成对一个 TRACE 的解析、翻译和插桩工作。

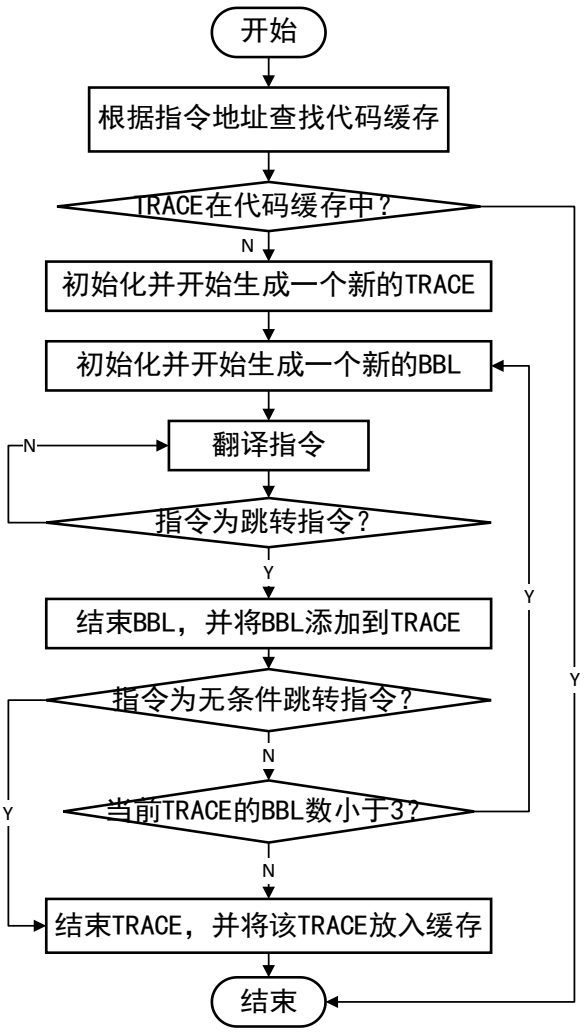

图 3.9 TRACE 生成流程图

3.2 API 的设计与实现

3.2.1 API 实现相关结构体

分析函数相关信息都是存储在 ANALYSIS_CALL 结构体中的，结构体以及其成员结构体的定义如图 3.10 所示。在框架中，同一位置可能会同时插桩多个不同的分析函数，这些分析函数被定义成了回调函数（callback_fun，后面简称 cb），为了保证 cb 的正确执行，按照插桩的前后顺序，将他们以链表的存储方

式连接在了一起。由于是对二进制代码的插桩，所以各粒度的插桩工作，从本质上是找到准确的插桩位置后，对一条汇编指令的插桩，即 INS 级别的插桩，因此存储上述 cb 链表的数据结构放在了 INS 的结构体中，INS 结构体的定义如图 3.8 所示。

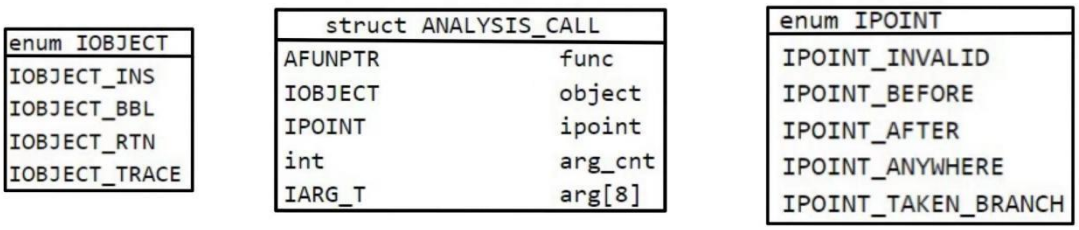

图 3.10 ANALYSIS_CALL 结构体及其包含成员定义图

框架当前的插桩信息都是储存在 PIN_STATE 结构体类型的全局变量 pin_state 中的，结构体的定义如图 3.12 所示。

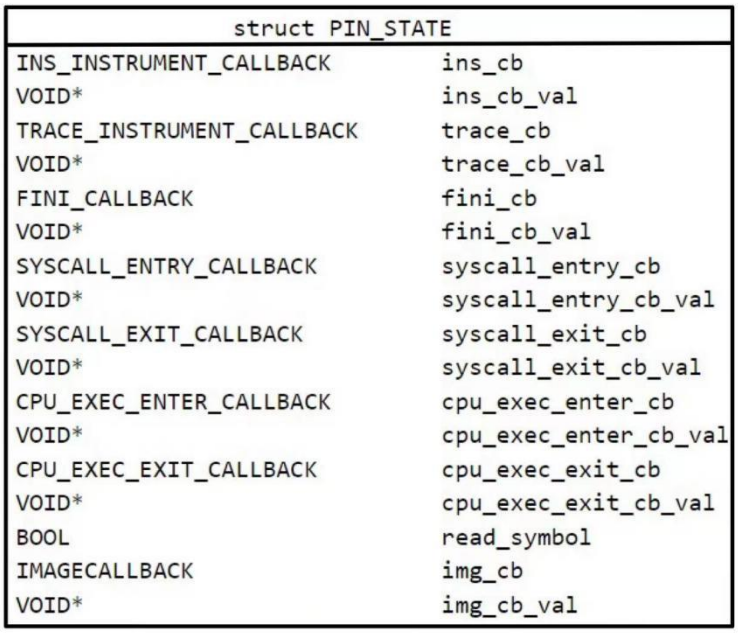

图 3.12 PIN_STATE 结构体成员

3.2.2 INS 级 API 的设计与实现

INS 级 API 主要分为两类，一类是插桩型 API（Instrumentation API）：包括向框架声明插桩粒度的 API 和向代码中插入分析函数的 API；另一类是信息检查型 API（Inspection API）：用于获取 INS 级别相关信息的 API。设计和实现 INS

级 API 共计个，其中 7 个为插桩型 API，55 个为信息检查型 API。

1. 插桩型 API

INS_InsertCall()：将函数 funptr 插桩在 INS 的特定位置，分别传递要插桩的 INS，插桩的位置（action），插入的分析函数（funptr）以及一个不定长的参数列表，该列表用于传递分析函数的参数。其中插桩位置有指令前（IPOINT_BEFOR)，指令后（IPOINT_AFTER），任意位置（IPOINT_ANYWHERE）和控制流指令后（IPOINT_TAKEN_BRANCH）四种。这里 IPOINT 是一个枚举类型，决定了分析函数的调用被插入到什么地方。插桩的对象可以是 INS、BBL、TRACE 和 RTN。下面对上述四个值进行解释：

IPOINT_BEFORE：在插桩对象的第一条指令之前插入分析函数的调用总是有效的。

IPOINT_AFTER：在插桩对象的最后一条指令的失败路径处插入分析函数的调用。对 INS，仅在 INS_IsValidForIpointAfter()函数为真的情况下适用；对 BBL，仅在 BBL_HasFallThrough()函数为真的情况下适用；对 TRACE，仅在 TRACE_HasFallThrough()函数为真的情况下适用；对 RTN，在所有返回路径处插桩。

IPOINT_ANYWHERE：在插桩对象的任意位置插入分析函数的调用，不适用于 INS_InsertCall()和 INS_InsertThenCall()函数。官方对 IPOINT_ANYWHERE 的解释是通常情况插在指令之前，但有时为了更高的运行效率可能插在其他的位置，在本框架的实现上默认都是指插在指令前，只是为了兼容 Intel Pin 的 API 保留了这个成员。

IPOINT_TAKEN_BRANCH：在插桩对象的控制流的执行边界处插入分析函数的调用，仅适用于 INS_IsValidForIpointTakenBranch()函数为真的情况。

实现 INS_InsertCall()函数功能的流程图如图 3.13 所示。动态二进制插桩主要是对二进制代码进行插桩工作，在需要插桩的位置，插入对函数的调用。为了确保不影响程序的执行结果，从整体上首先应该保存上下文，记录当前程序的执行状态，然后插入对函数的调用，跳转到执行分析函数的位置，在执行完分析函数跳回之后恢复上下文，继续原本程序的执行。

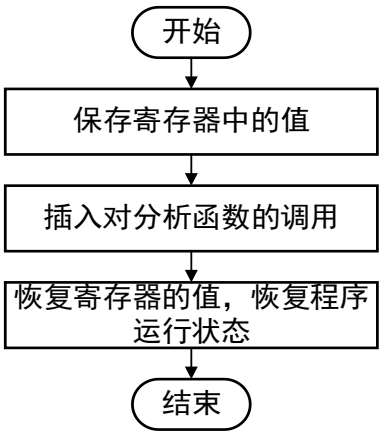

图 3.13 插桩函数实现流程图

细节上，首先初始化 cb 的参数列表，再利用 parse_iarg()解析参数列表，并将他们储存进 cb 中的参数结构体中。之后通过 ins_get_next_cb_position()函数找到插入的准确位置，并把初始化的分析模块连接到链表中。进入 insert_callback ()函数进行图 3.13 的操作流程，插入对函数的调用，之后恢复寄存器状态。由此实现了对 ins 级别粒度的插桩。流程图如图 3.14 所示。

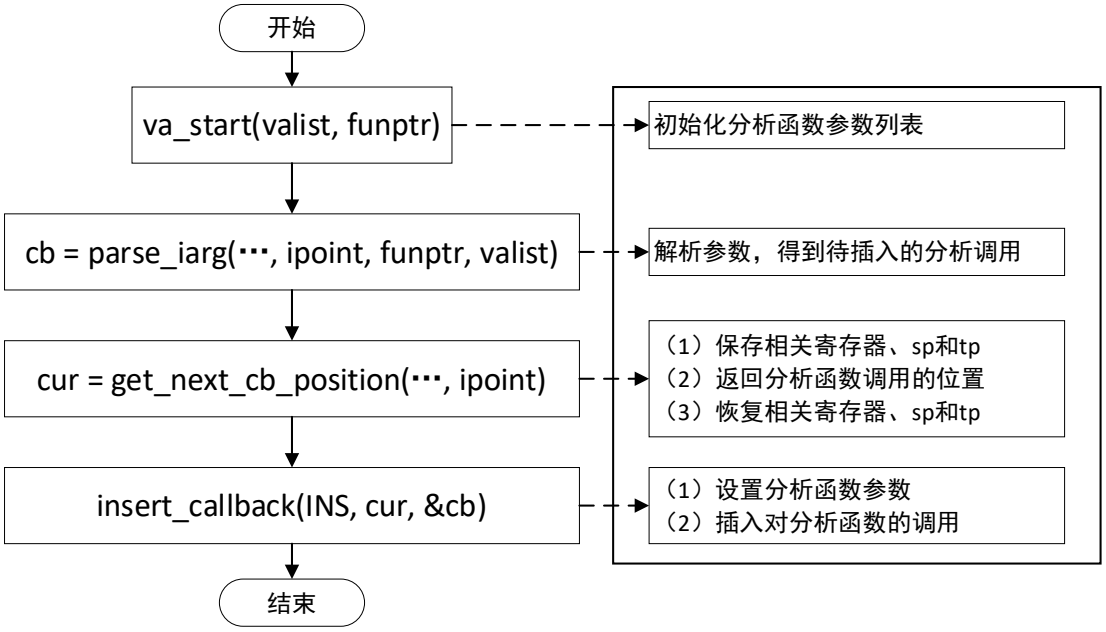

图 3.14 INS_InsertCall()流程图

2. 其他插桩型API

本团队在基础插桩API的功能基础上，增添技术细节，实现了其它插桩型API。这些API的名称及功能描述如表 3.1 所示。

表 3.1 其他 INS 级插桩 API 功能介绍表

API 名称	功能描述
INS_InsertIfCall (INS ins, IPOINT action, AFUNPTR funptr,...)	插入相对于 INS 的 funptr 用，并根据 funptr 的返回结果决定是否执行其后的 ThenCall 中的分析调用。当 funptr 的返回结果为 0 时，跳过 ThenCall 中的分析调用，当 funptr 的返回结果非 0 时，执行 ThenCall 中的分析调用。
INS_InsertThenCall(INS ins, IPOINT action, AFUNPTR funptr,...)	插入相对于 INS 的 funptr 调用，并根据 IfCall 的分析调用返回结果决定是否执行 funptr 调用。当 IfCall 中分析调用的返回值非 0 时，执行 funptr 调用。
INS_InsertPredicatedCall(INS ins, IPOINT action, AFUNPTR funptr,...)	插入相对于 INS 的 funptr 调用，并根据指令的 predicate 是否为真，决定是否执行。对于不需要判断 predicate 的指令，默认其 predicate 为 true。对于需要判断 predicate 的条件跳转指令，当其跳转条件满足时，predicate 为 false。
INS_InsertIfPredicatedCall(INS ins, IPOINT action, AFUNPTR funptr,...)	插入相对于 INS 的 funptr 调用，并根据分析函数 funptr 的返回值是否为 0 以及指令的 predicate 是否为真，决定是否执行 ThenCall 的分析调用。当指令不需要判断 predicate 时，该 API 等价于 INS_InsertIfCall。
INS_InsertThenPredicatedCall (INS ins, IPOINT action, AFUNPTR funptr,...)	插入相对于 INS 的 funptr 调用，并根据 IfCall 分析调用的返回值是否为 0 以及指令的 predicate 是否为真，决定是否执行 funptr 调用。当 IfCall 中分析调用的返回值非 0，且指令的 predicate 为真时，执行 funptr 调用。

INS_InsertIfCall、INS_InsertPredicatedCall、INS_InsertIfPredicatedCall、INS_InsertThenCall 和 INS_InsertThenPredicatedCall 中含 If 的 API 和含 Then 的 API 需成对出现，所以在 IfCall 中需设置标识，使得 ThenCall 可以正确执行，IfCall 中的插桩推迟到 ThenCall 中再插桩。为实现 API 功能，在 IfCall 插入的分析调用后再插入一条跳转指令 beqz，当所有条件都满足时，跳转指令的偏移值为 0，顺序执行 ThenCall 插入的分析调用，否则修正跳转指令偏移值，跳过 ThenCall 插入的分析调用，从而实现不执行 thencall 插入的分析函数的功能。表 3.2 展示

了为实现 API 而设置的相关标识和变量，图 3.16 为上述五个 API 的实现流程图。

表 3.2 相关标识和变量

名称	含义
ins_if_call_valid	用于判断是否出现过 IfCall 的标识
ins_predicate_valid	用于判断 IfCall 中是否出现过 Predicate 的标识
exec_If_cb	插入 IfCall 中的分析调用
reg_a0=funptr()	将 IfCall 中分析调用的返回结果保存在寄存器 a0 中
Insert_ins_if_jump	根据 a0 中的值插入跳过 ThenCall 中分析调用的指令
exec_then_cb	插入 ThenCall 的分析调用
Insert_ins_pre_jump	根据指令 predicate 插入跳过 ThenCall 中分析调用的指令
Fix_pre_offset	修正根据 predicate 插入的跳转指令的偏移值
Fix_if_offset	修正根据寄存器 a0 插入的跳转指令的便宜值
exec_predicate_cb	插入 PredicateCall 中的分析调用

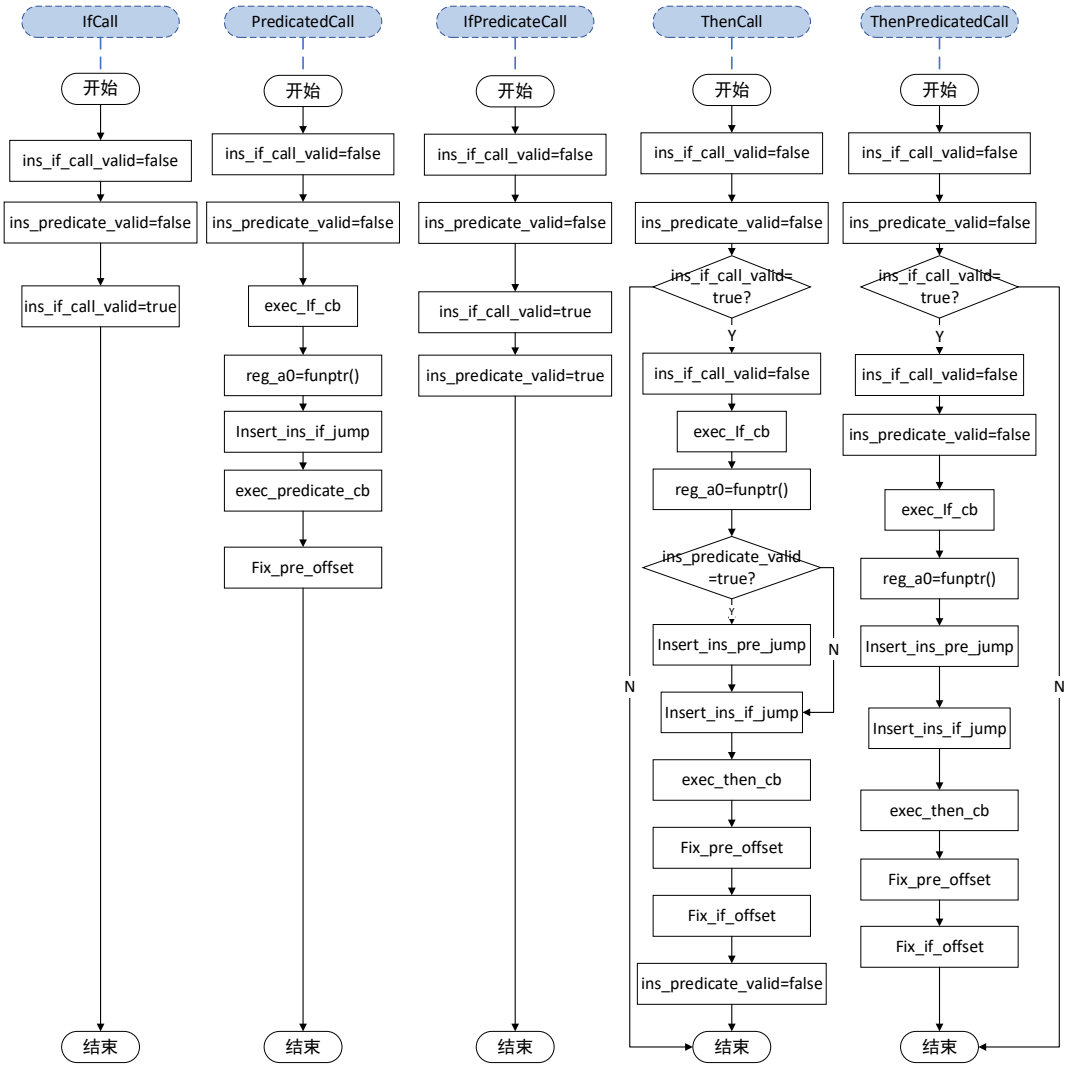

图 3.16 INS 级插桩型 API 实现流程图

3. 信息检查型API

由于框架对程序的底层封装，底层二进制指令信息对用户不透明。为满足用户获取底层信息的需求，本团队添加了信息检查API。API的功能表如表 3.3 所示。为了精确的对特定指令插桩，用户需要能够检查基本块中的每一条指令及其相关的信息，例如指令类型、操作数类型、访存地址及大小等，以判断是否为想要插桩的指令。此外，用户也可能需要在分析函数执行时记录执行到的指令的相关信息。

表 3.3 INS级信息检查型API功能介绍表

API 名称	功能
INS_EffectiveAddressWidth (INS ins)	获取指令的有效地址位宽
INS_GetPredicate (INS ins)	获取指令的 predicate
INS_IsMemoryRead (INS ins)	判断指令是否读内存
INS_IsMemoryWrite (INS ins)	判断指令是否写内存
INS_HasFallThrough (INS ins)	判断指令是否是更改控制流的指令，对于大多数指令和条件分支，返回 true，对于无条件分支和调用，返回 false
INS_Mnemonic (INS ins)	获取指令名称
INS_IsBranch (INS ins)	判断指令是否是分支指令
INS_IsDirectBranch (INS ins)	判断指令是否是直接分支指令
INS_IsDirectCall (INS ins)	判断指令是否是直接调用指令
INS_IsDirectControlFlow (INS ins)	判断指令是否是直接控制流指令
INS_IsCall (INS ins)	判断指令是否是函数调用或函数返回指令
INS_IsControlFlow (INS ins)	判断指令是否是控制流指令
INS_IsValidForIpointAfter (INS ins)	判断能否在指令后插入分析调用
INS_IsValidForIpointTakenBranch (INS ins)	判断能否在指令控制流边界处插入分析调用
INS_IsProcedureCall (INS ins)	判断指令是否是过程调用指令，过程调用指令特点就是会保存下条指令的 PC,LA 中 bl 和写 ra 寄存器的 jirl 是过程调用指令
INS_IsInterrupt (INS ins)	判断指令是否是系统调用或中断指令
INS_IsRet (INS ins)	判断指令是否是函数返回指令
INS_IsPrefetch (INS ins)	判断指令是否是加载内存数据到 Cache 的指令

续表 3.3 INS 级信息检查型 API

API 名称	功能
INS_IsSub (const INS ins)	判断指令是否是 Sub 指令
INS_IsMov (const INS ins)	判断指令是否是 Mov 指令
INS_IsAtomicUpdate (const INS ins)	判断指令是否是 AM*原子访存指令
INS_IsIndirectControlFlow (INS ins)	判断指令是否是间接控制流指令
INS_Opcode (INS ins)	获取指令操作码类型
INS_IsOriginal (INS ins)	判断指令是否是应用程序的原始指令
INS_Disassemble (INS ins)	获取汇编指令字符串
INS_ChangeReg (const INS ins, const REG old_reg, const REG new_reg, const BOOL as_read)	判断指令中是否有访问方式为 as_read 的 old_reg 被更改为 new_reg
INS_OperandCount (INS ins)	获取指令操作数个数
INS_OperandIsMemory (INS ins, UINT32 n)	判断指令的第 n 个操作数是否是内存相关的操作数，n 从 0 开始
INS_MemoryOperandCount (INS ins)	获取指令内存操作数个数
INS_MemoryOperandSize (INS ins, UINT32 memoryOp)	获取指令内存操作数字节大小
INS_MemoryOperandIsRead (INS ins, UINT32 memopIdx)	判断是否是读内存指令
INS_MemoryOperandIsWritten (INS ins, UINT32 memopIdx)	判断是否是写内存指令
INS_MemoryOperandIndexToOperandIndex (INS ins, UINT32 memopIdx)	将内存操作数的索引号转换为在指令中的操作数索引号
INS_OperandIsReg (INS ins, UINT32 n)	判断第 n 个操作数是否是寄存器操作数

续表 3.3 INS 级信息检查型 API

API 名称	功能
INS_OperandReg (INS ins, UINT32 n)	若第 n 个操作数为寄存器，返回该寄存器
INS_OperandIsImmediate (INS ins, UINT32 n)	判断第 n 个操作数是否是立即数操作数
INS_OperandImmediate (INS ins, UINT32 n)	若第 n 个操作数为立即数，返回该立即数
INS_OperandIsImplicit (INS ins, UINT32 n)	判断指令的第 n 个操作数是否是隐式操作数
INS_RegIsImplicit (INS ins, REG reg)	判断寄存器 reg 是否是指令中的隐式操作数
INS_OperandRead (INS ins, UINT32 n)	判断操作数的访问方式是否为读/读写
INS_OperandWritten (INS ins, UINT32 n)	判断操作数的访问方式是否为写/读写
INS_OperandReadOnly (INS ins, UINT32 n)	判断操作数的访问方式是否只为读
INS_OperandWrittenOnly (INS ins, UINT32 n)	判断操作数的访问方式是否只为写
INS_OperandReadAndWritten (INS ins, UINT32 n)	判断操作数的访问方式是否为读写
INS_IsSyscall (INS ins)	判断指令是否是系统调用指令
INS_Next (INS x)	获取下一条指令
INS_Prev (INS x)	获取上一条指令
INS_Invalid ()	获取一条空指令
INS_Valid (INS x)	判断指令是否有效
INS_Address (INS ins)	获取指令地址
INS_Size (INS ins)	获取指令字节大小

续表 3.3 INS 级信息检查型 API

API 名称	功能
INS_DirectControlFlowTargetAddress (INS ins)	获取直接控制流指令的目标地址
INS_NextAddress (INS ins)	获取下一条指令地址
INS_IsAddedForFunctionReplacement (INS ins)	判断指令是否被 Pin 为了功能替换而插入了指令
INS_Delete (INS ins)	删除指令

3.2.3 BBL 级 API 的设计与实现

设计和实现 BBL 级 API 共计 13 个，其中 4 个为插桩型 API，用于以 BBL 为单位向代码中插入分析函数，9 个为信息检查型 API，用于获取 BBL 的相关信息。

1. 插桩型 API

BBL 级插桩型 API 的名称及功能描述如表 3.4 所示。

表 3.4 BBL 级插桩型 API

API 名称	功能
BBL_InsertCall()	向框架声明分析函数，传递分析函数的参数，完成对分析函数的注入工作
BBL_InsertIfCall()	根据分析调用的返回结果判断是否执行 BBL_InsertIfCall()中的分析调用
BBL_InsertThenCall()	根据 BBL_InsertIfCall()中的分析调用返回结果判断是否执行分析调用，当 BBL_InsertIfCall()返回结果不为 0 时，执行分析调用

2. 信息检查型API

BBL级侦查型API的名称及功能描述如表 3.5 所示。

表 3.5 BBL级信息检查型API功能表

API 名称	功能
BBL_InsHead()	返回 BBL 中的第一个 INS
BBL_InsTail()	返回 BBL 中的最后一个 INS
BBL_Valid()	判定 BBL 是否有效
BBL_NumIns()	返回 BBL 中的 INS 的数量
BBL_Address()	返回当前 BBL 所在的地址
BBL_Next()	返回当前 BBL 下一个 BBL
BBL_HasFallThrough ()	判断 bbl 的最后一条指令是否是一个 fall-through 指令，若是则返回 true。其中 fall-through 指令是指可以顺序执行下一条指令的指令，即不影响控制流的指令。
BBL_Size()	返回当前 BBL 所占字节数
BBL_Prev()	返回当前 BBL 上一个 BBL

3.2.4 TRACE 级 API 的设计与实现

设计和实现 TRACE 级 API 共计 12 个，其中 4 个为插桩型 API，用于以 TRACE 为单位向代码中插入分析函数，8 个为信息检查型 API，用于获取 TRACE 的相关信息。

1. 插桩型 API

TRACE 级插桩型 API 的功能表如表 3.6 所示。

表 3.6 TRACE 级插桩型 API

API 名称	功能
TRACE_InsertCall()	向框架声明分析函数，传递分析函数的参数，完成对分析函数的注入工作

续表 3.6 TRACE 级插桩型 API

API 名称	功能
TRACE_InsertIfCall()	根据分析调用的返回结果判断是否执行 TRACE_InsertIfCall()中的分析调用
TRACE_InsertThenCall()	根据 TRACE_InsertIfCall()中的分析调用返回结果判断是否执行分析调用，当 TRACE_InsertIfCall()返回结果不为 0 时，执行分析调用

2. 信息检查型 API

TRACE 级信息检查型 API 的功能表如表 3.7 所示。

表 3.7 TRACE 级信息检查型 API 功能介绍表

API 名称	功能
TRACE_BblHead(TRACE trace)	返回 TRACE 中的第一个 BBL
TRACE_BblTail(TRACE trace)	返回 TRACE 中的最后一个 BBL
TRACE_NumBbl(TRACE trace)	返回 TRACE 中的 BBL 的数量
TRACE_NumIns(TRACE trace)	返回 TRACE 中的 INS 的数量
TRACE_Address(TRACE trace)	返回当前 TRACE 所在的地址
TRACE_Rtn(TRACE trace)	返回当前 TRACE 所属的 RTN
TRACE_HasFallThrough (TRACE trace)	判断 TRACE 的最后一条指令是否是一个 fall-through 指令，若是则返回 true。其中 fall-through 指令是指可以顺序执行下一条指令的指令，即不影响控制流的指令。
TRACE_Size(TRACE trace)	返回当前 TRACE 所占字节数

4 RTN 和 IMG 级 API 的设计与实现

4.1 符号解析

4.1.1 ELF 的内容格式

启动应用程序前，会加载应用程序的 ELF 文件，可以通过对 ELF 文件进行符号（Symbol）解析获取程序所包含的函数信息。ELF 的内容包括 ELF Header、Program Header Table、多个 Section 以及 Section Header Table。其中 ELF Header 用于描述该对象文件的各项信息，Program Header Table 则是一个 Program header 数组，每个 Section 的内容不同，字节长度也不一样，Section Header Table 是一个 Section Header 数组，所有 Section Header 都等长，Section Header 包含了其对应 Section 的各项元数据。ELF 具体内容格式如图 4.1 所示。

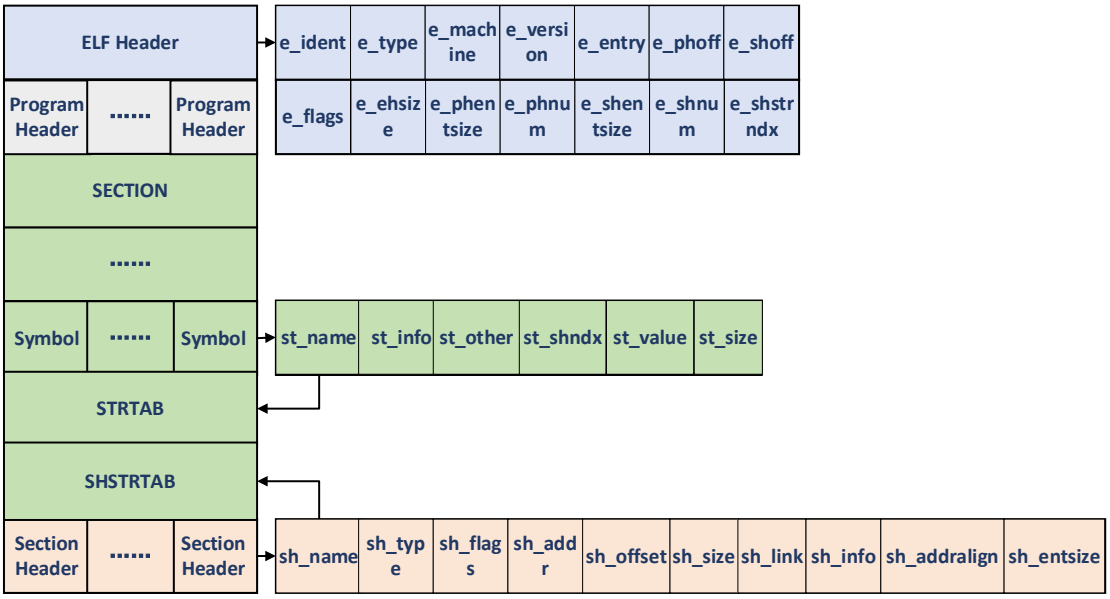

图 4.1 ELF 的内容格式图

其中有几个特殊的 Section 分别是 symtab、dynsym、strtab。symtab 和 dynsym 都是符号表，symtab 中包含了程序或库中的所有符号信息，这些符号在运行时不确定是否会用到，dynsym 则只包含了在运行时动态链接所需的符号，是一个精简的符号表。这里仅需要获取运行时信息，所以只需要 dynsym 中的符号信息就足够了。strtab 则存储了所有 Symbol 的符号名，该 Section 可以通过符号表的

符号名所在节块序号（st_name）或者符号表对应的 Section Header 中的节块链接（sh_link）找到。

4.1.2 符号解析的流程

如图 4.2 所示，符号解析的具体流程为：

（1）读取 ELF 文件的 ELF Header，根据文件类型（e_type）判断文件是否为共享文件或可执行文件；

（2）读取 Section Headers，根据节块的类型（sh_type）寻找 symtab 和 dynsym，这两个 Section 中都包含了所需 Symbol 的信息，找到其中一个即可。获取其 Section 序号。根据节块链接（sh_link）寻找 strtab，该 Section 包含了所有所需 Symbol 的名字。

（3）读取所有 Symbol 的名字和信息，收集满足条件的 Symbol 作为函数。

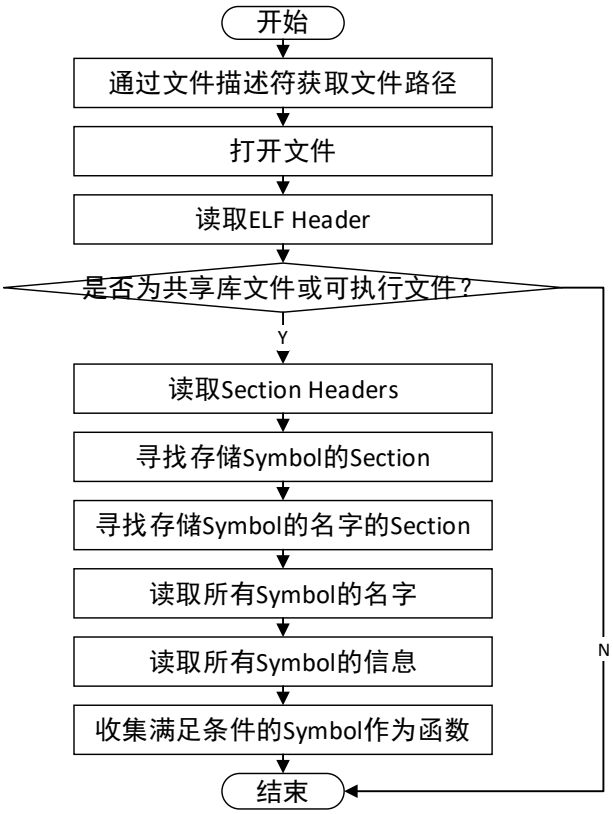

图 4.2 符号解析流程图

4.1.3 符号解析的代码实现

主要变量的数据结构如表 4.1 所示。

表 4.1 符号解析主要变量数据结构表

变量名	数据类型	变量描述
shnum	int	表示 Section Header 的个数
shsz	size_t (unsigned long)	表示 Section Header 的大小
secsz	uint64_t (unsigned long)	表示 Section 的大小
sym_idx	int	所有 Symbol 所在 Section 的索引
str_idx	int	所有 Symbol 名所在 Section 的索引
syms	Elf64_Sym *	用于存储所有 Symbol 的信息
strs	char *	用于存储所有 Symbol 的名字
nsyms	int	获取的 Symbol 的个数

符号解析的伪代码实现如图 4.3 所示，具体代码实现见仓库文件./target/loongarch/elf/new_elf_parser.c。

```
1   fd = open (pathname)
2   // 读取 ELF Header
3   pread (fd, ehdr)
4   // 根据 ELF 文件类型判断是否读取 Section Headers
5   if ehdr->e_type == ET_DYN || ET_EXEC do
6       shsz = ehdr->e_shnum * ehdr->shentsize
7       pread (fd, shdr, shsz)
8   end
9   // 根据 Section 类型寻找符号表
10  for i in ehdr->e_shnum do
11      if shdr[i].sh_type == SHT_SYMTAB || SHT_DYNSYM do
12          sym_idx = i
13          str_idx = shdr[i].sh_link
14      end
15  end
16  // 读取符号名和符号信息
17  secsz = shdr[str_idx].sh_size
18  pread (fd, strs, secsz, shdr[str_idx].sh_offset)
19  secsz = shdr[sym_idx].sh_size
20  pread (fd, syms, secsz, shdr[sym_idx].sh_offset)
21  // 收集符号
22  for sym in syms do
23      if sym.st_shndx == SHN_UNDEF || sym.st_shndx >=SHN_LORESERVE
24      || ELF64_ST_TYPE(syms[i].st_info) != STT_FUNC do
25          collect sym
26      end
27  end
```

图 4.3 符号解析伪代码流程图

4.2 RTN 级 API 和 IMG 级 API 的概念和数据结构的设计

4.2.1 RTN 级 API 的定义

RTN 级 API 是以函数为插桩对象设计实现的 API，应用程序运行时需要用到大量的符号，在动态链接时产生的相关符号信息即是所需收集的函数，可以通过符号解析进行 RTN 的获取。RTN 级 API 可以获取 RTN 的各种信息，允许在特定 RTN 的入口和出口处进行插桩分析，每个 RTN 有一个入口和多个出口。

4.2.2 IMG 级 API 的定义

IMG 级 API 是以二进制可执行文件为插桩对象设计实现的 API，每加载一个可执行文件就创建一个 IMG，每个 IMG 中包含多个 RTN，IMG 级 API 可以获取 IMG 的各种信息，允许在程序运行过程中对程序进行插桩和分析。

4.2.3 RTN 和 IMG 的数据结构

RTN 的属性包括名称、地址、字节大小、RTN 编号、指向下一个 RTN 的指针、执行下一个 RTN 的指针、需在 RTN 入口处插入的分析调用组、需在 RTN 入口处插入的分析调用的个数、需在 RTN 出口处插入的分析调用组和需在 RTN 出口处插入的分析调用的个数。

IMG 的属性包括文件路径、加载地址、IMG 编号、函数组、函数个数、指向下一个 IMG 的指针和指向上一个 IMG 的指针。

RTN 和 IMG 的具体数据结构定义如图 4.4 所示。

struct pin_rtn	
const char*	name
uint64_t	addr
uint64_t	size
uint32_t	id
struct pin_rtn*	next
struct pin_rtn*	prev
ANALYSIS_CALL*	rtn_entry_cbs
int	entry_cbs_num
ANALYSIS_CALL*	rtn_exit_cbs
int	exit_cbs_num

struct pin_img	
const char*	path
uintptr_t	load_base
uint32_t	id
RTN	img_rtn
int	rtn_num
struct pin_img*	next
struct pin_img*	prev

图 4.4 RTN 和 IMG 的数据结构图

4.2.4 RTN 和 IMG 的获取

1、全局变量的数据结构如表 4.2 所示。

表 4.2 获取 RTN 和 IMG 全局变量数据结构表

变量名	数据类型	变量描述
IMG_id	int	每加载一个 IMG，IMG_id 值就加一，将 IMG_id 赋给 img->id
RTN_id	int	每加载一个 RTN，RTN_id 值就加一，将 RTN_id 赋给 rtn->id
IMG_array	struct pin_img*	用于存放 IMG 的数组
RTN_array	struct pin_rtn**	用于存放所有 IMG 的 RTN 的数组
MAX_IMG_NR	宏定义	最多存放的 IMG 个数
MAX_IMG_RTN_NR	宏定义	每个 IMG 最多存放的 RTN 个数

2、主要函数如表 4.3 所示。

表 4.3 获取 RTN 和 IMG 主要函数描述表

函数名	返回类型	函数描述
IMG_alloc	IMG	获取一个 IMG 并初始化
RTN_alloc	RTN	获取一个 RTN 并初始化
image_add_symbol	void	收集 IMG 中的所有 RTN
image_get_symbol_by_name	RTN	通过 RTN 名称获取 RTN
get_symbol_by_pc	RTN	通过 pc 地址获取 RTN
get_symbol_name_by_pc	const char*	通过 pc 地址获取 RTN 名称
print_collected_symbols	void	打印所有的 RTN

4.3 RTN 和 IMG 级 API 的设计实现

4.1.3 RTN 级 API 的设计实现

设计和实现 RTN 级 API 共计 14 个，其中 13 个为信息检查型 API（Inspection API），用于获取 RTN 的相关信息，1 个为插桩型 API（Instrumentation API），用于在函数入口和出口处进行插桩。

1、RTN 级信息检查型 API 的名称及其功能描述如表 4.4 所示，具体代码实

现见仓库文件./target/loongarch/symbols.c。

表 4.4 RTN 级信息获取 API

API 名称	功能
IMG RTN_Img(RTN rtn)	返回 RTN 所在 IMG
RTN RTN_Next(RTN rtn)	返回下一个 RTN
RTN RTN_Prev(RTN rtn)	返回上一个 RTN
RTN RTN_Invaild(void)	返回一个无效 RTN
BOOL RTN_Valid(RTN rtn)	判断 RTN 是否有效
const char * RTN_Name(RTN rtn)	返回 RTN 名
UINT32 RTN_Id(RTN rtn)	返回 RTN 编号
USIZE RTN_Size(RTN rtn)	返回 RTN 字节大小
const CHAR* RTN_FindNameByAddress(ADDRINT address)	根据地址找 RTN 名
RTN RTN_FindByAddress(ADDRINT address)	根据地址找 RTN
RTN RTN_FindByName(IMG img, const CHAR *name)	根据名字找 RTN
UINT32 RTN_NumIns(RTN rtn)	返回 RTN 的指令数
ADDRINT RTN_Address(RTN rtn)	返回 RTN 地址

2、RTN 级插桩型 API 为 RTN_InsertCall，该 API 并未真正对 RTN 进行了插桩，只是记录了函数入口和出口处待插入的分析调用，真正进行插桩操作的是_RTN_InsertCall。为实现函数插桩功能所编写的主要函数如下，具体代码实现见仓库文件./target/loongarch/ins_instrumentation.c。

（1）记录函数入口处待插入的分析调用：

static VOID RTN_add_entry_cb(RTN rtn, ANALYSIS_CALL *cb)

（2）记录函数出口处待插入的分析调用：

static VOID RTN_add_exit_cb(RTN rtn, ANALYSIS_CALL *cb)

（3）获取函数入口处待插入的分析调用：

static ANALYSIS_CALL *RTN_get_entry_cbs(uintptr_t pc, int *cnt)

（4）获取函数出口处待插入的分析调用：

static ANALYSIS_CALL *RTN_get_exit_cbs(uintptr_t pc, int *cnt)

（5）记录函数入口和出口处待插入的分析调用：

VOID RTN_InsertCall(RTN rtn, IPOINT ipoint, AFUNPTR funptr, ...)

（6）插入分析调用：

static VOID _RTN_InsertCall(INS INS, ANALYSIS_CALL *cb)

（7）根据 TRACE 的出入口是否为 RTN 的出入口，对 TRACE 插入分析调用：

VOID RTN_instrument(TRACE trace)

若通过记录函数中包含的所有指令的方式，实现对 RTN 的插桩，由于无法确定哪些函数会在程序运行过程中执行，这将会产生大量不必要的内存空间浪费，因此 RTN 级插桩 API 通过与 TRACE 建立联系实现对 RTN 的插桩。在程序运行过程中会不断解析指令翻译得到 TRACE，获取 TRACE 的第一个 BBL 中的第一条 INS，检查其是否为要函数插桩的函数入口，若是则在该条 INS 前依次插入待插入的分析调用函数；获取 TRACE 的最后一个 BBL 中的最后一条 INS，检查其是否为要函数插桩的函数出口，若是则在该条 INS 后依次插入待插入的分析调用函数，RTN 级插桩 API 的代码实现流程图如图 4.5 所示。

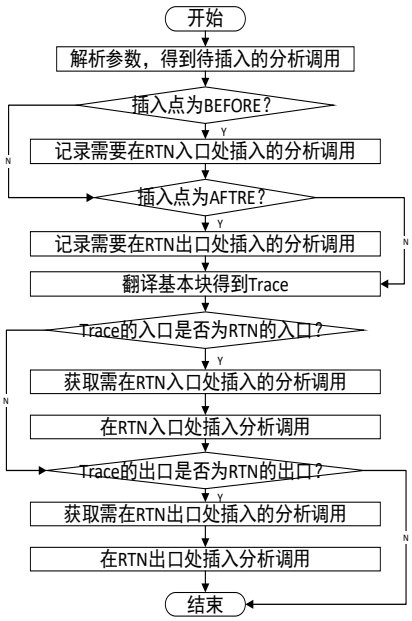

图 4.5 RTN 级 Instrumentation API 实现流程图

4.2.3 IMG 级 API 的设计实现

设计和实现 IMG 级 API 共计 10 个，其中 9 个为信息检查型 API（Inspection API），用于获取 IMG 的相关信息，1 个为注册 API，用于注册 IMG 级的插桩函数。这 10 个 API 的名称及其功能描述如表 4.5 所示，具体代码实现见仓库文件./target/loongarch/symbols.c。

表 4.5 IMG 级 API 描述

IMG 级 API	功能描述
IMG IMG_Next(IMG img)	返回下一个 IMG
IMG IMG_Prev(IMG img)	返回上一个 IMG
IMG IMG_Invalid(void)	返回一个无效 IMG
BOOL IMG_Valid(IMG img)	判断 IMG 是否有效
RTN IMG_RtnHead(IMG img)	返回 IMG 的第一个 RTN
RTN IMG_RtnTail(IMG img)	返回 IMG 的最后一个 RTN
const char * IMG_Name(IMG img)	返回 IMG 名
UINT32 IMG_Id(IMG img)	返回 IMG 编号
IMG IMG_FindImgById(UINT32)	根据 IMG 编号找 IMG
void IMG_AddInstrumentFunction (IMAGECALLBACK fun, VOID *val)	注册 RTN 级插桩函数

5 系统测试与分析

5.1 功能测试

5.1.1 INS，BBL，TRACE 定义测试

测试 TRACE，BBL，INS 的解析和生成是否满足定义。测试方法：编写测试用 Pintool，将所 TRACE 中所有的指令数据进行打印，包括 TRACE 的地址，每个 TRACE 中的 BBL 的信息，以及其中的 INS 指令信息，对打印结果进行检查。如果每个 TRACE 的 BBL 数都小于等于 3，同时拥有小于 3 个 BBL 的 TRACE 都是以无条件跳转指令或者系统调用结尾的那么就证明 TRACE 的定义正确。同时如果正确的输出了被测试程序的原始输出，则说明动态二进制插桩插入的指令并没有影响原程序的执行，证明框架各模块的工作无误。如图 5.1 是一个简单的被测试文件，图 5.2 是 Pintool 执行之后的截取的输出。输出“I’m 4!”代表插桩后的程序正确执行，TRACE 的生成和翻译块的执行都无误，满足预期。

```
int add(int i)
{
    i += 1;
    return i;
}

int main()
{
    int i = rand()%10;
    if(i == 0 )
    {
        printf("I'm 0!\n");
    }
    else
    {
        add(i);
        printf("I'm %d!\n",i);
    }
    return 0;
}
```

图 5.1 被插桩程序

```
24913   The max_insns is 512The 1650 trace is:
24914   The 1 bbl is:
24915   pc: 0x400091fd50    ldptr.w         $t0, $tp, 0xfffffe3c
24916   pc: 0x400091fd54    addi.d          $sp, $sp, 0xfffffe0
24917   pc: 0x400091fd58    st.d            $ra, $sp, 0x18
24918   pc: 0x400091fd5c    st.d            $s0, $sp, 0x10
24919   pc: 0x400091fd60    st.d            $s1, $sp, 0x8
24920   pc: 0x400091fd64    stptr.d         $s2, $sp, 0x0
24921   pc: 0x400091fd68    bnez            $t0, 0xd
24922   The 2 bbl is:
24923   pc: 0x400091fd6c    addi.w          $a7, $zero, 0x40
24924   pc: 0x400091fd70    syscall
24925
24926   The max_insns is 512The 1651 trace is:
24927   The 1 bbl is:
24928   pc: 0x400091fd6c    addi.w          $a7, $zero, 0x40
24929   pc: 0x400091fd70    syscall
24930
24931   I am 4!
24932   The max_insns is 512The 1652 trace is:
24933   The 1 bbl is:
24934   pc: 0x400091fd74    lu12i.w         $t0, 0xffffffff
24935   pc: 0x400091fd78    or              $s0, $a0, $zero
24936   pc: 0x400091fd7c    bltu            $t0, $a0, 0x18
24937   The 2 bbl is:
24938   pc: 0x400091fd80    ld.d            $ra, $sp, 0x18
24939   pc: 0x400091fd84    or              $a0, $s0, $zero
24940   pc: 0x400091fd88    ld.d            $s0, $sp, 0x10
24941   pc: 0x400091fd8c    ld.d            $s1, $sp, 0x8
24942   pc: 0x400091fd90    ldptr.d         $s2, $sp, 0x0
24943   pc: 0x400091fd94    addi.d          $sp, $sp, 0x20
24944   pc: 0x400091fd98    jirl            $zero, $ra, 0x0
24945
```

图 5.2 TRACE 定义测试结果

5.1.2 INS，BBL，TRACE 插桩 API 测试

1. 基础注册函数API

注册函数成功插入分析函数获取程序的状态信息，同时程序的输出不受影响则代表API运行成功。

如图 5.3，为利用INS_InsertCall()统计程序指令数的输出结果，上面是打印的运行过程中的指令，最后的数字为利用插桩API的输出结果，API正确。

```
[thread 572159] 2864191 pc: 0x4000b7b100        bne             $s1, $s0, 0xfffffffd
[thread 572159] 2864192 pc: 0x4000b7b104        or              $a0, $s5, $zero
[thread 572159] 2864193 pc: 0x4000b7b108        bl              0x1bff6
[thread 572159] 2864194 pc: 0x4000beb0e0        or              $a2, $a0, $zero
[thread 572159] 2864195 pc: 0x4000beb0e4        pcaddu12i       $a3, 0xce
[thread 572159] 2864196 pc: 0x4000beb0e8        ld.d            $a3, $a3, 0x62c
[thread 572159] 2864197 pc: 0x4000beb0ec        lu12i.w         $a1, 0xffffffff
[thread 572159] 2864198 pc: 0x4000beb0f0        addi.w          $a7, $zero, 0x5e
[thread 572159] 2864199 pc: 0x4000beb0f4        or              $a0, $a2, $zero
[thread 572159] 2864200 pc: 0x4000beb0f8        syscall
Total Ins Count: 2864200
```

图 5.3 指令统计输出结果

测试 TRACE 级别注册函数和基础插桩函数的正确性。测试方法：已知原框架有对基本块级别插桩的 API，编写 Pintool 分别用基本块级的插桩 API 和 TRACE 级的插桩 API 对相同的一个程序进行插桩，分别统计执行过的 TRACE 数量和 BBL 数，输出结果进行对比。在这里他们同时对一个输出“hello world!”的程序进行插桩，得到的结果如图 5.4 所示，BBL 的数量大约是 TRACE 的 2 到 3 倍，符合预期。

```
TRACE: 332466, INS: 5200497
BBL: 503916
TRACE: 1764, BBL: 3387
[cx@localhost ~]$
```

图 5.4 TRACE_InsertCall()测试结果

2. 其他插桩 API

测试 TRACE_InsertIfCall()和 TRACE_InsertThenCall()的正确性。测试方法：编写 Pintool,分析函数如图 5.5 所示，在 TRACE_InsertIfCall()的分析函数的返回值是 1 时 TRACE_InsertThenCall()才会执行，周期性设置前者的返回值，每 3 次返回一次 1，统计这两个分析函数的执行次数，预期 TRACE_InsertIfCall()是 TRACE_InsertThenCall()执行次数的 3 倍。执行结果如图 5.6 所示，满足预期。

```
static UINT64 icount = 0;
static int i = 0;

int trace_count()
{
    i++;
    if(i%3 == 0) return 1;
    else return 0;
}

static VOID show_trace()
{
    icount++;
}

static VOID Trace(TRACE trace, VOID* v)
{
    TRACE_InsertIfCall(trace, IPOINT_BEFORE, (AFUNPTR)trace_count, IARG_END);
    TRACE_InsertThenCall(trace,IPOINT_BEFORE, (AFUNPTR)show_trace, IARG_END);
}


static VOID Fini(INT32 code, VOID* v)
{
    fprintf(stderr, "ins_number: %d\n", i);
    fprintf(stderr, "ThenCall_number: %ld\n", icount);
    /* fprintf(stderr, "BBL: %ld, INS: %ld, Avg: %f INSs/BBL\n", bbl_exec_nr, ins_exec_nr, (double)ins_exec_nr / bbl_exec_nr); */
}
```

图 5.5 测试 TRACE_InsertIfCall()和 TRACE_InsertThenCall()用 Pintool

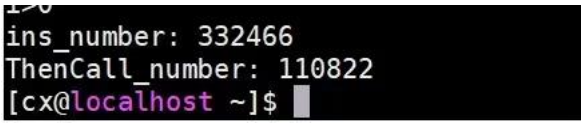

图 5.6 TRACE_InsertIfCall()和 TRACE_InsertThenCall()测试结果

验证 INS_InsertIfCall()和 INS_InsertThenCall()的思路与上述思路相同，实验结果如图 5.7 所示，满足预期。

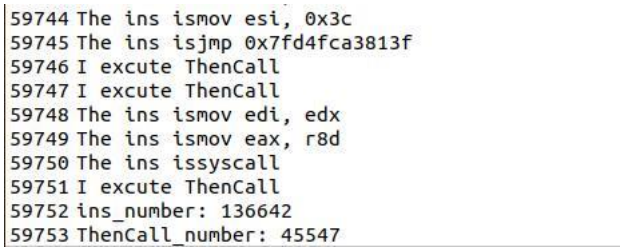

图 5.7 INS_InsertIfCall()和 INS_InsertThenCall()

验证 INS 的 PredicatedCall 系列插桩函数的正确性。测试方法：编写 PinTool 输出指令条数为 3 的倍数且指令 predicate 为 true 的指令。预期结果：输出的指令其对应的指令条数为 3 的倍数且其 predicate 为 true。测试结果如图 5.8 和 5.9 所示。

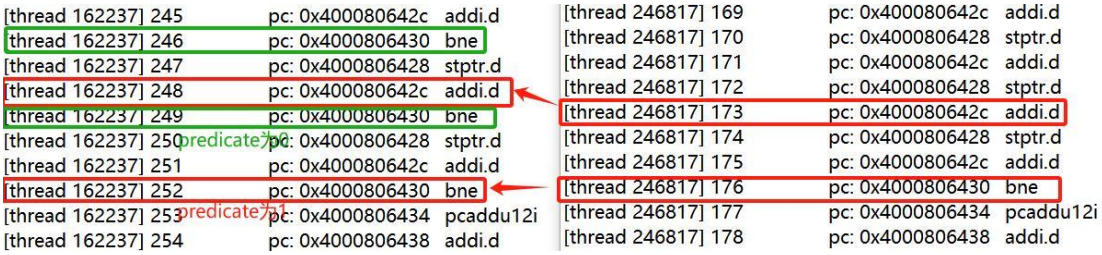

图 5.8 验证 INS 的 PredicatedCall 系列插桩函数的正确性

上图左边是原始输出的指令，其中绿色框是 predicate 为 false 的指令的例子，红色框是 predicate 为 true 的例子，右边是 predicate 为 true 的指令。

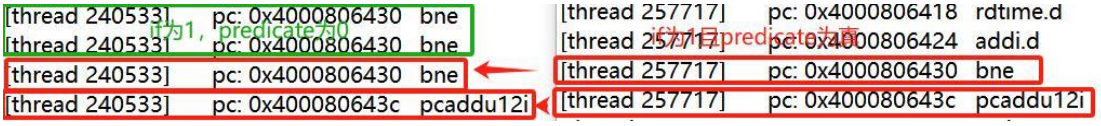

图 5.9 验证 INS 的 PredicatedCall 系列插桩函数的正确性

上图左侧为IfCall中分析调用返回值为 1 的指令，右侧为IfCall中分析调用返回值为 1 且Predicate为true的指令，即测试结果，与预期结果相同。

6.1.2 INS、BBL 和 TRACE 信息获取 API 的测试

INS 级信息获取 API 是返回 INS 块基础信息的 API。

INS_EffectiveAddressWidth (INS ins)：获取当前指令有效地址位宽的 API。

测试方法：编写 Pintool 调用 API，通过检查返回结果的正确性，验证 API 的正确性。编写的 Pintool 如图 5.10 所示，测试结果如图 5.11 所示。API 运行结果满足预期。

```
static UINT64 icount = 0;

static VOID show_ins(UINT64 pc, UINT32 opcode)
{
    ++icount;
    /* print ins */
    char msg[128];
    Ins ins;
    la_disasm(opcode, &ins);
    sprint_ins(&ins, msg);
    fprintf(stderr, "[thread %d] %lu\tpc: 0x%lx\t%s\n", PIN_ThreadId(), icount, pc, msg);
}

static VOID Instruction(INS ins, VOID* v)
{
    UINT32 ins_ea_width=1;
    ins_ea_width=INS_EffectiveAddressWidth(ins);
    fprintf(stderr, "ins_ea_width:%u",ins_ea_width);
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)show_ins, IARG_UINT64, ins->pc, IARG_UINT64, ins->opcode, IARG_END);
}
```

图 5.10 INS_EffectiveAddressWidth()测试 Pintool

```
ins_op:273ins_ea_width:0ins_op:273ins_ea_width:0ins_op:300ins_ea_width:64ins_op:300ins_ea_w
idth:64ins_op:300ins_ea_width:64ins_op:300ins_ea_width:64ins_op:300ins_ea_width:64ins_op:30
0ins_ea_width:64ins_op:300ins_ea_width:64ins_op:300ins_ea_width:64ins_op:300ins_ea_width:64
ins_op:300ins_ea_width:64ins_op:300ins_ea_width:64ins_op:300ins_ea_width:64ins_op:300ins_ea
_width:64ins_op:38ins_ea_width:0ins_op:300ins_ea_width:64ins_op:273ins_ea_width:0ins_op:273
ins_ea_width:0ins_op:275ins_ea_width:0ins_op:292ins_ea_width:64ins_op:273ins_ea_width:0ins
```

图 5.11 INS_EffectiveAddressWidth()测试结果

INS_IsSub(INS ins)：检查当前指令是否 sub 指令的 API。

测试方法：编写 Pintool 调用 API，通过检查返回结果的正确性，验证 API 的正确性。编写的 Pintool 如图 5.12 所示，Pintool 的功能是打印 API 返回值为真的指令，测试结果如图 5.13 所示。API 运行结果满足预期。

```
static VOID check(UINT64 pc, UINT32 opcode)
{
    /* print ins */
    char msg[128];
    Ins ins;
    la_disasm(opcode, &ins);
    sprint_ins(&ins, msg);
    fprintf(stderr, "[thread %d]\tpc: 0x%lx\t%s\n", PIN_ThreadId(), pc, msg);
}

static VOID Instruction(INS ins, VOID* v)
{
    if(INS_IsSub(ins))
        INS_InsertCall(ins,IPOINT_BEFORE, (AFUNPTR)check, IARG_UINT64, ins->pc,
}
```

图 5.12 INS_IsSub()测试 Pintool

```
[thread 3966893]        pc: 0x4002aa8afc        sub.d           $s0, $s0, $a0
[thread 3966893]        pc: 0x4002aa8b64        sub.d           $a0, $s3, $s0
[thread 3966893]        pc: 0x4002aa9d6c        sub.d           $a0, $zero, $a0
[thread 3966893]        pc: 0x4002aabcf4        sub.d           $s2, $s2, $s5
[thread 3966893]        pc: 0x4002aabf0c        sub.d           $t2, $t1, $s5
[thread 3966893]        pc: 0x4002aab350        sub.d           $t1, $t1, $t0
[thread 3966893]        pc: 0x4002aab358        sub.d           $t0, $s1, $t0
[thread 3966893]        pc: 0x4002aa7d08        sub.d           $a1, $a1, $t0
```

图 5.13 INS_IsSub()测试结果

其余 INS 级信息检查 API 都由类似的测试方法得到预期结果，测试用 Pintool 在仓库./target/loongarch/pin/pintool 文件夹中

BBL 级信息检查 API 测试结果如图 5.14 所示，满足预期。

```
The 1764 trace is:
The address of the first ins in the trace is:274890404064
The address of the first ins in the bbl(by bbl api) is:274890404064
The bblnum of the trace is:1
The insnum of the bbl is:7
The size of the bbl is:28
Does this bbl have fall-through ins,or not?:1
The 1 bbl is:
pc: 0x4000beb0e0        or              $a2, $a0, $zero
pc: 0x4000beb0e4        pcaddu12i       $a3, 0xce
pc: 0x4000beb0e8        ld.d            $a3, $a3, 0x62c
pc: 0x4000beb0ec        lu12i.w         $a1, 0xffffffff
pc: 0x4000beb0f0        addi.w          $a7, $zero, 0x5e
pc: 0x4000beb0f4        or              $a0, $a2, $zero
pc: 0x4000beb0f8        syscall
```

图 5.14 BBL级API测试结果

TRACE 级信息检查 API 测试结果如图 5.15 和图 5.16 所示，满足预期。

```
The 1763 trace is:
The address of the first ins in the trace is:274889945344
The address of the first ins in the trace(by trace api) is:274889945344
The bblnum of the trace is:2
The insnum of the trace is:3
The size of the trace is:12
Does this trace have fall-through ins,or not?:0
The 1 bbl is:
pc: 0x4000b7b100        bne             $s1, $s0, 0xfffffffd
The 2 bbl is:
pc: 0x4000b7b104        or              $a0, $s5, $zero
pc: 0x4000b7b108        bl              0x1bff6

The 1764 trace is:
The address of the first ins in the trace is:274890404064
The address of the first ins in the trace(by trace api) is:274890404064
The bblnum of the trace is:1
The insnum of the trace is:7
The size of the trace is:28
Does this trace have fall-through ins,or not?:0
The 1 bbl is:
pc: 0x4000beb0e0        or              $a2, $a0, $zero
pc: 0x4000beb0e4        pcaddu12i       $a3, 0xce
pc: 0x4000beb0e8        ld.d            $a3, $a3, 0x62c
pc: 0x4000beb0ec        lu12i.w         $a1, 0xffffffff
pc: 0x4000beb0f0        addi.w          $a7, $zero, 0x5e
pc: 0x4000beb0f4        or              $a0, $a2, $zero
pc: 0x4000beb0f8        syscall

[cx@localhost ~]$
```

图 5.15 TRACE 级 API 测试结果

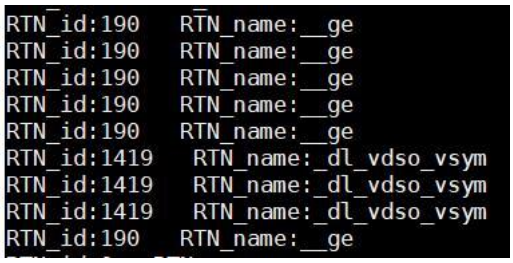

图 5.16 TRACE 级 API 测试结果

6.1.3 RTN、IMG 级插桩 API

测试RTN级instrumentation API：使用插桩实例，调用RTN_InsertCall和RTN_FindByName检测memory.c文件执行过程中内存的分配和释放情况，memory.c文件中开辟了一个字节大小为 0x34 的数组，并对其进行了内存释放。测试结果如图 5.17 所示，每进行内存分配之前，打印字符串“malloc(分配的字节大小)”，进行内存分配之后，打印字符串“return(分配的内存地址)”，每进行内存释放之前，打印字符串“free(分配的字节大小)”，可见RTN_InsertCall正确执行。

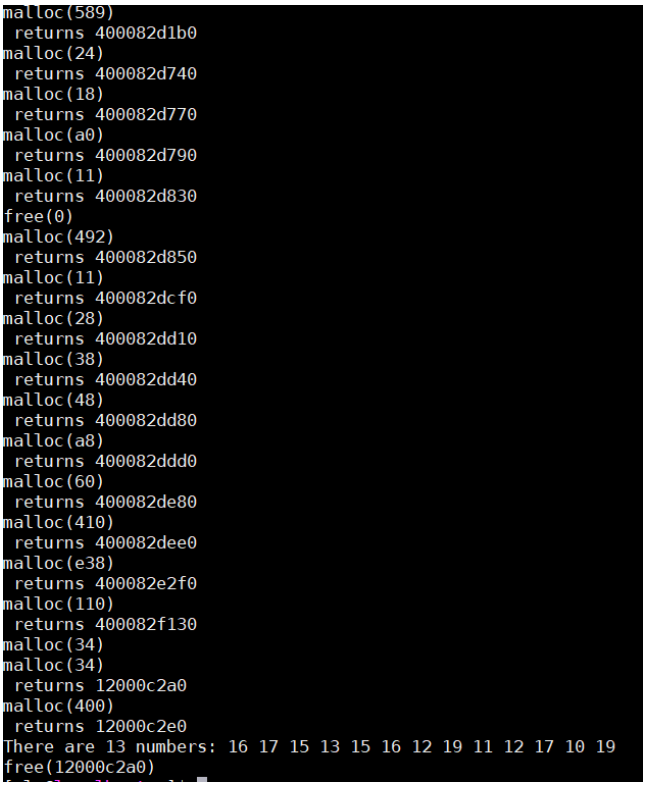

图 5.17 RTN 级 Instrumentation API 测试结果

6.1.4 RTN、IMG 级信息检查 API

测试RTN级API：编写插桩工具，调用RTN_Valid、RTN_Next、RTN_Id、RTN_Name、RTN_Size、RTN_NumIns和RTN_Address查看hello.so文件运行过程中产生的函数的编号、名称、字节大小、包含的指令条数和起始地址，并调用RTN_FindByName和RTN_FindByAddress根据最后一个函数的名称和地址寻找RTN，打印出找到的RTN的名称和地址。测试结果如图 5.18 所示，每个函数的编号名称等信息都正确打印，根据最后一个函数的名称和地址找到的函数正是最后一个函数，所有API均可正确执行。

```
RTN_id   RTN_name   RTN_size   RTN_numins   RTN_addr
   5090          __lxstat64          88          22          274887472352
RTN_id   RTN_name   RTN_size   RTN_numins   RTN_addr
   5091          __wcstombs_chk          88          22          274887601184
RTN_id   RTN_name   RTN_size   RTN_numins   RTN_addr
   5092          pututxline          4          1          274887779088
RTN_id   RTN_name   RTN_size   RTN_numins   RTN_addr
   5093          dup3          28          7          274887476568
RTN_id   RTN_name   RTN_size   RTN_numins   RTN_addr
   5094          timespec_get          124          31          274887315520
274887315520 : The last RTN is timespec_get
```

图 5.18 RTN级Inspection API测试结果

测试IMG级API：编写插桩工具，调用IMG_AddInstrumentFunction、IMG_Next、IMG_Valid、IMG_Id和IMG_Name注册IMG级插桩函数，查看hello.c程序运行中用到的所有.so文件，打印出这些文件的编号和名称。测试结果如图 5.19 所示，正确打印出所有IMG的编号及其名称，所有API均可正确执行。

```
The 1 IMG is /home/glp/hello
The 2 IMG is /usr/lib64/ld-2.28.so
The 3 IMG is /usr/lib64/libc-2.28.so
```

图 5.19 IMG 级 API 测试结果

6.2 性能测试

使用CoreMark分别对本插桩框架和Intel Pin进行分数评估，测评结果如表 5.1 所示。本插桩框架及各项粒度插桩的执行效率和Pin还有较大差距，为了实现TRACE和BBL级别API执行的正确性，目前基本块链接模块属于失效状态，使得插桩框架每执行完一个TB之后，必须要进行上下文切换，多次的上下文切换产生了大量的时间开销，后续将重新添加和更改基本块链接模块，以提高插桩框架

执行效率。

指令插桩则会在每一条指令前插入分析函数的调用，大量的分析函数调用开销巨大。而进行基本块插桩后，每执行一个基本块才会调用一次分析函数。因此插桩工具的编写者通常会选择对基本块插桩，而不是对每条指令插桩。插桩性能还与分析函数的复杂程度有关，分析函数越复杂，程序就需要花费更多时间在执行分析函数上。

表 5.1 本插桩框架和Pin框架的CoreMark分数评估

	原生执行	插桩框架	指令插桩	基本块插桩	函数插桩
本插桩框架执行分数	12818.869	48.501	2.988	47.281	48.371
本插桩框架执行效率	100%	0.38%	0.023%	0.37%	0.38%
Pin执行分数	22822.366	19533.793	1859.197	8932.922	17736.786
Pin执行效率	100%	85.59%	8.15%	39.14%	77.71%

6 总结和展望

本文首先简介了动态二进制插桩技术，解释了其强大的功能以及对程序分析的重要意义，同时还说明了在龙芯平台开发一款全新的插桩框架的重要性。之后从整体上介绍了插桩框架的工作流程，以及实现插桩工作的各个模块的功能，在理解框架的基础上才能正确的为框架添加新的功能。

在原有框架基础上，本文的做出了许多创新性的工作。

第一，修正了原有框架的不足。本文通过实验测试，更加仔细的重新定义了框架原有的 BBL 粒度，使其符合实际运行的同时适配 TRACE 粒度的定义和插桩工作。

第二，本文从无到有的为一款应用于龙芯平台上的插桩框架设计并实现了一个 TRACE 级插桩粒度。参照英特尔的 Pin，从 x86 平台上开展了一系列测试工作，得到了准确且细致的 TRACE 的定义。从这个过程学到了一切都应进行尝试，并以实验结果为准，官方的定义也可能不全面，同时要大胆猜测小心验证，善于质疑，发现问题。

第三，添加了新的插桩接口。从易用性的思想出发，对标英特尔公司的插桩工具 Pin，添加了 TRACE 粒度的插桩接口。

第四，本文设计并实现了对 RTN 和 IMG 粒度的定义，并添加了相关的插桩接口。

框架实现了基本的插桩功能，然而还有一些不足之处可以改进。

第一，软件的 API 不够丰富。英特尔的 Pin 中有一些 API 因基础架构原因无法在龙芯平台实现，同时软件目前实现的 API 又是完全参照英特尔公司的定义，因此从功能的丰富度上不如 Intel Pin。在后续的工作上，本团队认为可以根据龙芯架构的特点，创新性的加入实现全新功能的 API，丰富软件的功能。

第二，从执行效率上也有待提高。在框架中是以一个翻译块为单位进行插桩和运行的，而一个翻译块中只含有一个 TRACE，一个 TRACE 中又最多只有 3 个 bbl。于是每需要一个翻译块就进行一次上下文切换，进行一次调度模块和指令解析模块控制权的切换，效率很低。而且对于 TRACE 中只有 3 个 BBL 的定义，也

是为了兼容 Intel pin 的定义，如果一个 TRACE 中有更多的 BBL，对执行效率也会有一定的提升。在后面的工作中可以探索出适合龙芯的 TRACE 中的 BBL 数，同时可以进一步完善对于 TRACE 级粒度的定义，使其更加适配龙芯的架构。

第三，对同一位置的多次插桩处还可以进行优化。在当前的框架中，为了保证程序的正确执行，会在回调函数前后分别保存寄存器中的数据和还原寄存器数据，这样虽然能保证程序的正确性，却带来很多冗余的操作。当在同一位置进行插桩时，从理论上只需要在第一个回调函数前保存寄存器中的值，并在最后一个回调函数后重新加载寄存器，这样优化能节省很大的运行开销，同时还能释放冗余指令所占据的储存空间，是可行且有意义的优化思路。

在比赛的后续阶段，本后续团队将努力改善上述不足，继续完善插桩框架，提高框架运行的效率。希望在不久的将来，本团队独立开发的动态二进制插桩工具能够运行在自研国产芯片上，为龙芯平台上的软件开发赋能。

参考文献

[1]龙芯中科技术股份有限公司. 龙芯架构参考手册卷一：基础架构[M]. 1.03. 龙芯中科技术股份有限公司, April 2023.

[2]LUK C K, COHN R, MUTH R, et al. Pin: building customized program analysis tools with dynamic instrumentation[J]. Acm sigplan notices, 2005, 40(6): 190-200.

[3]Dey, Sandip. Analysis of Performance Overheads in DynamoRIO Binary Translator (2020): n. pag. Print.

[4]NETHERCOTE N, SEWARD J. Valgrind: A program supervision framework[J]. Electronic notes in theoretical computer science, 2003, 89(2): 44-66.