基于 QEMU 的龙架构平台插桩软件

高林萍[1]，徐伟[2]，陈曦[1]，穆奕博[1]，张开[1]

（1. 中国科学技术大学先进技术研究院，安徽 合肥 230026；2. 中国科学技术大学计算机科学与技术学院，安徽 合肥 230026）

摘　要： 随着软件规模和复杂性指数级增长，对程序运行时行为的监控和分析变得越来越困难，动态二进制插桩（DBI）技术是解决这一问题的有效手段，现有成熟的二进制插桩软件有 Pin、Valgrind 等，主要支持 x86、ARM 等主流架构，但在新兴的自主指令集架构上缺乏支持。龙架构（LoongArch）作为我国自主研制的指令集架构，具有较好的自主性、先进性与兼容性，但其发展时间较短，生态环境尚不完善，特别是在调试工具链方面存在明显短板。为了填补这一空白，推动龙架构生态的成熟，推出一款支持龙架构的 DBI 软件具有重要意义。本文旨在设计并实现一款基于 QEMU（Quick Emulator）框架的 DBI 软件，以支持龙架构的程序监控与分析。该软件对标 Pin，设计实现了 5 个基础插桩粒度及相关插桩应用程序编程接口（API），并在此基础上开发了 20 多个插桩工具，供用户直接使用或学习插桩工具的编写。为了提升框架性能，通过优化条件跳转指令的翻译、基本块链接和插桩内联等方法对框架进行了改进。性能测试结果表明，优化后的框架在指令级插桩效率上提升了 100 多倍，基本块级插桩效率提高了近 33 倍。

关键词： 动态二进制插桩；Pin；龙架构；QEMU 框架；插桩应用程序编程接口；插桩工具

源代码链接： https://github.com/GaoLinping-arch/my-qemu-instrument

中图分类号： TP273　　**文献标志码：** A　　**DOI：** 10.19678/j.issn.1000-3428.0070725

An Instrumentation Software for LoongArch Based on QEMU

GAO Linping[1], XU Wei[2], CHEN Xi[1], MU Yibo[1], ZHANG Kai[1]

(1. Institute of Advanced Technology, University of Science and Technology of China, Hefei 230026, Anhui, China;
2. School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, Anhui, China)

【Abstract】 As software scale and complexity grow exponentially, monitoring and analyzing program runtime behavior has become increasingly challenging. Dynamic Binary Instrumentation (DBI) is an effective solution to this problem, with mature tools such as Pin and Valgrind supporting mainstream architectures such as x86 and ARM. However, these tools lack support for emerging domestic instruction set architectures such as LoongArch. LoongArch, a self-developed instruction-set architecture in China, exhibits high levels of autonomy, advancement, and compatibility. However, owing to its relatively short developmental history, the ecosystem remains incomplete, particularly in the debugging toolchain. Developing a dynamic binary instrumentation tool is important to address this gap and promote the maturation of the LoongArch ecosystem. This study aims to design and implement a dynamic binary instrumentation tool based on the Quick Emulator (QEMU) framework to support program monitoring and analysis in LoongArch. The tool, modeled after Pin, implements five fundamental instrumentation granularities and related Application Programming Interfaces (APIs), along with over 20 instrumentation tools for direct use or as learning resources for tool development. To enhance performance, the framework is optimized through improvements in conditional jump instruction translation, basic block linking, and instrumentation inlining. Performance tests demonstrate that the optimized framework achieves improvement of over 100 times in instruction-level instrumentation efficiency and nearly 33 times in basic block-level instrumentation efficiency.

【Key words】 Dynamic Binary Instrumentation (DBI); Pin; LoongArch; Quick Emulator (QEMU) framework; instrumentation Application Programming Interface (API); instrumentation tool

0　引言

步入 21 世纪以来，CPU 进入了高速发展的时代，而 CPU 的发展离不开指令系统的支撑，我国 CPU 研发应采用兼容指令系统还是自主研发指令系统是学术界和产业界长期争论的一个话题。采用兼容指令系统虽可直接利用现有的成熟软件，但限制了自主软件的研发，不利于长远发展。于是 2020 年龙芯中科推出了一套自主研发的国产指令系统架构——龙架构[1]（LoongArch），从指令系统的根源上实现了自主，摆脱了自主软件产业发展长期受制于人的局面，对芯片国产化具有重要意义。

基金项目： 中国科学院基础与交叉前沿科研 B 类先导专项（XDB0660000）。

作者简介： 高林萍，女，硕士，主研方向为二进制翻译、编译优化；徐伟，高级实验师、博士；陈曦、穆奕博，硕士研究生；张开，博士研究生。

收稿日期： 2024-12-19　　**修回日期：** 2025-03-17　　**E-mail：** glp@mail.ustc.edu.cn

龙架构如今已完成了许多基础软件系统的移植和开发,但其软件生态仍待进一步完善。随着计算机技术的高速发展,软件的复杂性也在呈指数级增长,高效的程序分析技术就变得尤为重要。动态二进制插桩(DBI)技术[2]则在程序分析方面呈现出了巨大优势,它是一种能够在机器代码级别动态分析程序执行情况的技术,通过将特定代码注入运行程序中,在不改变原程序执行结果的前提下完成对程序运行时情况的分析和统计,其可以提供高效的错误检测的程序测试方法,在程序优化、故障定位、内存检测、安全审计等方面都有广泛应用[3-5]。随着 DBI 技术的不断发展,各种框架层出不穷,而 QEMU(Quick Emulator)[6]作为一个开源的仿真器,其包含的二进制翻译、基本块生成、代码调度等模块都可用于实现 DBI。

软件生态对龙架构的发展起着决定性的作用,然而龙架构推出的时间较短,其生态还处在初级发展阶段,设计和实现一款能够支持龙架构的高性能插桩工具对龙平台软件生态环境的发展完善具有重要作用和意义,可以为后续龙架构上软件的代码优化和程序性能分析提供必要支持。受上述启发,本文对标一款如今应用最广泛的成熟插桩软件 Pin,提出了基于 QEMU 框架的可在龙架构平台上运行的 DBI 软件,该软件可提供 5 种基础的插桩粒度和相关插桩应用程序编程接口(API)。

本文的主要贡献包括 3 个方面:

1)对标 Pin 软件,使用 C/C++语言设计实现了一个基于 QEMU 的可以支持龙架构平台的插桩框架。该框架支持 5 种基本插桩粒度,这些粒度包括指令级、基本块级、轨迹级、函数级和镜像级。设计实现了 5 个插桩粒度对应的 113 个 API,用户可以通过引用头文件对这些 API 进行调用。

2)设计实现了 20 多个插桩工具。这些插桩工具的主要功能涵盖了指令分析、寄存器分析、访存分析、控制流分析以及函数分析 5 个应用场景。用户可以直接使用这些工具对程序进行插桩,也可以通过这些工具学习如何编写所需功能的插桩工具。

3)优化框架性能。通过条件跳转指令翻译优化、基本块链接和内联插桩提高了框架性能和插桩效率。测试结果表明,优化后的指令级插桩效率提高了 100 多倍,基本块级插桩效率提高了近 33 倍,和 Pin 软件相比,插桩效率已在同一个数量级。

本文第 1 章介绍了插桩框架的发展现状;第 2 章介绍了龙架构平台的插桩框架,包括各级插桩粒度、插桩 API 和插桩工具的详细设计实现;第 3 章介绍了插桩框架的性能优化,包括条件跳转翻译优化、两种基本块链接方法和内联插桩的实现;第 4 章使用 CoreMark 对框架和 Pin 进行了执行效率测试对比,并和主流插桩软件进行了功能对比。

1 相关工作

DBI 技术最早由 HUANG[7]于 1978 年提出,他将程序插桩技术看作是一种除了能够获取程序本身输出的信息外,还可以收集程序检测错误的附加信息的方法,在保证不修改原程序逻辑性能的基础上,向程序中添加一些特定代码,获得程序运行时的控制流和数据流信息,初步提出了插桩软件的设计思想。后来 BRUENING 等[8]于 2003 年合作实现了一个用于程序动态分析和优化的开源框架 DynamoRIO,提供了一个简单高效的轻量级 API,便于构建轻量级 DBI 工具[9]。不久,NETHERCOTE 等[10]又发布了一种专为构建重量级 DBI 工具[11]而设计的插桩框架 Valgrind。LUK 等[12]则于 2005 年首次提出了遵循 ATOM(Analysis Tools with OM)[13]模型的新型插桩软件 Pin。经过二十多年的发展,这些框架逐渐成熟,已经成为现有的主流插桩框架,可以支持 IA-32、X86 和 ARM 等指令架构。Pin 作为其中应用最广泛的 DBI 软件,为用户提供了多种插桩粒度和丰富的插桩 API[14-15],用户可以调用这些 API 设计具有特定功能的插桩工具(PinTool)[16-18],但这些功能强大的 API 并未开源。

DBI 作为一个灵活高效的代码分析技术,有广泛的应用场景。国产架构上的 DBI 研究正在从“工具移植”向“深度定制”发展,重点解决了性能开销、安全性问题等多领域问题。林昊等[19]提出基于 DBI 面向 CryptoAPI 加密应用的检测方法,借助动态二进制分析平台 Pin 记录加解密信息,设计算法检测,有效检测出两类密钥安全漏洞。陈金宝等[20]开发的轻量化工具 DBI-Go,采用静态分析与 DBI 技术,助力开发人员找出逃逸算法错误,同时无需修改 Go 的编译运行时,可以适配不同版本的 Go。卢帅兵等[21]基于二进制翻译,开发出支持多平台的内核函数调用跟踪框架,在二进制翻译的中间代码阶段生成用于特定信息获取的指令,实现对函数调用的跟踪。

插桩软件一直在二进制翻译上有较高需求,随着二进制翻译技术的发展,拥有灵活动态二进制翻译模块的开源指令集仿真器 QEMU 也受到了越来越多的关注。ZENG 等[22]于 2015 年基于 QEMU 设计了一款可以与 Pin-API 兼容的插桩框架 PEMU,其支持 OS 内核插桩。邹伟等[23]于 2019 年提出一种基于 QEMU 的 DBI 框架,实现了对基本

块的运行时统计采集，完成了对嵌入式系统软件代码的中间代码级的检测以及日志信息处理工具的开发。COTA 等[24]设计了一个基于 QEMU 的高性能跨 ISA 机器仿真器和检测工具 Qelt。GAN[25]则于 2023 年提出了一种基于 QEMU 的 Linux 系统调用的插桩框架，该框架结合了基于接口和盲检测的方法收集系统调用信息，并引入了系统调用跟踪机制来提高分析的可靠性。

上述框架的实现与应用，都充分表明了 QEMU 作为插桩框架实现平台的可行性和高效性。然而这些研究都是基于 QEMU 完成了在主流指令架构上的特定插桩操作，并未提供一个可使用的多功能的插桩软件，更无法支持在龙架构平台上运行。本文基于 QEMU 设计了一个与 Pin 高度兼容的可以支持龙架构平台的插桩软件。

2 龙架构平台插桩框架

本文实现的插桩框架基于 QEMU 开发，使用 C/C++语言编写完成。QEMU 在本文系统中主要提供虚拟机运行环境，由于动态二进制翻译和 DBI 技术有一定程度上的相似性——都采用即时编译生成新的二进制代码代替原始程序，因此 QEMU 中的许多功能模块都可以应用在本文的插桩框架中。

2.1 插桩框架组成

本文系统的整体架构如图 1 所示，其主要由四大部分组成，分别是插桩 API、调度器、即时编译器和代码缓存管理。

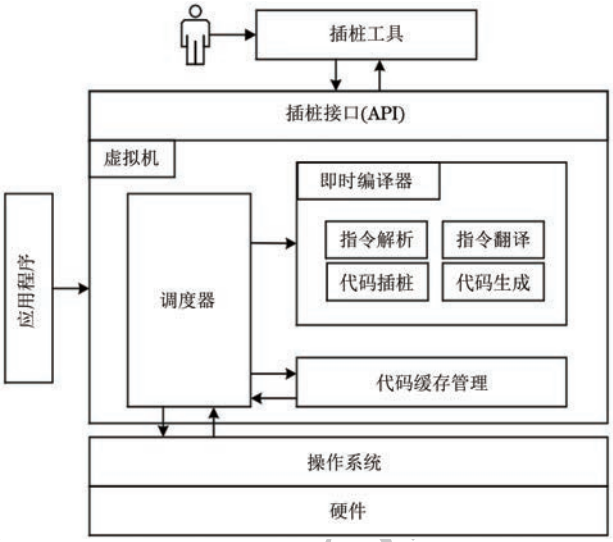

图 1 插桩框架图

Fig. 1 Instrumentation framework diagram

插桩 API 是插桩框架为用户提供的可以通过插桩工具调用的插桩接口，也是本项目的主要工作内容，用于指令代码的判断，以及将特定代码注入原程序的指令流中，获取程序运行时信息。执行调度器负责协调各个部件完成对应用程序的插桩，并调度执行插桩后的指令代码。即时编译器在截获应用程序的指令流后，对二进制代码进行解析、翻译，生成新的指令序列，再对得到的指令序列进行插桩，最后生成插桩后的代码并交由调度器执行。代码缓存模块负责保存插桩后的代码，避免以后对相同指令的重复插桩，保证插桩的执行效率。

2.2 插桩原理

图 2 展示了插桩框架的运行流程。以统计程序

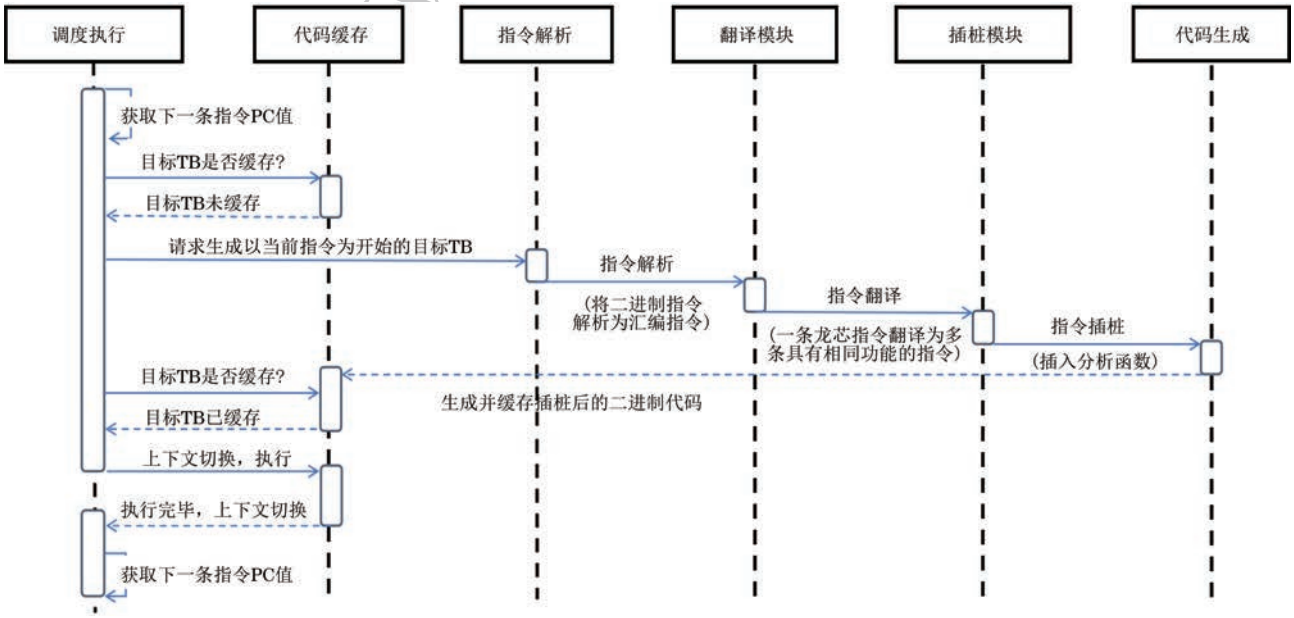

图 2 插桩运行时序图

Fig. 2 Instrumentation runtime sequence diagram

中指令数的插桩工具为例，插桩框架启动时，会首先加载用户编写的插桩工具，完成插桩前的函数注册等初始化工作，包括计数函数、输出函数等。虚拟机在截获程序的执行流以后，通过指令的 PC 值查找其所在的基本块是否在代码缓存中，若不在，则会请求插桩，由即时编译器对指令进行解析、翻译和插桩，在统计指令数的例子中为更新指令计数值，完成一个翻译基本块（TB）的生成后，会把得到的代码块放入代码缓存中并查找执行。执行完成后，继续获取下一条指令的 PC 值，重复上述流程，直至原始程序运行结束并输出统计结果。

2.3 插桩粒度

本文对标 Intel Pin 软件，共设计实现了 5 个基础插桩粒度和相应的插桩 API，图 3 展示了这些粒度之间的包含关系（彩色效果见《计算机工程》官网 HTML 版，下同）。各粒度的详细定义如下：

1）指令块（INS）级：以 INS 为单位进行插桩，INS 是含多条指令的指令块。

2）基本块（BBL）级：以 BBL 为单位进行插桩。BBL 为一个单入口单出口的指令序列，以跳转指令结尾，一个 BBL 包含多个 INS。

3）轨迹（TRACE）级：以 TRACE 为单位进行插桩。TRACE 为一个单入口多出口的指令序列，通常以无条件跳转指令结尾，但若在生成过程中，产生第 3 个 BBL 时还未遇到无条件跳转指令，则会以条件跳转指令结尾，包含 1～3 个 BBL。

4）函数（RTN）级：以 RTN 为单位进行插桩。RTN 是相关函数信息。

5）镜像（IMG）级：以程序映像为单位进行插桩。每个 IMG 对应一个.so 文件，包含多个 RTN。

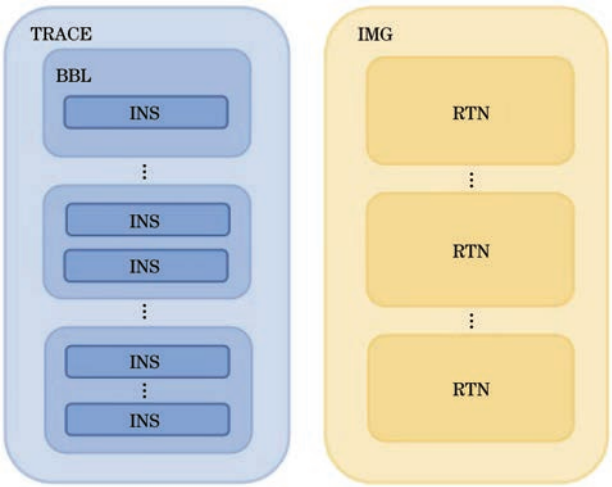

图 3 插桩粒度体系

Fig.3 Instrumentation granularity hierarchy

一条龙芯指令经翻译和插桩后变为多条指令，这些指令放在一个 INS 结构体中，多个 INS 构成一个 BBL，而一个或多个 BBL 又构成一个 TRACE，INS、BBL 和 TRACE 是程序边运行边生成的。每加载一个动态共享库文件就产生一个对应的 IMG，程序运行时需解析多个符号信息，即函数信息，一个 IMG 包含多个 RTN，RTN 和 IMG 的信息是在程序运行前就收集完成的，因此可以将 INS、BBL 和 TRACE 看作一个体系，将 RTN 和 IMG 看作一个体系。在 2.2 节的例子中，统计指令数的插桩工具主要针对 INS 级这一插桩粒度进行插桩，从而完成原始程序中指令总数的统计工作。

2.4 插桩 API

在编写插桩工具时，需要使用不同粒度的插桩 API 来进行编写，从而完成相应的功能。在上面的例子中，需要使用 INS 级插桩 API 来完成 INS 的插入，使用 TRACE 级和 BBL 级 API 来遍历翻译块中的指令。本节将以体系为单位详细介绍各个粒度 API 的设计实现和插桩工具的编写方法，这些 API 主要可以分为三大类：一类是信息检查型 API，用于获取各粒度的基本信息，如指令类型等；一类是插桩型 API，用于将特定代码插入程序特定位置；还有一类是注册型 API，用于注册插桩 API，申明插桩粒度。插桩工具则通过调用插桩 API 完成对程序的插桩操作。

2.4.1 INS、BBL 和 TRACE 的解析

INS、BBL 和 TRACE 3 个粒度的具体解析流程如下：

1）INS 级粒度的解析：一条原始的指令经处理后，会生成与原始指令功能相同的一条或多条指令，这些指令共同组成一个 INS。

2）BBL 级粒度的解析：BBL 是一个单入口单出口的指令序列，以跳转指令结尾，多个 INS 共同组成一个 BBL。

3）TRACE 级粒度的解析：TRACE 是程序能够连续执行的最大指令序列，通常以无条件跳转指令结尾，由 1～3 个 BBL 组成。

首先根据指令地址判断 TRACE 是否在代码缓存中，若不在，则初始化一个新的 BBL 和一个新的 TRACE，判断当前指令是否为跳转指令，若是，则结束当前 BBL 的生成，进一步判断当前指令是否为无条件跳转指令，若是，则结束当前 TRACE 的生成，否则，继续判断当前 TRACE 中的 BBL 数是否小于 3，若是，继续生成 BBL 并添加到当前 TRACE 中，反之则结束当前 TRACE 的生成。经

过以上流程，则可以解析获取 INS、BBL 和 TRACE 3 个粒度。

2.4.2 RTN 和 IMG 的解析

每加载一个.so 文件，就会产生一个对应的 IMG，也会产生对应的 ELF 文件，ELF 的内容包括 ELF Header、Program Header Table、Sections 以及 Section Header Table。其中有几个特殊的 Section 分别是 symtab、dynsym、strtab，symtab 和 dynsym，它们都是符号表。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 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。这里只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 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。在获取所有的符号信息后，筛选出满足条件的符号信息，即为所需要的 RTN 信息。

2.4.3 INS、BBL 和 TRACE 级插桩 API

框架共设计实现了 64 个 INS 级 API、13 个 BBL 级 API 和 12 个 TRACE 级 API，部分 API 的信息描述如表 1 所示。DBI 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，插桩的位置主要包括插桩对象前和插桩对象后。为了确保不影响程序的执行结果，从整体上看，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转回来之后恢复上下文，继续原本程序的执行。

表 1 部分 API 信息描述

Table 1 Partial API specifications

API 名称	级别	描述
INS_InsertCall	INS	插入相对指令的调用
INS_IsBranch	INS	判断是否为分支指令
BBL_InsertCall	BBL	插入相对 BBL 的调用
TRACE_InsertCall	TRACE	插入相对 TRACE 的调用

在统计指令数的插桩工具中，使用的核心 API 为 INS 级 API——INS_InsertCall，该 API 是最基础的插桩型 API。INS_InsertCall 的代码实现流程如图 4 所示，首先初始化分析函数的参数列表，再利用 parse_iarg()解析参数列表，并将它们存储到分析调用的参数结构体中。之后通过 ins_get_next_cb_position()获取分析调用待插入的准确位置。最后通过 insert_callback()函数插入对分析函数的调用。由此，便完成了 INS 级的插桩操作。

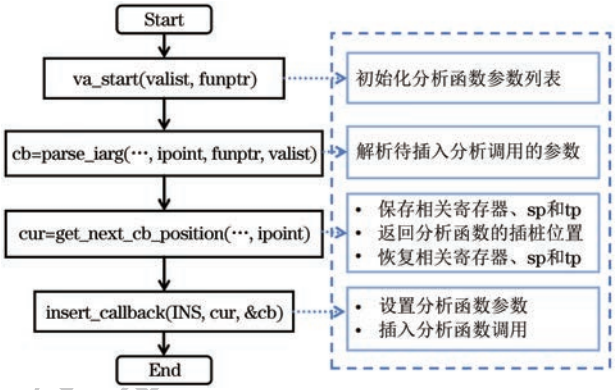

图 4 INS_InsertCall 的代码实现

Fig.4 The code implementation of INS_InsertCall

2.4.4 RTN 和 IMG 级插桩 API

框架共设计实现了 14 个 RTN 级 API 和 10 个 IMG 级 API。从底层逻辑来看，插桩只能是针对指令插桩，而 RTN 的信息在程序运行前就已经收集完成了，由 RTN 数组存储，与 INS 并无直接关联，这也导致了无法对 RTN 插桩，因此如何将 INS 和 RTN 关联起来就成了对 RTN 插桩的关键。考虑到 RTN 的出入口一定是 TRACE 的出入口，可以在开始时就记录 RTN 待插入的分析调用，在解析生成 TRACE 的过程中，通过判断 TRACE 的出入口是否为 RTN 的出入口，从而获取 RTN 的待插桩点，完成对 RTN 的插桩。

2.5 插桩工具

包括统计程序指令数的插桩工具在内，框架共设计编写了二十多个 PinTool 实例，用户可以直接使用这些 PinTool 对程序进行插桩来完成不同的分析需求，也可以通过这些 PinTool 学习如何编写自己的 PinTool，每个 PinTool 主要由四部分组成：

1)主函数：负责注册插桩函数，声明插桩粒度，完成插桩的主流程。

2)分析函数：该函数是 PinTool 的核心部分，被插入到特定位置的函数，负责在程序运行过程中收集信息。

3)插桩函数：通过调用插桩 API 获取指令信息，并在指定位置插入分析函数。

4)结束函数：用于打印获取的程序运行时的相关信息。

图 5 展示了一个名为 BBL_count 的 PinTool，其中 main()是主函数，在主函数中首先使用 PIN_Init()初始化了插桩工具，接着使用 TRACE_AddInstrumentFunction()声明了插桩粒度为 TRACE。bbl_statistic()是分析函数，可以统计执行的 BBL 数量和 INS 数量。Trace()是插桩函数，调用了 BBL_InsertCall()插入了相对于 BBL 的分

析函数。Fini()为结束函数,打印出了程序运行过程中 BBL 和 INS 的执行数量。

```
#include "pintool.h"
#include "../ins_inspection.h"
static uint64_t bbl_exec_nr = 0;
static uint64_t ins_exec_nr = 0;
static VOID bbl_statistic(uint64_t ins_nr)
{
    ++bbl_exec_nr;
    ins_exec_nr += ins_nr;
}
static VOID Trace(TRACE trace, VOID* v)
{
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)bbl_statistic,
IARG_UINT64, BBL_NumIns(bbl), IARG_END);
    }
}
static VOID Fini(INT32 code, VOID* v)
{
    fprintf(stderr, "BBL: %ld, INS: %ld, Avg: %f INSs/BBL\n",
bbl_exec_nr, ins_exec_nr, (double)ins_exec_nr / bbl_exec_nr);
}
static INT32 Usage(void) { … }
int main(int argc, char* argv[])
{
    if (PIN_Init(argc, argv)) return Usage();
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_AddFiniFunction(Fini, 0);
    return 0;
}
```

```
$ ./qemu-loongarch64 -t
./target/loongarch/pin/pintools/libb
bl_count.so /usr/bin/gcc --version
…
BBL: 201179, INS: 956629, Avg: 4.755114
```

图 5 BBL_Count 的代码结构

Fig.5 The code structure of the BBL_Count

3 性能优化

在完善了框架和设计实现了 5 个粒度的通用 API 后,对框架进行了性能优化,通过条件跳转指令的翻译优化和基本块链接减少了上下文切换次数,又通过内联插桩,减少了分析函数调用产生的开销,提高了框架的运行效率。

3.1 条件跳转指令翻译优化

原框架中每个 TRACE 只包含一个 BBL,在原来的翻译中,每个以条件跳转为结尾的 BBL 执行完毕后,无论跳转条件是否成立,最后都会回到上下文切换查找下一个要执行的 BBL。完善后的框架中每个 TRACE 包含多个 BBL,于是当前 TRACE 中本可以按顺序执行的下一个 BBL 却要经上下文切换查找后才可以执行,造成大量不必要的查找开销。因此修改条件跳转指令的翻译:增加一个条件跳转不成立时的上下文切换标志,初始化为 false。每次开始翻译一个新的 TRACE,只有当此时的 BBL 是 TRACE 中的最后一个 BBL,且该 BBL 以条件跳转指令为结尾时,该标志才为 true,最后一个 BBL 才会在跳转条件不成立时跳转到上下文切换寻找下一个 TRACE 基本块,否则则顺序执行当前 TRACE 中的下一个 BBL,避免每执行完一个 BBL,都需要进行上下文切换才能查找到下一个将要执行的 BBL,大幅减少了不必要的上下文切换开销。

3.2 基本块链接

条件跳转指令翻译的修改,虽然减少了上下文切换次数,但只有 TRACE 内的 BBL 可以不经上下文切换的查找,直接顺序执行下一个 BBL,TRACE 之间的 BBL 都未链接,而 TRACE 数量是巨大的,且会重复执行,太多的上下文切换查找将影响插桩效率。为此本文提出了两种基本块链接方法,尽量让一个 BBL 执行完成后可以直接跳转到目标 BBL 执行。

方法 1 链接部分 BBL。

翻译得到基本块时,无条件跳转指令和最后一个 BBL 中的条件跳转指令都可以是 TRACE 的最后一条指令,对这类指令保留跳转出口,对于 TRACE 中非结尾 BBL 中的条件跳转指令,当不满足跳转条件时,顺序执行当前 TRACE 内的下一个 BBL,当满足跳转条件时,直接通过跳转出口跳转到上下文切换,查找下一个执行的基本块,不进行基本块链接。该方法相当于只对每个 TRACE 的最后一个 BBL 进行了链接。

方法 2 链接全部 BBL。

由于龙芯中指令地址的最后两位是 00,可以利用这两位传递出口索引号,也就是可以传递出 4 个标识信息。除了索引号外,还须设有一个无效索引标识,这也限制了出口的个数最多为 3 个,当 TRACE 基本块中每个 BBL 都以条件跳转为结尾时,其出口数最多,因此也限制了 TRACE 中 BBL 的个数最多为 2 个。每执行完一个 BBL,记录已执行的 BBL 所属 TRACE 的地址和该 BBL 的跳转出口在 TRACE 所有出口中的索引号,执行下一个 BBL 时,获取正在执行的 BBL 所属 TRACE 的地址,将上一个 BBL 和当前 BBL 链接起来,完成所有基本块之间的链接。这样就使得每个 BBL 在执行完后,无论是否满足跳转条件都可以直接找到下一个该执行的目标 BBL,该方法相当于对所有 BBL 都进行了链接。

3.3 内联插桩

内联插桩主要分为两个层面:一个是用户层面,用户在编写 PinTool 的时候可以通过消除简单控制流,比如减少条件语句的使用,使生成的二进制代码跳转语句更少,运行更高效,也可以将分析函数改为 inline 函数,将函数体直接“内联”到调用

处，从而避免函数调用和返回过程中的压栈/出栈等操作，省去调用过程中的额外二进制指令，以提高插桩效率；另一个是系统层面，框架为用户提供了一些简单加法的内联插桩 API，通过直接将待加减的变量加载到寄存器中进行操作，可以有效减少简单分析函数调用的开销和相关内存开销，从而提高插桩效率。

4 测试和评估

本章将龙芯平台上的插桩软件和主流的插桩软件进行了功能对比，并使用 CoreMark 在 64 位 x86 平台上对 Intel Pin 进行了系统和不同粒度的插桩性能测试，其中系统测试时运行整体插桩框架并记录运行效率，粒度测试时运行 INS 级、BBL 级和 RTN 级 3 个粒度的 PinTool 来测试各个粒度在插桩时的性能。同时在龙芯平台上对原框架、使用方法 1 进行基本块链接的 1.0 框架和使用方法 2 进行基本块链接的 2.0 框架进行了同样的测试，并对测试结果作出了分析。

如表 2 所示，龙平台插桩软件相较 DynamoRIO 和 Valgrind 而言，提供了更为丰富的插桩粒度，而和 Pin 相比，则拥有更强的内存动态分配能力，且该插桩软件工具填补了龙架构生态中插桩工具的空白，尤其在龙芯 3A/B/C 系列处理器的指令解码、寄存器分配和原子操作插桩方面实现了底层优化。

表 2 插桩软件功能对比

Table 2 Functionality comparison of instrumentation software

插桩软件	龙平台插桩软件	Intel Pin	DynamoRIO	Valgrind	QEMU
架构支持	原生支持龙架构	x86/ARM	x86/ARM	x86/ARM	多架构模拟
插桩粒度	指令、基本块、镜像、轨迹、函数	指令、基本块、镜像、函数	指令、基本块、函数	基本块	基本块、镜像
动态分配能力	动态重分配策略＋低内存	静态策略	部分动态	高内存	模拟器级高开销
可编程接口	龙架构专用 API	x86/ARM 指令 API	通用 IR 接口	固定工具链	TCG 中间代码
扩展领域	国产化定制(侧信道防御/龙芯 PMU 集成/异构计算加速/性能分析)	安全/性能分析	安全/性能分析	内存检测	动态二进制翻译

CoreMark 的测试程序是一个轻量级的 C 程序，是一个标准的、精简的基准测试工具，旨在提供一个跨平台的一致性性能评估工具。CoreMark 的执行分数是衡量 CPU 性能的关键指标，表示 CPU 在单位时间内完成 CoreMark 测试的次数。使用插桩框架运行 CoreMark，执行分数越高，表明软件插桩性能越好。由于 Intel Pin 和目前已有的插桩软件都无法对龙架构程序插桩，因此这里做了 Pin 针对 x86 架构的插桩测试和不同版本的该框架针对龙架构的插桩测试。主要测试数据如表 3 和表 4 所示。其中：执行效率为执行分数除以原生执行分数(10 524)，原生执行分数是指 CoreMark 直接在电脑系统上运行的分数；Pin 框架指在 64 位 x86 架构上使用 Intel Pin 进行测试，即使用 Intel 官方的插桩软件作为对照；原框架指未使用任何基本块链接技术，只实现简单插桩的最基础软件框架；基本块链接 1.0 指采用了第 1 版基本块链接优化技术的插桩框架，在该版本中，只有 TRACE 的最后一个 BBL 链接在一起，即只有部分 BBL 进行了基本块链接；基本块链接 2.0 指采用了第 2 版基本块链接优化技术的插桩框架，在该版本中，所有的基本块都链接在了一起。在这 4 种框架上使用 CoreMark 测试各粒度的执行效率，插桩框架一列指 CoreMark 在相应的插桩框架上进行执行，但未进行任何粒度插桩的测试分数；INS 级插桩一列表示对 CoreMark 程序进行 INS 级粒度插桩分析之后的测试分数；BBL 级插桩一列指对 CoreMark 程序进行 BBL 级粒度插桩分析之后的测试分数；RTN 级插桩一列指对 CoreMark 程序进行 RTN 级粒度插桩分析之后的测试分数。

表 3 框架 CoreMark 测试分数

Table 3 The CoreMark testing score of framework

框架	原生执行	插桩框架	INS 级插桩	BBL 级插桩	RTN 级插桩
Pin 框架	22 822	19 533	1 859	8 932	8 788
原框架	10 524	47	3	45	47
基本块链接 1.0	10 524	60	51	58	60
基本块链接 2.0	10 524	核心转储	343	1 515	核心转储

表 4 框架插桩效率

Table 4 Instrumentation efficiency of framework %

框架	原生执行	插桩框架	INS 级插桩	BBL 级插桩	RTN 级插桩
Pin 框架	100	85.60	8.100	39.10	38.50
原框架	100	0.45	0.028	0.43	0.45
基本块链接 1.0	100	0.57	0.480	0.55	0.57
基本块链接 2.0	100	核心转储	3.300	14.40	核心转储

原框架和基本块链接 1.0 框架的各级插桩粒度执行效率对比如图 6 所示，原框架、INS 级插桩、BBL 级插桩和 RTN 级插桩的执行分数分别为 47、3、45 和 47。方法 1 链接后插桩框架、INS 级插桩、BBL 级插桩和 RTN 级插桩的执行分数分别为 60、51、58 和 60。因为只进行了最后一个 BBL 之间的链接，TRACE 中可能的 BBL 数有 1、2、3，从 TRACE 中非最后一个 BBL 跳转至新 TRACE 的概率极大，因此各级插桩执行效率虽有提升，但幅度不大。

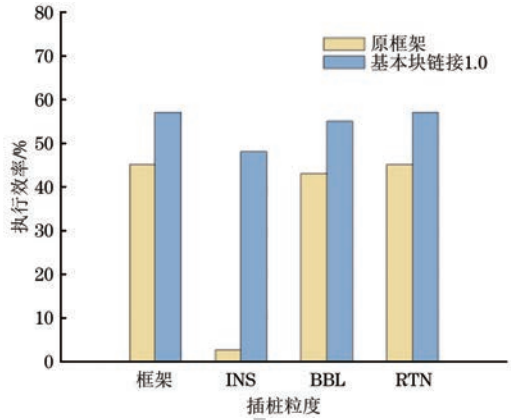

图 6 原框架与基本块链接 1.0 框架执行效率对比

Fig.6 Comparison of execution efficiency between the original framework and the basic block linking 1.0 framework

原框架、使用方法 1 链接后和使用方法 2 链接后的框架各级插桩粒度执行效率对比如图 7 所示。使用方法 2 链接后框架 INS 级插桩分数为 343，BBL 级插桩分数为 1 515，方法 2 对全部基本块进行了链接，不管 TRACE 是从第几个 BBL 跳转至新 TRACE，都会将该基本块与新 TRACE 中的目标 BBL 链接在一起，而为基本块添加新的出口成员则有效解决了链接混乱错误的问题，执行效率提升显著，整体性能有极大幅度的提升，但是 RTN 级插桩却会出现核心转储，核心转储是操作系统在程序异常终止时生成的文件，记录了程序崩溃时的内存状态、寄存器值、调用栈等信息，这表示插桩过程存在程序异常。经过 gdb 调试发现是系统生成的一条 st 指令访问了非法内存，这可能和 QEMU 本身运行有关。

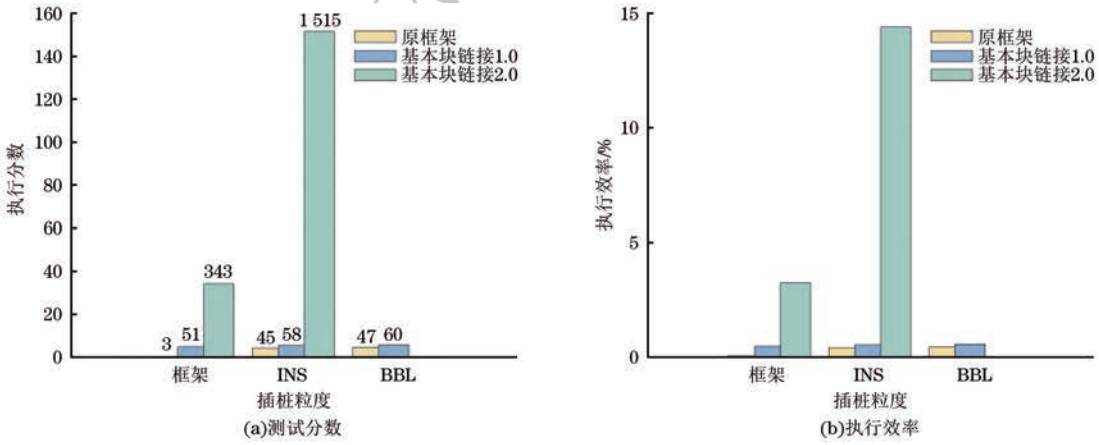

图 7 原框架、基本块链接 1.0 框架和基本块链接 2.0 框架测试分数和效率对比

Fig.7 Comparison of testing score and execution efficiency among the original framework, the basic block linking 1.0 framework and the basic block linking 2.0 framework

5 结束语

本文基于 QEMU 设计了一个与 Pin 高度兼容且支持龙架构的 DBI 软件。阐述了 INS、BBL、TRACE、RTN 和 IMG 5 个基础插桩粒度的详细定义及解析生成过程，以 INS 级的核心插桩 API 为例，介绍了 5 个插桩粒度相关 API 的具体设计过程，同时也描述了插桩工具的详细结构，并基于设计实现的 API 编写了许多 PinTool 实例供用户使用和学习。之后通过对条件跳转指令翻译的修改、基本块链接和内联插桩提高了框架的执行效率，并使用 CoreMark 对框架的插桩效率进行了测试对比，

测试结果显示优化后的框架效率有明显提升。最后，将该 DBI 软件的代码在 GitHub 上进行开源，以供后续完善。与主流架构的插桩软件相比，该软件的插桩效率还有一定的提升空间，且该框架目前实现的 API 都是与 Pin 高度兼容的，未来可以针对龙架构指令特点，实现一些特色化 API 并考虑跨平台运行，进一步扩大使用范围。

参考文献

[1] 胡伟武，汪文祥，吴瑞阳，等. 龙芯指令系统架构技术[J]. 计算机研究与发展，2023，60(1)：2-16.
HU W W，WANG W X，WU R Y，et al. Loongson instruction set architecture technology[J]. Journal of Computer Research and Development，2023，60(1)：16.（in Chinese）

[2] GORGOVAN C，CALLAGHAN G，LUJÁN M. Balancing performance and productivity for the development of dynamic binary instrumentation tools：a case study on Arm systems[C]// Proceedings of the 29th International Conference on Compiler Construction. New York，USA：ACM Press，2020：132-142.

[3] CHEN Z，ZHANG Q，WU J，et al. A source-level instrumentation framework for the dynamic analysis of memory safety [J]. IEEE Transactions on Software Engineering，2023，49(4)：2107-2127.

[4] AÇıCı K，UĞURLU G. A reverse engineering tool that directly injects shellcodes to the code caves in portable executable files[C]//Proceedings of the International Conference on Theoretical and Applied Computer Science and Engineering (ICTASCE). Ankara，Turkey：IEEE Press，2023：42-45.

[5] ARIF M，ZHOU R Y，HO H M，et al. Cinnamon：a domain-specific language for binary profiling and monitoring [C]// Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Seoul，Korea：IEEE Press，2021：103-114.

[6] BELLARD F. QEMU，a fast and portable dynamic translator[C]// Proceedings of the Annual Conference on USENIX Annual Technical Conference. New York，USA：ACM Press，2005：41.

[7] HUANG J C. Program instrumentation and software testing [J]. Computer，1978，11(4)：25-32.

[8] BRUENING D，GARNETT T，AMARASINGHE S. An infrastructure for adaptive dynamic optimization [C]// Proceedings of the International Symposium on Code Generation and Optimization. San Francisco，USA：IEEE Press，2003：265-275.

[9] KIM D，KIM S，RYOU J. Design and implementation of user-level dynamic binary instrumentation on ARM architecture[J]. The Journal of Supercomputing，2018，74(8)：3583-3595.

[10] NETHERCOTE N，SEWARD J. Valgrind：a framework for heavyweight dynamic binary instrumentation [J]. ACM SIGPLAN Notices，2007，42(6)：89-100.

[11] PEREIRA R，STELLE G，CARRIBAULT P. Taskgrind：heavyweight dynamic binary instrumentation for parallel programs analysis [C]// Proceedings of the SC24-W：Workshops of the International Conference for High Performance Computing，Networking，Storage and Analysis. Atlanta，USA：IEEE Press，2025：214-221.

[12] 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.

[13] SRIVASTAVA A，EUSTACE A. ATOM：a system for building customized program analysis tools[C]//Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. New York，USA：ACM Press，1994：196-205.

[14] LUECK G，PATIL H，PEREIRA C. PinADX：an interface for customizable debugging with dynamic instrumentation [C]// Proceedings of the Tenth International Symposium on Code Generation and Optimization. New York，USA：ACM Press，2012：114-123.

[15] BRUENING D. Efficient，transparent，and comprehensive runtime code manipulation [D]. Cambridge，USA：Massachusetts Institute of Technology，2004.

[16] HILL J H，FEIOCK D C. Pin ＋ ＋：an object-oriented framework for writing Pintools[C]// Proceedings of the 2014 International Conference on Generative Programming：Concepts and Experiences. New York，USA：ACM Press，2014：133-141.

[17] CHATTERJEE N，THAKUR S S，DAS P P. Resource management in native languages using dynamic binary instrumentation (PIN)[M]// CHAKI R，CORTESI A，SAEED K，et al. Advanced computing and systems for security. New Delhi，India：Springer India，2015：107-119.

[18] ARAFA P，KASHIF H，FISCHMEISTER S. DIME：Time-aware dynamic binary instrumentation using rate-based resource allocation [C]// Proceedings of the International Conference on Embedded Software (EMSOFT). Montreal，Canada：IEEE Press，2013：1-10.

[19] 林昊，康绯，光焱. 基于 DBI 的密钥安全性检测[J]. 网络与信息安全学报，2017，3(11)：50-58.
LIN H，KANG F，GUANG Y. Key security detection based on dynamic binary instrumentation[J]. Chinese Journal of Network and Information Security，2017，3(11)：50-58.（in Chinese）

[20] 陈金宝，张昱，李清伟，等. DBI-Go：动态插桩定位 Go 二进制的非法内存引用[J]. 软件学报，2024，35(6)：2585-2607.
CHEN J B，ZHANG Y，LI Q W，et al. DBI-go：dynamic binary instrumentation for pinpointing illegal memory references in go binaries[J]. Journal of Software，2024，35(6)：2585-2607.（in Chinese）

[21] 卢帅兵，张明，林哲超，等. 基于动态二进制翻译和插桩的函数调用跟踪[J]. 计算机研究与发展，2019，56(2)：421-430.
LU S B，ZHANG M，LIN Z C，et al. Dynamic binary translation and instrumentation based function call tracing [J]. Journal of Computer Research and Development，2019，56(2)：421-430.（in Chinese）

[22] ZENG J Y，FU Y C，LIN Z Q. PEMU：a PIN highly compatible out-of-VM dynamic binary instrumentation framework[C]//Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. Istanbul Turkey. New York，USA：ACM Press，2015：147-160.

[23] 邹伟，高峰，颜运强. 基于 QEMU 的 DBI 技术[J]. 计算机研究与发展，2019，56(4)：730-741.
ZOU W，GAO F，YAN Y Q. Dynamic binary instrumentation based on QEMU [J]. Journal of Computer Research and Development，2019，56(4)：730-741.（in Chinese）

[24] COTA E G，CARLONI L P. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation[C]//Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. New York，USA：ACM Press，2019：74-87.

[25] GAN C Y. Research on high-reliability and portable dynamic binary instrumentation for linux syscall[C]//Proceedings of the 3rd International Symposium on Computer Technology and Information Science (ISCTIS). Chengdu，China：IEEE Press，2023：725-729.

文字编辑 金胡考
栏目编辑 宋 圆