西安电子科技大学

硕士学位论文

基于动态二进制翻译的逆向调试器的设计与实现

姓名：刘涛

申请学位级别：硕士

专业：计算机软件与理论

指导教师：刘西洋

20080101

摘要

随着现代软件系统复杂程度的不断提高，作为软件错误诊断和程序理解的重要手段之一的调试也变得越来越重要。可逆向调试技术可以支持向后回滚程序状态，使程序状态回到其历史执行点。程序员利用逆向调试器能加速定位程序中的错误和缺陷，从而大大提高软件开发的效率。逆向调试的基础是程序的逆向执行，当前程序逆向执行研究主要集中在程序设计语言级别的逆向支持和通过程序植入来记录和重现程序历史状态等几种途径。

本文研究了利用动态二进制翻译技术对程序进行植入来实现对程序执行状态的记录和重放技术，并在此基础上设计和实现了基于QEMU的逆向调试器PORD。PORD 主要由两部分构成，其一是基于修改的 QEMU 的虚拟机，它用来执行被调试程序。另一部分是提供用户命令界面的 R-GDB，它是在 GDB 的基础上修改而来的，两部分之间通过远程调试协议进行通信，由 R-GDB 解析用户命令并通知虚拟机执行相应的操作。根据为保存程序状态而进行植入方式的不同，PORD 具有 Binary-Translate 模式和 Binary-Copy 两种模式。前者应用在源程序和宿主机拥有不同的指令集架构的情形下，可以为程序员提供跨平台的可逆调试环境。当具有相同指令集架构时，Binary-Copy 模式可以提供与本地执行速度相当的逆向调试环境。

实验证明了本文设计并实现的逆向调试器 PORD 在保证传统调试器功能的前提下能够快速高效的逆向回滚，重现程序的历史状态。

关键词：动态二进制翻译　逆向执行　可逆调试　软件调试

Abstract

With the growing complexity of modern software systems, debugging is more and more important technology to diagnose faults and bugs of software, especially the technology of reversible debugging. A reversible debugger can execute program backward to historical points, which enables programmers to speed up the location of the cause of the program failure and improve the debugging efficiency dramatically. Reversible debugging is based on reversible execution. The current researches of reversible execution mainly focus on the designing new programming language to support reversible execution and the recording and replaying program's historic status through program instrumentation.

This thesis studies a technology of recording and replaying program's status through program instrumentation based the technology of dynamic binary translation. And this thesis designs and implements a reversible debugger called PORD which is on the basis of the technology. PORD mainly includes two parts, one is virtual machine based the modified QEMU to run the debuggee, and another is R-GDB based the modified GDB to interpret user commands. R-GDB communicates with virtual machine through the modified gdb remote debugging protocol. R-GDB interprets the user command and notifies the virtual machine to execute corresponding actions. PORD provides two modes in reverse execution: Binary-Translate and Binary-Copy mode. Binary-Translate mode is applied to the case that the guest application and the host platform possess different instruction-set architecture. This mode enables the cross-platform reverse debugging. When the guest application and the host platform are constructed with the same instruction-set architecture, Binary-Copy mode is employed.

The experiment in this thesis shows that PORD can backtrack to historic point fast and efficiently to reconstruct program's historic status.

Keyword: **dynamic binary translation** **reverse execution**
reversible debugging **software debugging**

西安电子科技大学

学位论文创新性声明

秉承学校严谨的学风和优良的科学道德，本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果；也不包含为获得西安电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中做了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切相关责任。

本人签名：刘涛　　日期 2008.1.7

西安电子科技大学

关于论文使用授权的说明

本人完全了解西安电子科技大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属西安电子科技大学。学校有权保留送交论文的复印件，允许查阅和借阅论文；学校可以公布论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存论文。同时本人保证，毕业后结合学位论文研究课题再撰写的文章一律署名单位为西安电子科技大学。

（保密的论文在解密后遵守此规定）

本学位论文属于保密在___年解密后适用本授权书。

本人签名：刘涛　　日期 2008.1.7

导师签名：刘西洋　　日期 2008.1.7

第一章 绪论

1.1 研究背景

本文的研究工作来源于西安电子科技大学软件工程研究所的“十一五”国家某项目，可逆向调试技术是其中的一个重点研究课题。

随着信息技术的飞速发展，以及 IT、互联网、电信的融合，使得软件系统无处不在。软件系统不仅担负着越来越重要的任务，而且这些软件系统也日益复杂化、大型化。程序中出现故障或错误是无可避免的，尤其是庞大的软件系统，调试是程序员用来诊断错误并修改错误的一个极其重要的手段[1]。不可否认，调试也是一门科学，调试对于软件缺陷和错误的诊断以及程序的理解来说，是不可或缺的一种重要手段，但是在现今绝大多数的程序员眼中，调试还是更多的停留在经验的范畴之内。为了减轻程序员在程序调试上所耗费的精力，软件调试技术一直在朝着准确化、自动化方向发展。但是，现今软件的调试或多或少存在一些问题阻碍着其向前发展。

因为操作系统固有的特性，在调试过程中，被调程序如果出现了错误并不会立即崩溃掉，相反地，它可能会一直执行下去直到操作系统的违例操作出现，程序才会被终止。通常为了诊断出程序缺陷的位置，整个调试的过程是一个多次的不断迭代过程：(1)首先通过人工经验分析软件中某个可能有问题的位置，并设置一个断点，(2)然后启动新的调试会话，执行程序到断点处，检测此时程序的状态是否跟预期一致，(3)分析检测的结果，如果已经能够准确定位错误之所在，那么就不需要再重新设置断点进行迭代了，否则就必须多次地重复整个调试过程直到找到错误的根源为止。通过调试过程中对软件行为的多次的重现，程序员可以准确地跟踪导致软件出错的根源，从而进一步对其进行修正，消除缺陷。然而对于程序员来说，整个调试过程是非常单调、重复和耗时的。实际上，随着系统的庞大和集成的模块的增多，整个调试过程中的时间、人力和物力的耗费将会越来越大，甚至可能会达到无法控制的地步。

另外，当传统的调试软件面对大型软件系统，尤其是长时间运行的系统就会显现出一些弊端，例如数据库系统需要运行“一段时间”之后才能处于稳定的状态，调试一次耗时过长就可能出现很大的不确定性，现场信息极易被调试活动所破坏，从而导致程序执行的历史状态难以重现。由上述分析可知，调试器需要有忠实快速地重现程序的任意历史执行状态的能力。逆向执行可以通过各种手段使得程序可以完全一致地逆向回去到某个执行过的历史的点，它将能够为程序员提

供一种快速定位错误源头的重要方法。这样，将大大减轻程序员在软件调试上的负担，从而降低软件开发上的巨大的人力物力的耗费。

在软件工程领域，自从M.V.Zelkowitz[2]于1973年提出逆向执行这一概念以来，在这方面的研究就一直没有中断过。尤其是进入二十一世纪以来，在计算机的计算速度和计算能力得到大幅提升的情况下，可逆向执行的研究所遇到的一些诸如存储代价和时间开销等问题都在慢慢地得到解决。

与此同时，虚拟化技术在当今计算机硬件能力不断提升的前提下也得到了突飞猛进的发展，例如服务器集群的虚拟化、存储设备的虚拟化和嵌入式芯片虚拟化等。单就软件虚拟化来说，就有应用程序级别的虚拟化和系统级的虚拟化。程序级别虚拟化可以使得程序以进程的方式运行在不同指令集架构平台上，而系统级虚拟化可以在裸机上运行多个不同的操作系统平台，达到硬件资源的充分利用。程序在虚拟机上虚拟运行，这样通过虚拟机就可以收集程序在运行过程中的运行状态和执行路径等轮廓数据，并且完完全全控制程序的执行。二进制翻译技术是软件虚拟化技术中比较成熟的一种技术，它是一种直接翻译可执行二进制程序的技术，并把一种处理器上的二进制程序翻译到另外一种不同指令集架构的处理器上执行。正是利用虚拟机的虚拟执行这个特性，使得可以通过在翻译过程中植入特定的额外操作或者是直接利用虚拟机来记录保存程序执行状态，然后通过控制虚拟机重建程序历史执行状态，并在此基础上检查程序状态找出程序错误的根源。

1.2 国内外研究现状

逆向执行技术从上世纪八十年代提出至今，已经取得了长足的进展，尤其是在虚拟化技术蓬勃发展的今天，逆向执行技术的部分研究成果已经走出了实验室进入了商用阶段。

M.V.Zelkowitz[2]于1973年提出可逆向的执行的概念，其最初的目标就是使程序员在程序执行的过程中可以控制程序按照自己的意愿回溯到任意历史执行点。为了实现这个目标，他提出了在程序设计语言层面提供回溯或逆向语法的构想。其后C.Lutz[3]在此基础上于1986年提出了可逆向的语言Janus，并提供了可逆语言的语法定义。2007年，Tetsuo和Robert[4]在其论文中证明了Janus语言的可逆向执行，并实现了一个自翻译的Janus解释器，用于程序的逆向执行。另一方面，通过程序植入创建程序执行历史日志的逆向执行研究也进行的如火如荼。1989年，Stuart I. Feldman和Channing B. Brown实现了一个Motorola 68000平台的可逆向调试器IGOR[5]，通过修改编译器、加载器以及静态库的方式在程序中植入代码，并且引入周期性的检查点来记录内存页和文件块，从而实现了比较快速的逆向执行。PROVIDE[6]为C语言的子集提供了一个虚拟的调试环境，并且程序在PROVIDE的

解释器上执行，解释器将整个程序执行状态记录到数据库中。虽然PROVIDE仅仅提供了C语言子集的逆向调试，但是它提供了从虚拟执行方向来记录程序执行日志的方法。ZStep94[7]也实现了Lisp的源代码步进工具，同样ZStep94通过其特定的解释器保持了程序的历史状态。

另一方面，Chen等人[8]和Algul等人[9][42]分别在其论文中提出了基于汇编代码植入的方法。他们通过植入汇编代码记录下了所有即将改变的状态。但是，这样会带来一个问题，就是记录这些状态需要很多的存储空间和读写时间。动态切片技术[10][37] 对此作了小幅的改进。动态切片技术把状态进行了等价划分，降低了空间存储的需求量，但是这带来一定的运行时间负载。LVM[11]是一个提供了逆向执行的功能的轻量级虚拟机，它将原子性的事务作为一个可逆向执行的最小单位，并且事务的长短可以按照需求在执行的过程中变化。但是在LVM上执行的二进制程序必须先利用其自身的Leonardo C编译器编译，然后才能够执行。RecPlay[12]实现了非确定性的多线程并发程序的逆向调试，通过植入内存访问记录的汇编指令，在运行过程中自动检测数据竞争(Data Race)，并支持与传统的调试器的接口。

由于计算机硬件价格的降低和现在处理器的运算能力的不断提升，虚拟机得到快速迅猛的发展[45]，出现了以动态二进制翻译为主的全虚拟化模式和以 Hypervisor 层的 VMM 监控为主的半虚拟化模式以及虚拟硬件支持的基于 Hypervisor 的全虚拟化模式[36][38]。Peter.Chen[13][44]基于 UMLinux 虚拟机实现了操作系统的可逆向调试器 TTVM(Time-Travel Virtual Machine)。TTVM 间隔一定时间做一个检查点并通过虚拟机去记录操作系统的所有状态，提供了 reverse step, reverse breakpoint, reverse watchpoint 等逆向调试命令，使得在逆向重放阶段按照时间先后逐个恢复检查点状态。Toshihiko Koju 等人[14]于 2005 年提出了一个高效和通用的基于虚拟机方案的逆向调试器，调试程序运行于虚拟机之上，提供了 Native 模式和 VM 模式。在 Native 模式下将不会进行状态的记录，两种模式之间可以快速切换，虚拟机首先翻译被调试程序，并在需要植入代码的地方对程序进行代码植入，然后执行翻译好的代码，将记录下来的状态日志存入文件缓存中，通过读取这些日志文件缓存重构程序历史状态。Virtech 公司于 2007 年初推出了其嵌入式软件开发平台 Simics[15]，包含了开发工具、模拟器和调试器等，其嵌入式的支持多种平台的调试器也开始支持程序的逆向回滚执行。

相比国外对逆向执行技术的研究进行得如火如荼，国内在这方面的研究还是处于起步阶段。

1.3 论文的研究问题

在程序的逆向执行技术中，大致上有几种途径可以实现基本的程序逆向执行。

一是通过产生逆向码，在需要逆向执行的时候执行生成的逆向码使得程序返回到以前的状态，这是一种基于程序源代码的直接想法，但是所能够解决的问题比较有限，仅仅是一些简单的算术逻辑代码，而且逆向代码必须在程序的高级源代码上才能生成。二是利用类 Unix 系统中的 fork 系统调用产生子进程，而子进程是对父进程所有状态的一个完整复制。在程序不同的执行点上产生新的子进程后，在需要逆向的时候，直接跳到想要恢复的点所对应的子进程就可以了。这种方式必须是基于类 Unix 系统，方法比较巧妙，但是灵活性可移植性不高，对于比较大的程序需要做更多检查点的时候就会耗费系统大量的资源。三是在程序动态执行的时候对程序状态进行记录，通过这些记录下来的状态日志重建程序历史时刻的状态，达到逆向目的。一般来说，都是通过程序植入来实现记录程序执行状态的。相对而言，这是一种比较直观的方法，而且就现在的情况来看，也是比较成熟的一种方式。这也正是我们的研究方向，利用程序植入跟踪记录程序执行状态的代码，并将传统调试工具结合进来实现一整套完整的逆向调试工具。

我们将逆向调试工具定位为能够调试不带调试符号信息的二进制可执行程序，而且要求被调试程序在调试器上能够以可以接受的速度运行，同时在被调试程序执行结果的正确性不受植入代码的影响的前提条件下，结合成熟的传统调试工具控制被调试程序正确无误地重现任意历史执行点的程序状态。正是基于这样的考虑，我们采取基于动态二进制翻译的虚拟化机制，使得程序在一边执行一边翻译的过程中进行有选择的代码植入和程序状态的记录保存。对于调试器后端，采用 linux 系统传统的调试器 GNU Debugger(GDB)[16]，并在虚拟机中加入 GDB 的调试服务端同 GDB 进行远程调试通信。

1.4 论文的主要工作

在 1.3 节对研究问题分析的基础上，本论文着眼于如何在动态执行的过程中在最小影响范围下进行被调试程序的植入，并且将传统调试器 GDB 同逆向执行部分结合起来。因为本文提出的可逆向调试器是针对语言无关的可执行的二进制代码，所以选取动态二进制翻译类型的虚拟机为被调试程序的执行环境，在虚拟机内加入程序植入模块，在程序被翻译和执行的过程中植入存储状态的代码。在此基础上需要把 GDB 的服务桩模块调整加入到虚拟机中，以便同 GDB 的前端进行命令交互调试通信。如图 1.1 所示，我们实现了基于动态二进制翻译技术的逆向调试器 PORD（Portable and Optimized Reversible Debugger)。PORD 主要分为两部分：同用户进行命令交互的调试器后端 R-GDB（Reversible GDB）和用于运行被调试程序的虚拟机核心部分。

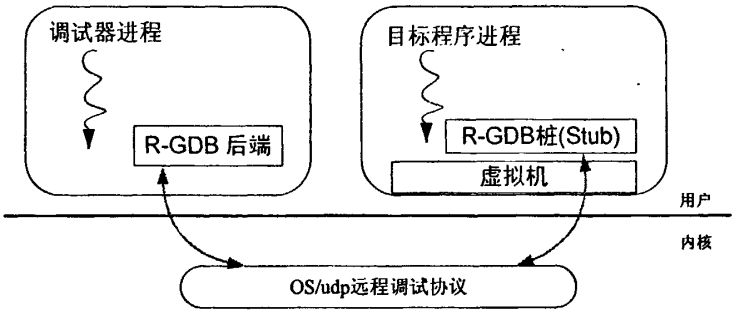

图 1.1 PORD 基本组成部分

在调试器后端 R-GDB 部分中，主要是分析 GDB 调试器的远程调试协议部分和远程调试运行流程，并修改 GDB 远程调试通信协议和增加逆向执行的命令，如 reverse <line number>等。对于运行被调试程序的虚拟机核心部分来说，主要是分析了动态二进制翻译技术原理，并在动态二进制翻译模拟器 QEMU[17]的基础上加入程序动态植入模块和历史状态管理模块，修改 GDB 的桩程序以响应远程调试协议数据包，并根据命令在虚拟机上重建程序历史执行状态。

1.5 论文的组织结构

本文共分六章，第一章为绪论部分，主要讨论了本文的研究背景，国内外研究现状，本文所研究问题和主要工作，以及我们实现的逆向调试器 PORD 的基本组成部分；第二章介绍了基于动态二进制翻译的逆向调试相关技术和基本知识，给出了二进制翻译和逆向调试的研究热点和技术手段；第三章讨论了基于二进制翻译模拟器 QEMU 的逆向执行技术；第四章详细的阐述了逆向调试器 PORD 的设计与实现；第五章在对第三章和第四章中提出的逆向调试器 PORD 进行不同检查点间隔粒度和不同翻译模式的实验，得出实验分析和结论；第六章对本论文进行了总结并对进一步的工作进行了展望。

第二章 基于二进制翻译的逆向调试相关技术

随着软件系统的日益复杂和庞大，调试在软件开发中发挥的作用越来越大。而现在传统的调试对于多线程并发、异常中断等非确定性的状态很难以进行重现，并且对于长时间运行软件，重新启动一次调试会话都将等待很长的时间，所有这些都将为程序的开发增加大量的人力物力消耗。逆向执行将大大改变这一局面，但是就现在已有的研究来说，逆向执行并没有完全被传统的符号调试器很好的利用，而且可以执行类似 Undo 功能的调试器在调试会话期间的性能又不能让人满意[18]。

2.1 可逆向执行技术

逆向执行就是使得程序可以折返或回滚到程序一个先前的状态[2]。尽管逆向执行对传统的调试器而言是一个非常有用的特性，可是大部分传统调试器都并没有实现这项功能，不外乎都是因为逆向执行需要庞大的记录和存储历史数据的空间，并且在执行过程中有大量的运行时时间负载。其实在软件领域已经存在很多的类似回滚、Undo 的研究，例如数据库系统中的事务处理和回滚、文件编辑中的 Undo 和 Redo 等，都是在时间和空间影响小的前提下增加的非常有用的功能。

从上个世纪八十年代提出逆向执行的概念以来，如何减小历史数据的空间存储和运行时的额外时间耗费，一直以来都是软件工程和系统方向研究的重要课题之一。主要有两方面的研究，一是直接构造可以逆向的程序和执行环境，主要是在语言层面，修改编译器、加载器或者解释器使得程序中包含可以逆向执行的指令。由于各种原因，现今在语言方向的逆向研究仍然处于实验室验证和模拟状态中，暂时没有能够成熟应用的实例，更多的还是一个研究的框架和工具，例如 Jikes RVM[19]。二是转换已经存在的不可逆向执行的程序为可以逆向的程序。这一方向的研究大多是以程序历史状态的记录和重放为基础，如何减小程序记录和重放对源程序以及程序执行时间和存储空间等造成的影响[43]。基本上来说，有三种方式在这一方向上做出了大量的研究和成果，（1）通过源程序生成逆向执行代码，在逆向时，执行这些已经生成的对应的代码。（2）利用类 Unix 系统中 fork 系统调用创建子进程，因为 fork 调用创建的子进程是其父进程的副本。（3）记录程序状态为日志文件，重放时利用记录下的日志重建程序历史状态。这也是现今比较成熟和直接的一种方式。本文实现的逆向调试器就是基于这种方式。所以在本文中更多的是关注关于程序状态日志文件的记录和重放这种方式。

2.2 可逆向执行的技术途径

在 2.1 节中已经给出了实现逆向执行的相关定义和基本分类，本节就其中的最主要的几种研究和实现方法给出详细的说明和解释。

2.2.1 逆向代码的生成

所谓的逆向代码生成就是为程序生成其对应的“反”代码，通过执行这些“反”代码回溯到程序的历史状态[20]。其基本思想就是记录一个程序位置的指针并通过当前的所有变量的值和前面执行的指令，计算出这个变量的前一个历史值，前一个执行的指令可以通过记录下来的程序位置指针获得。要获得程序的逆向代码，必须有程序的高级源代码存在，所以这个逆向执行方式需要存在高级源代码。例如，下一个将要执行的指令 $x := y - i$，我们可以获得其相应的逆向代码为 $y := x + i$。

为了更好的产生相应的逆向代码，例如 C 程序的结构可以被分成以下几类，每一类都有不同的逆向代码生成方式。

1) 顺序指令，例如赋值指令。
2) 选择指令，例如 if-else 结构，case 结构。
3) 迭代指令，例如 while、for 结构。
4) 非结构化指令，例如，goto，break 指令等
5) 函数调用指令

以下给出一个示例简单介绍一下逆向码的处理：

原始程序		记录下的 TraceFile		逆向程序处理	
6	a=2;	1	a=0;	6	/*读历史记录*/
7	b=-1;	2	b=0;	7	/*读历史记录*/
8	while(a>0){	3	Goto_Line 7;	8	/*读历史记录*/
9	b=a%2;	4	b=-1;	9	/*读历史记录*/
10	a--;			10	++ a;
11	}	5	Goto_Line 10;	11	/*读历史记录*/
		6	b=0;		
		7	Goto_Line 10;		

图 2.1 while 结构的逆向执行码生成

产生的 TraceFile 和逆向程序如图 2.1 所示，逆向执行的时候，如果是一般诸如 $a = b\ op\ c$;模式的赋值语句，将直接从 Tracefile 中读取历史记录值；而诸如 $a--$; 语句可以将其符号左移并取反得 $++a$; 同时对于原始程序中第 11 行和循环体结束部分，Tracefile 中将其前一指令均记为第 10 行，使得程序在 while 循环中可以回

滚到前面执行过的语句。

从示例可以看出逆向代码生成实现逆向程序执行还是需要大量的变量状态的记录的，只是对于简单且存在逆向语句(Inverse statement)的那些源代码才能够通过逆向码的执行达到恢复历史状态的目的，但是大量的代码只能够通过记录程序的状态才能够做到恢复历史状态。

2.2.2 类 Unix 系统调用 fork

在 Unix 以及类 Unix 系统中，一个现存的进程可以通过调用 fork()系统调用（syscall）来创建一个新的进程。由 fork 创建的新进程被称为子进程，子进程是父进程的一个拷贝，也就是说，子进程获得父进程的进程数据空间、堆和栈的复制品，但是父子进程并不共享这些数据空间，只是拥有创建时刻相同的复制品。而且一般来说，fork 使用写时复制（Copy-On-Write）技术，所以只要在子进程里面不做更改，那么它将拥有在创建进程的时刻与父进程相同的数据空间、堆和栈。

正是利用类 Unix 系统中 fork 系统调用的这个特性，GNU Debugger（GDB）在其最新测试版中推出了逆向调试的试验性版本。GDB 在调试过程中通过检查点或 cp 命令在当前断点处设置一个检查点（Checkpoint），其核心是通过 fork 在当前产生一个新的子进程，但是不启动这个子进程，也就是说对被测试程序的进程做一个数据空间、栈和堆的复制，实际上就是记录下了程序在此刻的执行状态。然后在未来的某时刻，如果需要逆向回到某个历史检查点，就可以通过 restart <checkpoint number> 这个命令直接跳到这个检查点所对应的子进程上去执行。

可以说，利用类 Unix 系统调用 fork 的这个特性进行状态的记录，对 GDB 本身改动比较小，因此巧妙的实现了逆向调试功能。但是这种方法也有其自身的局限性，（1）系统调用 fork 是一个非常耗费资源的操作，这种方法是不能够调试非常庞大的程序。（2）在调试过程中，只能够通过命令触发的方式保存程序检查点，也就是说并非自动保存检查点，这样程序很难快速并且准确的回滚到任意的程序历史执行点上去。（3）在经过一次回滚之后，因为程序流程已经跳到子进程上执行，而父进程被阻塞，因此不可能再逆向回到其他的检查点上去。

2.2.3 程序状态日志记录和重放

程序状态日志记录和重放技术主要是通过各种方法使得程序在执行过程中输出程序的执行状态（包括执行路径、内存空间、栈、堆和其他如文件、套接字等系统资源状态），然后在需要逆向的时候，通过读取这些记录下来的状态日志文件，重建程序历史执行点的状态。从源程序是否被改变的角度来说，有通过修改源代

码植入的记录状态的代码和不改变源程序通过外界影响来记录状态的代码两种。从是否需要重新编译的的角度来说，分为需要特别的编译器重新编译的源代码和二进制程序两类。因为本文目标在于处理常见的二进制可执行程序，所以关注的对象是只有二进制可执行程序且不需要再编译的程序的逆向执行的研究。

2.3 二进制翻译技术

二进制翻译是指使用软件将一种体系结构的二进制代码翻译成另一种体系结构的指令；可以将一种操作系统上的软件翻译成另外一种操作系统软件；也可以将一种二进制文件代码翻译成另外一种二进制文件代码。它可以很好的解决软硬件之间的矛盾，从而推动计算机技术的发展。从本质上说，二进制翻译技术也是一种编译技术，其处理的目标是一种二进制代码，经过二进制翻译处理之后生成另外一种机器的目标二进制代码。

按照传统的编译器前、中、后端的划分，二进制翻译在概念上也可以分为前端解码器、中段分析优化器和后端优化编码器三个阶段。前端解码器根据源机器的指令结构特点，以及可执行文件的格式规定，通过指令模式匹配对二进制码进行处理，完成类似反汇编的功能。中端分析优化器的功能是完成两级中间表示的转换工作，逐渐去除代码中源机器特性，实现到目标机器的转化，并且对二进制程序进行分析和部分优化。后端优化编码器类似于一般编译器，其功能是从一种中间语言生成优化的目标机代码，主要是完成前端到机器相关代码的转换。

2.3.1 二进制翻译方法分类

一般来说，基于软件的二进制翻译方法，可以分为三类：解释执行、静态翻译、动态翻译。

解释执行对源处理器代码中的每条指令实时解释执行，即翻译一句执行一句，系统并不保存也不缓存解释过的指令，不需要用户的干涉，也不进行任何优化。软件解释器相对来说容易开发，比较容易与老的体系结构高度兼容，但其代码的执行效率非常差[21] [22]。

静态翻译是在源处理器二进制代码执行之前对其进行翻译，将源机器上的二进制可执行程序文件 A 完全翻译成目标机器上的二进制可执行程序文件 B，然后在目标机上执行程序 B。因为可执行文件 B 已经保存下来，所以一次翻译的结果可以多次使用。静态翻译器离线翻译程序，有足够的时间进行更完整细致的优化，代码执行效率较高。然而，因为程序中存在的间接跳转和间接调用等运行时才能确定的因素，使得静态翻译器可能无法完整地翻译一个程序[23]，因而需要依赖解

释器的支持；静态翻译器需要终端用户的参与，这给用户使用造成了很大不便。

动态翻译则是在程序运行时对执行到的二进制程序片断进行翻译，克服了静态翻译的一些缺点，例如，不能知道控制流中某点的寄存器或内存的值，因此不能实现代码挖掘和继续翻译等；动态翻译还可以解决大部分实际情况中的代码自修改问题，而这在静态翻译是不可能的发现和解决的；动态翻译可以利用执行时的运行时动态信息来发掘静态编译器所不能发现的优化机会；动态翻译器对用户可以做到完全透明，无需用户干预，是现在研究和实用领域比较常用的一种二进制翻译手段。

简单的说，解释执行就是一下子只翻译一条指令，静态翻译全部翻译完所有指令，动态翻译介于这两者之间，一次翻译一小段代码，并对翻译好的代码进行缓存。总结一下这三种二进制翻译手段的优缺点如表 2.1 所示。

表 2.1 二进制翻译方法的优缺点比较

翻译方法	优点	缺点
解释执行	易于开发，不需要用户干涉，高度兼容	代码执行效率很差
静态翻译	离线翻译，对代码能够做更好的优化，代码可以重用，代码执行效率较高。	依赖解释器、运行环境的支持，需要终端用户的参与，给用户使用造成了不便
动态翻译	无需解释器和运行环境支持，不需要用户参与，可利用动态信息发掘优化机会	翻译的代码执行效率不如静态翻译高，对目标机器有额外的空间和时间开销

2.3.2 二进制翻译系统框架

本文的翻译方法主要是动态二进制翻译，本节就一般进程动态二进制翻译器框架进行详细介绍。

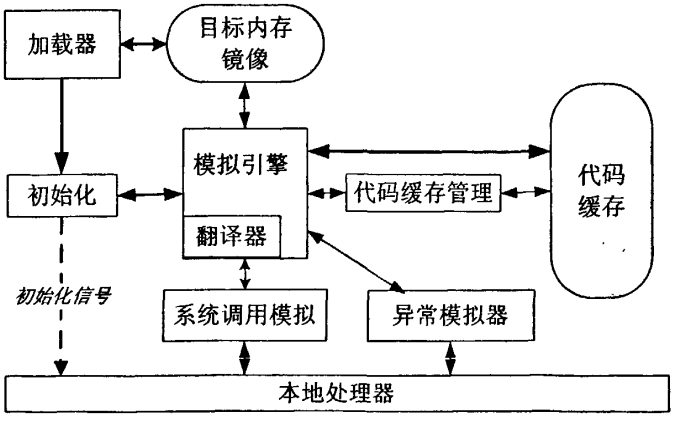

图 2.2 动态二进制翻译框架

图 2.2 是一个应用程序进程级的动态二进制翻译器框架。其各主要的功能模块

如下：

1. 加载器把目标代码和数据写到内存中目标内存镜像中，并且加载虚拟机的代码。尽管目标内存镜像中存放了应用程序的所有代码和数据，但是对于翻译器来说，它们都是数据，因为目标代码没有被直接运行。
2. 加载完毕后，控制移交给初始化模块。初始化模块为模拟过程中用到的代码缓存分配内存空间。初始化同时也唤醒本地操作系统来建立为所有的陷阱创建信号处理程序。初始化完毕之后，模拟过程开始。
3. 模拟引擎使用翻译器来模拟目标代码。代码缓存中的内容是由翻译器写入的。在整个模拟过程中，代码缓存中存放的是可执行代码。
4. 由于代码缓存的大小有限，当新的翻译代码生成时，由代码缓存管理器负责决定清除缓存中的那部分代码来让出空间。
5. 在模拟过程中，目标程序会执行系统调用，这时候，系统调用模拟器会把它翻译成恰当的一个或一组本地系统上的系统调用，并且处理与该调用相关的返回信息。
6. 虚拟机利用异常模拟器处理在翻译器执行时或者已翻译的代码执行时出现所有的陷阱和中断。在一些情况下，当模拟过程出发的陷阱导致本地操作系统向虚拟机发出信号时，异常管理器会获得控制权，并进行相应得处理；其他情况下，模拟器检测到异常状态就会跳转到异常处理。

由图 2.2 可以看出，动态二进制翻译器在翻译并执行代码的过程中，可以捕获程序的各种状态。在本文后续部分将详细阐述如何去捕获这些程序运行时的各种状态。

2.4 二进制植入技术

二进制植入技术在跟踪程序的运行路径和运行状态，并对程序进行优化等方面有着重要的作用。一般来说，植入包含两部分，一是决定哪里或者哪些代码需要被植入，二是植入点需要执行的代码，即被插入的代码部分。也可称这两部分为植入代码和分析代码。

2.4.1 静态二进制植入技术

所谓的静态二进制植入就是先不执行程序，通过静态扫描代码找出相应的植

入点，同时利用静态分析技术得到分析代码，然后将分析代码插入到对应的植入点，完成代码植入。在源代码级别，静态植入一般在程序的抽象语法树（AST）上进行，该方法首先需要得到程序的抽象语法树，然后遍历语法树，找出植入点，同时在语法树上分析得到植入代码，然后直接在语法树上以子树的方式插入植入代码的语法树片段，最后将语法树还原成源代码。对于二进制代码也是类似的分析，以下我们就以静态二进制植入的典型 ATOM[24]以及其后的 EEL[25]、Etch[26]为例详细阐述静态二进制植入的具体方式。

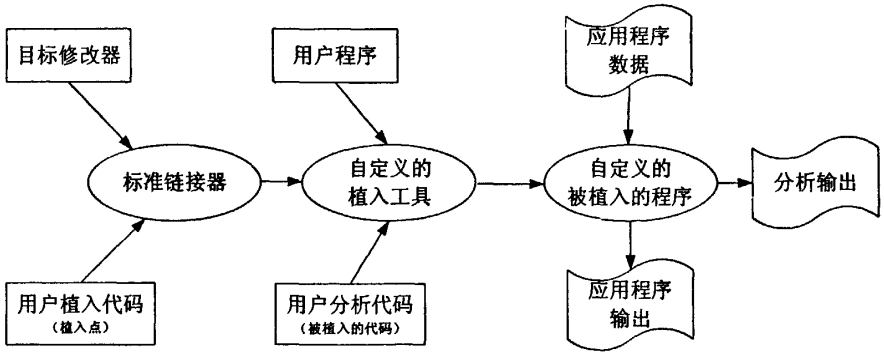

图 2.3 静态二进制植入流程

图 2.3 是 ATOM 静态植入工具的工作流程。ATOM 将用户将要植入的点称为植入代码，将被植入在植入点的代码称为分析代码。首先，合并用户进行植入操作的代码函数并将其编译后加入一个自定义的植入工具中。这个工具将根据在用户自定义的植入代码对应用程序进行植入。然后，利用自定义的工具结合应用程序创建一个被植入的可执行的应用程序。

静态二进制植入有一系列难以避免的缺陷阻碍着其进一步发展。最严重的就是静态植入不能够区分可执行代码中的数据和代码，静态工具没有足够的信息去区分两者。另外程序中还存在间接跳转、间接调用、共享库文件和代码自修改等问题。这些信息都是运行时才能够确定的，因而静态植入方式是无法解决这一类问题的。

2.4.2 动态二进制植入技术

正因为程序中有很多诸如间接跳转和代码自修改等问题在程序不运行的时候没有办法确定，所以出现了运行时对程序植入，即动态二进制植入。一般来说，有两种方式实现动态植入：基于探针的技术（Probe-based）和基于即时编译技术（jit-based）的动态二进制植入。动态二进制植入在程序的导向和程序路径分析[40][41]等方面都有广泛的应用。

2.4.2.1 基于探针的动态植入

基于探针模式的动态植入借鉴了静态二进制植入中对程序的植入方法，覆写内存中的代码段数据，使得程序跳转到被植入的分析代码中执行分析操作，在完成操作之后再跳转回原覆写地址，继续往下执行。这种技术被称为“跳床”技术（Trampolines）。

Dyninst[27]是基于这类技术的典型代表，按照对植入点的先后，有在植入点前执行分析代码和植入点后执行分析代码两类。如图 2.4 所示，通过修改加载到内存中的应用程序代码跳转到 Base Tramp（基本跳床）中，先根据用户定义是在指令前还是在指令后植入，通过跳转到 Mini-Tramp（小跳床）上执行分析代码，图中是在指令前执行分析代码。为了防止当前程序执行状态被植入的分析代码所影响，在 Mini-Tramp 上必须先保存当前所有寄存器，并建立参数列表，然后才能执行用户自己定义的分析代码，在执行完分析代码函数之后还需要恢复执行分析代码前的保存的寄存器。然后控制流回到 Base-Tramp，执行被覆写的原指令，最后通过 Base-Tramp 跳回到应用程序继续执行。

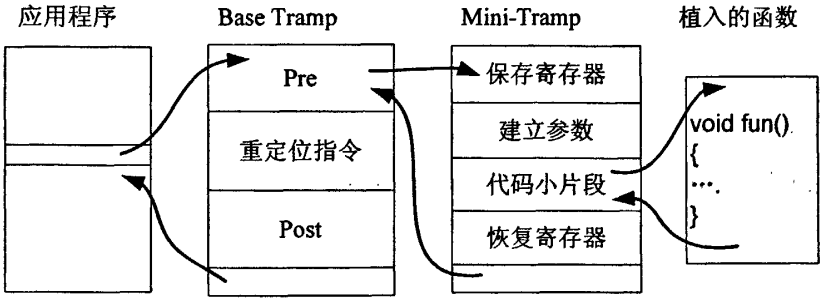

图 2.4 Dyninst 中使用的“跳床”技术

但是基于探针的动态二进制植入技术也有自己的不足之处。第一，整个植入对程序员不是透明的，因为在内存里面的指令将被覆写用来利用“跳床”技术(trampolines)跳到植入的分析代码中。第二，在某些指令变长的系统上，例如 x86 平台，更多的时候指令的长度要小于利用“跳床”技术要替换的指令的长度，从而覆写多条源代码指令，导致植入不成功。第三，跳床技术的实现大多需要多层的分支调用才能够实现，所以造成开销很大。虽然有这些缺点，但是基于探针或跳床技术的动态植入技术应用比较广泛的，如果对于植入粒度要求不是很细的应用还是非常高效和简单的，例如微软研究院的 Detours[28]植入工具，主要用于 Windows 下函数的植入，在函数调用流的跟踪方面有很大的实用价值。

2.4.2.2 基于即时编译的动态植入

随着计算机硬件的成本的降低和计算速度的提高，即时编译技术（JIT，Just In Time）也呈越来越成熟的势态。Java 的虚拟机 JVM 成功地应用在服务器、个人桌面应用和网络服务等重要领域。

Dynamo[29]是 HP 公司开发的一个动态优化器的原型。它输入二进制可执行代码，通过解释执行并观察程序的行为而不需要任何采样代码，并且不需对代码进行预分析。解释程序时收集的 Profiling 信息可以用来动态选择执行的 Hot-Path，然后对这些 Hot-Path 进行编译和优化，并存储在缓冲区中，当下次再被执行的时候就不必解释执行了，直接从缓存区中执行就可以了，从而使程序的执行效率得到大幅度的提升。

DynamoRIO[30]是利用动态翻译在植入上的扩展，在动态二进制翻译的实现上与早期的 Dynamo 有些不同,它的所有程序代码都是在代码级执行。应用程序代码被分成一个个基本块(Basic Block)。在每个基本块结束时,RIO 重新获得控制权并检查控制转移指令的目标.如果目标不在基本块缓存中,RIO 就会把它放人代码缓存中并把执行权交给新的基本块缓存。并且 DynamoRIO 还向外提供了丰富的 API 接口用于对程序进行优化和植入。

同 DynamoRIO 类似，Pin[31][32]和 Valgrid[33]都是基于即时编译的动态植入的工具。它们都向外界提供了丰富的 API 接口，在这些 API 接口的基础上，可以实现很多通过植入实现的工具，例如内存泄露监测、程序优化分析等。正因为这些基于即时编译的动态植入工具提供了如此丰富的 API 接口，提高了灵活性，但是也牺牲了代码执行性能，快速应用于试验性系统还是非常不错的，实际应用就要考虑一下了。

本文就是以基于即时翻译的动态二进制翻译器 QEMU[17]为基础，对 QEMU 进行修改以支持植入来达到我们对程序进行逆向执行目的的。有关 QEMU 的详细技术将在后面讲到。

2.5 本章小结

本章主要介绍和讨论了基于二进制翻译的逆向调试技术所需的相关技术及其优缺点。在已有的逆向调试技术中，基于程序状态日志记录和重放的逆向执行技术最为成熟，Virtech 公司[15]已经利用这种技术实现了嵌入式系统开发工具 Simics，在 Simics 中提供了逆向调试的功能。利用类 Unix 系统的 fork 调用特性实现的逆向执行最为简便和巧妙，但同时灵活性也最小。GNU Debugger[16]也在其

最新版本中增加了利用这种技术实现的试验版的逆向调试功能。基于逆向码生成技术的逆向执行因为其固有的复杂性和不完全性，其所有的成果基本还是处于研究阶段，并未达到可以实用的阶段，但是因为其减少了程序状态存储在空间和时间上的大量开销，因此也是今后的一个研究方向。本文主要对基于动态二进制翻译和植入技术的程序状态记录和重放进行了研究。所以在本章也对二进制的翻译技术和植入技术进行了探讨。

第三章 基于 QEMU 的逆向执行技术

通过第二章对逆向执行相关技术进行的分析和探讨，本文在动态二进制翻译和二进制植入技术基础上，提出了进行程序状态日志记录和重放为实现手段的逆向调试技术。QEMU 是典型的基于动态二进制翻译技术的模拟器，本章将对基于 QEMU 的逆向执行技术的实现方法进行探讨。

3.1 动态二进制翻译系统 QEMU

QEMU[17]是一种基于源动态二进制翻译器的快速机器模拟器。它可以支持 X86、PowerPC、ARM 和 Sparc 等多种平台的模拟，并且可以在 X86、PPC、ARM、Sparc、Alpha 和 MIPS 上运行。同时，QEMU 支持全系统级别和应用程序进程级别两种级别的模拟。全系统级的模拟可以使得一个完全的不加任何修改的操作系统在 QEMU 上运行。进程级别的模拟可以使得基于指令集架构（ISA）A 的应用程序运行在具有不同指令集架构（ISA）的平台 B 之上，而且不用作任何修改。

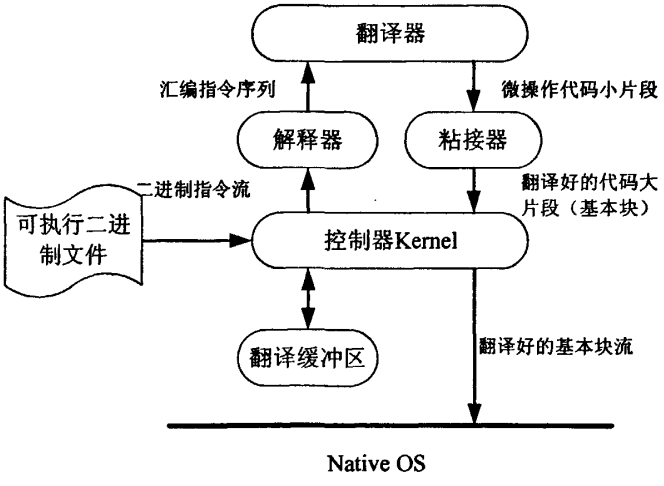

图 3.1 QEMU 基本结构和流程

如图 3.1 所示，QEMU 系统由控制器核心、解释器、翻译器和翻译代码缓存等几部分组成。在模拟器运行时，控制核心会维护一个软件的虚拟 CPU，它包括通用寄存器、段寄存器、标志寄存器、PC 等。被模拟的目标机器的所有资源均通过这个虚拟的 CPU 进行访问。QEMU 模拟流程分为以下几个步骤：首先控制器先对目标源二进制文件进行加载，类似程序的加载器，区分出程序的入口、符号表等执行信息。然后通过解释器解释这些二进制流，识别成汇编指令，翻译器将这些汇编指令翻译成等价的微操作序列。这里微操作被定义为模拟 CPU 里的一个

原子性的计算操作，不可分割。粘接器根据这些微操作序列将它们对应的宿主机相关的二进制代码片段（已经离线编译保存在 OBJ 文件中）拼接起来形成一个基本块对应的翻译好的代码片段，并缓存到翻译代码缓存区中，这些翻译好的代码和源二进制基本块的信息保存在 TB（Translated Block）中。最后，在目标机上执行这些翻译好的代码，从而实现源二进制代码的虚拟执行。

3.1.1 QEMU 的运行方式

QEMU 是一个采用可移植动态二进制翻译技术的通用 CPU 模拟器。它具有应用程序用户级和全系统级两种的模拟功能。

用户级二进制翻译系统提供了一个虚拟的操作系统。从一个用户程序进程的角度来看，这个操作系统提供了一个逻辑内存空间和用户级的指令与寄存器供这个进程使用。用户进程可以使用机器的 I/O 系统，但只能通过这个操作系统的系统调用来使用这些 I/O 接口。一个用户级的二进制翻译系统实际上就是一个运行的独立进程。它只支持应用进程运行，在进程运行时创建，同这个进程一起结束。而一个系统级的二进制翻译系统提供了一个完整的、持久的系统环境来支持操作系统和大量用户程序。它可以为其上的操作系统提供虚拟的硬件资源，比如网络、I/O 和图形化用户接口。所以这种方案主要是模拟进程可以看到的部分，例如系统调用的接口、源机器的指令集（Instruction Set）。因为在不同的操作系统里系统调用存在很大的差别，就可能造成在宿主机上无法找到对应的源机器操作系统的系统调用。所以用户机的二进制翻译系统必须正确而且高效的处理系统调用。一般来说有两种模拟系统调用的方法，一是根据源机器操作系统的系统调用格式对目标操作系统上的系统调用进行封装，例如 QEMU；二是直接在目标操作系统上实现源机器操作系统上系统调用。例如 Wine。

QEMU 的系统级的二进制翻译系统提供了一个完整的虚拟机器。它需要模拟一个完整的源机器硬件包括硬件的内存管理单元以及一些外部输入输出设备，例如网卡、显卡、声卡等。模拟一个完整的源机器硬件，系统级模拟必须区分特权指令和非特权指令。对于跨平台运行程序，非特权指令可以直接运行在宿主机的 CPU 上，否则需要通过二进制翻译先翻译成宿主机的代码，然后才能运行。而特权指令却需要进行特殊的处理，因为被模拟的操作系统往往都运行在 Ring3 级别，而用户 Ring3 级别不具有各特权指令的执行权限，只能通过宿主机的操作系统内核即 Ring0 才能执行。所以这些特权指令要用一些特定的代码来模拟陷阱，通常由宿主机上的操作系统提供的操作原语组成，此外宿主机的操作系统必须对这些特权指令做权限检测，确保符合系统的访问特权设置。

因为本文关注的是应用程序级的逆向执行，所以研究焦点主要集中在 QEMU

的用户级的模拟。

3.1.2 QEMU 的翻译过程

传统的解释翻译器的主循环是取指令，然后分析，根据指令的操作码进入一个 Case 结构中执行相应的指令处理函数，最后回到主循环继续执行这一过程。其详细结构如图 3.2 所示。

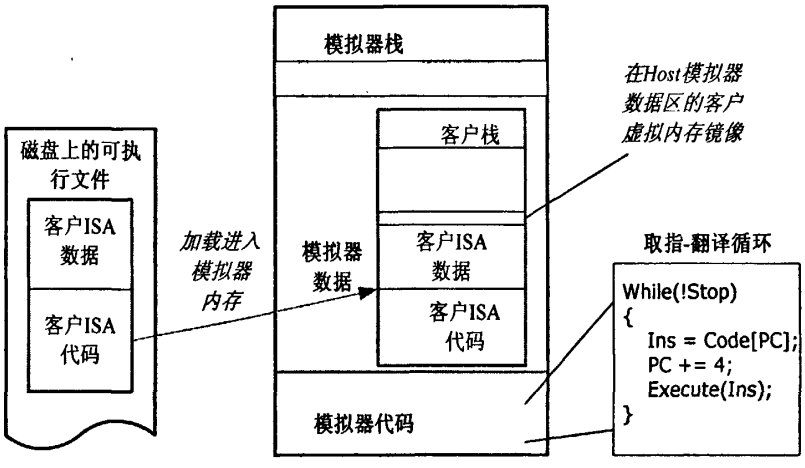

图 3.2 传统解释翻译器

在磁盘上的源二进制可执行程序文件被模拟器当作数据加载到模拟器的进程空间的数据段中，包括源程序的数据段和代码段。而模拟器拥有自己的模拟器代码段和栈空间，并且还需要模拟源机器平台的 CPU 和进程运行栈空间。模拟器的核心代码就是一个取值-翻译执行的大循环。

在 2.3.1 节中已经对解释器、静态二进制翻译器、动态二进制翻译器进行了概要的分析，他们最主要的区别在于一次翻译单位的选取，也就是说一次所翻译的代码的长度。如果每翻译一条指令就立即执行就是解释器；而如果翻译的单位扩大到整个被虚拟的程序，也就是说在翻译完程序的所有指令后再执行，则被称为静态二进制翻译器。而动态二进制翻译器的翻译单元是介于解释器和静态二进制翻译系统之间的。其区别于静态翻译器的地方在图 3.2 中的表现就是取值-翻译循环过程中，PC 并非是 $PC+=4$;而是加上这一次动态翻译的指令内容大小。

一般来说，动态翻译器都是以基本块（Basic Block，BB）为单位。一个基本块就是在这个代码块内除了第一条指令为入口，最后一条指令为出口外，不存在任何其他的入口或者出口，即常说的“单入单出”。“单入单出”有个特点就是只要进入这个执行块，就能够知道在哪里出去，可以被看作一个原子性的分段，基本块就是基于这个特点。只要确定了翻译的入口，就可以知道翻译的出口，这样

便于翻译，同样也便于完成执行，对于确定性的跳转出口，还可以将多个基本块串联起来以进行优化，加速程序的执行效率。选择基本块（BB）做翻译单位，相对于使用单条指令为单位的翻译，可以省去很多有关函数调用的操作，比如说堆栈的维护，包括参数和局部变量以及返回地址的处理。此外还可以充分利用目标机的优势，用尽可能少的目标机指令来实现源机器程序中多条指令的语义同时可以充分挖掘基本块内的指令并行性，给编译器提供了优化空间。

在翻译过程中 QEMU 为了能达比较好的可移植性，采用了逐条指令解释翻译的方法，先把基本块内的所有源机器指令分成若干个自己设计的微操作，然后把这些预先编译好的微操作无缝地拼接起来。这个过程不同于传统的解释器需要频繁的调用微操作函数。而是采用了 Direct Threaded Code 技术，省去了函数调用的开销和堆栈的维护开销。

同时为了做到无缝连接微操作（Mirco-Operation），QEMU 做了几种特殊的处理。首先，在编译微操作的时候，采用了-fomit-frame-point 编译选项。在 QEMU 被编译时不再把帧指针保存在固定的寄存器上，这样可以避免产生一些保存和设置和恢复帧指针的指令，同时还额外的提供了一个寄存器用于寄存器映射优化。例如，在 X86 的体系结构下，该编译选项可以省去保存恢复寄存器 ebp、esp 的步骤。其次，在设计微操作的函数实现时不使用参数和局部变量。对于有些源机器指令包含了一些常量，如相对跳转指令中的立即数等；还有另外一些参数必须在执行的时候才能确定，有可能还和不同的体系结构相关。在 QEMU 的实现中，所有的微操作函数使用全局变量来为这些常量预先占位，而在分解指令时计算出这些常量的值，并按照基本块内出现的顺序保存在固定的 buffer 上。在连接微操作时，根据 GCC 的重定位全局变量信息，直接把缓存区中的数据拷贝到原来全局变量的引用位置。在翻译产生的代码中以立即数的形式出现，利用这种方法来得到高效率的宿主机代码。最后，利用微操作函数的标识来表示微操作的起始位置，并在拷贝的时候去除函数末尾的返回指令 ret，从而得到微操作函数中真正有意义的实际代码。

总之 QEMU 以基本块为单位进行翻译，但是在翻译基本块的过程中又只是采用逐条翻译指令的方法。因为它是以基本块为单位来翻译的，所以它也只能精确到基本块。通过上面的做法 QEMU 可以避免函数的调用的开销，做到微操作的无缝连接。但是从微观上来说 QEMU 的翻译单元还是一条条单个汇编指令，因为源机器的二进制指令被分解成若干个对应的微操作，组成了微操作序列。

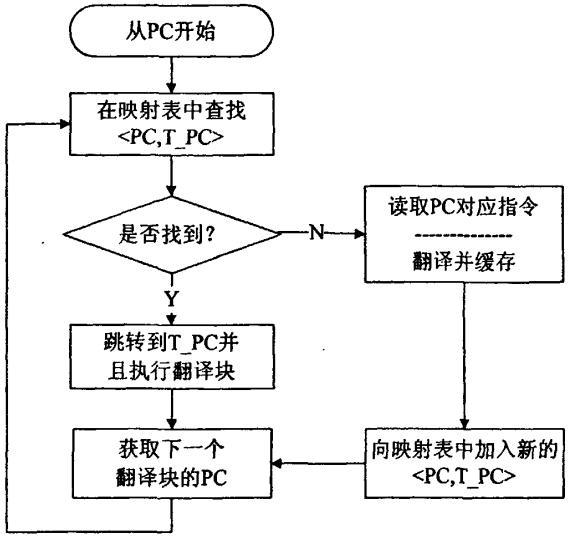

图 3.3 翻译-执行过程

在QEMU中将源二进制代码的基本块和翻译好的并存放在缓存区的对应的代码片段作为一个映射<PC,T_PC>保存下来，称为 TB（Translation Block），用来保存上层源二进制代码和下层翻译代码之间的基本信息及其两者之间的纽带映射关系。

图 3.3 给出了 QEMU 翻译-执行的流程示意图。QEMU 在对源二进制代码进行翻译缓存到翻译代码缓存区，每当运行完一个翻译基本单元之后（对 QEMU 来说，因为优化原因，翻译基本单元有可能是一个基本块或者多个连续的基本块，后面讲到），首先检测 TB 中保存的<PC，T_PC>映射对，如果在 TB 列表中没有保存，则表明此 PC 对应的代码并未翻译，所以必须启动翻译器翻译以 PC 为首的基本块的代码。在翻译完成之后，将对应的<PC,T_PC>信息存入 TB 结构列表中。当下一次碰到此 PC 对应的代码的时候，模拟引擎会利用 TB 映射表中查找 PC 对应的<PC,T_PC>映射对，然后在代码缓存器里查找需要执行的已翻译的，如果查找成功则直接执行，如果查找失败，则激活翻译器翻译所需的代码。

在一个基本块被翻译完之后并不是立即进行后续的优化操作，而是在执行完这个翻译好的基本块的代码之后，通过一定的策略对这些存储在翻译代码缓存区的代码进行优化。例如，如果前一个执行的基本块是直接跳转到刚刚执行的基本块的，则直接修改前一个执行的基本块的末尾跳转目标从跳转到模拟器查找<PC,T_PC>映射对修改为直接跳转到本基本块所在的缓存区中对应的地址。当下次前一个模块被执行完之后，程序的执行流程不是转移到模拟器，而是直接执行其跳转的下一个基本块。

在动态二进制翻译系统中，程序虚拟执行时间由两部分构成，一是对程序进行解释翻译以及优化的过程，二是对翻译后的代码进行执行的过程。但是对于程

序执行来说，第二部分对翻译代码的执行才是实际的工作部分。通过减少程序解释翻译的时间开销或者减少翻译后代码执行的时间开销都是能够提高程序执行效率的。例如，上面讲到的减少翻译代码执行向模拟器的跳转次数，增加翻译代码的执行连贯度，都是可以加快程序虚拟执行速度的优化手段。

3.1.3 QEMU 的运行流程

在启动 QEMU 模拟执行应用程序之后，首先加载源机器的程序文件与相应的库文件，然后初始化 CPU 的状态，然后进入动态二进制翻译的主流程，一边翻译一边执行。在遇到系统调用的时候则执行 QEMU 模拟的系统调用，通过调用到宿主机操作系统中对应的系统调用来完成源机器程序所期望的功能。当源二进制程序进程结束的时候，QEMU 模拟进程也同时退出执行。图 3.4 显示其执行流程。

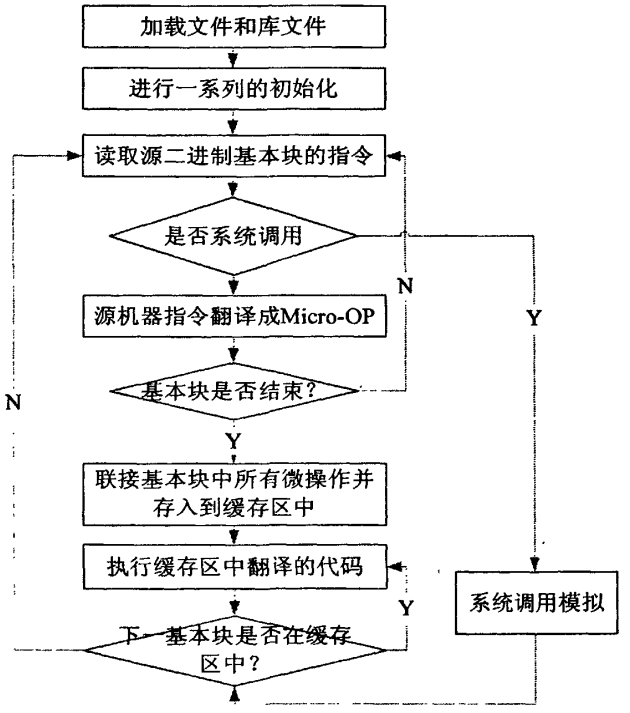

图 3.4 QEMU 执行流程

3.1.3.1 二进制程序的加载和初始化

由于整个系统是用户级的模拟系统，所以系统在运行时使用和模拟器的翻译

源程序具有相同的虚拟地址空间。本系统的处理方式是将系统本身先被强制加载到一个相对较高的地址处，而将翻译源程序按照其编译时设定的地址加载。加载过程由控制核心完成。具体的加载过程是分为两步的，一是打开文件检查其属性，二是将文件加载到程序执行进程空间中的指定位置。而整个加载过程的核心内容是识别翻译源文件的结构，从而获得加载的位置的相关信息。

在本文中使用的是 Unix 系统中的 ELF 文件格式，ELF 文件可以从两个角度去分解，一是从文件链接的角度，文件由一个一个的片（Section）组成，二是从程序执行的角度，由段（Segment）组成。这些区分都是有 ELF 格式的头（ELFHeader）来决定的。

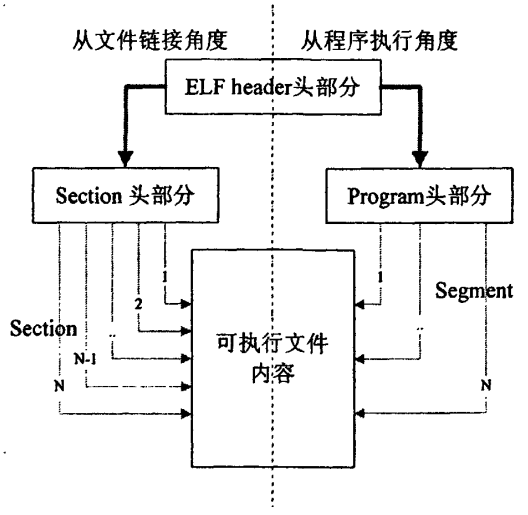

图 3.5 ELF 文件格式

如图 3.5 所示，在整个 ELF 文件格式中只有 ELFHeader 的结构和位置是固定的，其他各段都是可变的，并且都保存在 Section 的头部或者 Program 的 Segment 头部，这两个头部的信息记录在 ELF 的头部结构中。在 QEMU 加载程序时，控制器会产生一个数据结构 linux_binprm 用于保存整个 ELF 文件的基本参数，同时进行整个 ELF 文件的属性检查。在确定完文件属性和权限之后，控制核心就开始分析 ELF 的文件，确定程序的 Segment 表头的位置，并读取其内容，确定各个段的起始位置和段名，并进行相应的内存映射。

3.1.3.2 源二进制程序的翻译过程

QEMU 用 C 语言的函数设计一系列的微操作（Micro-OP），这些微操作在 QEMU 被编译的时候已经被编译成宿主机相关的二进制代码了。在运行源机器程序时，QEMU 先将每条源机器指令翻译成更微小的若干操作，然后通过直接拷贝

拼接的方式连成与宿主机相关的代码片段，并存入 QEMU 的缓存区。通过直接执行缓存区中的宿主机代码完成源二进制程序的虚拟执行。通常来说，QEMU 在执行翻译代码之间会先判断该基本块是否被翻译，如果翻译了，就直接执行缓存中相应的代码。

对于每条指令来说，首先解释器进行解释，负责将一条一条的指令从源二进制程序流中区分出来。同时解释器还需要做以下工作：

- ✓ 对指令进行长度判断，语义解释，并将其中与寄存器相关的指令转换为对于中间变量的操作指令，并将一些复杂的指令分解为几个简单的操作；
- ✓ 识别二进制程序中的跳转指令，函数返回等控制流指令，并计算出跳转的目标地址，同时将目标地址写入解释后的指令；
- ✓ 对系统调用进行统一处理，并将其传给控制器核心，进而提交给操作系统做出相应的处理；

继而，下面的工作由翻译器来处理。对于解释器产生的伪指令序列，翻译器将根据顺序找到一一对应的 C 语言程序段，这个程序段已经被编译成宿主机上的可执行代码。翻译器还得在这些代码中加入立即数和跳转的目的地址，从而转换成可以在宿主机上执行的代码。

最后由粘接器将这些代码小片段拼接起来，最终完成一个完整的基本块的二进制可执行程序。下面将给出一个例子，对此过程进行描述。

例如 x86 平台下指令：

```
stl %eax, 0xfffffffc4(%ebp)          ；将 eax 值存入 ebp-3b 这个内存地址中
```

首先，解释器会分析出这条指令，并将其分解成以下几条微操作（Micro-OP）：

```
MOVL_A0_EBP        ；A0 = EBP
ADDL_A0_IM(-3b)    ；A0 = A0 + (-3b)
MOVL_T0_EAX        ；T0 = EAX
STL_RAW_T0_A0      ；将 T0 值存入 A0 地址空间
```

翻译器会根据生成的这几个微操作找出其对应的 C 代码实现函数：

```
void op_movl_A0_EBP(void){
A0 = env->regs[R_EBP];
}
void op_addl_A0_im(void){
A0 = (uint32_t)(A0 + PARAM1); //PARAM1 是全局参数
}
```

```
void op_movl_T0_EAX(void){
T0 = env->regs[R_EAX];
}
void op_stl_T0_A0(void){
*(unsigned long*)A0 = T0;
__asm__ __volatile__("" : : : "memory");
}
```

在其中立即数（-3b）会被控制器保存为立即数在堆栈中，作为参数 PARAMn 使用。最后粘接器把这三个 c 函数对应的微操作码连接起来，存入 Cache 中，就完成了对指令 stl 的翻译过程。需要说明的是，在整个模拟操作中，T0，T1，T2 作为中间变量进行运算，在所有微操作中，不存在任何以机器寄存器为操作数据的微操作。

QEMU 保存所有已经翻译的基本块的信息，包括起始地址、长度以及其翻译完之后对应的缓存代码的地址信息，同时保存的还有本基本块和其他基本块之间的跳转信息等。所有这些信息保存在前面所述的结构 TB（Translation Block）中。QEMU 通过 hash 表来查找源代码中以 PC 为起始的基本块是否已经被翻译并缓存起来，并确定其翻译后代码的地址位置，便于直接运行。

3.1.3.3 系统调用模拟

系统调用的模拟是用户级虚拟机或模拟器需要重视的一个重要问题。如 3.1.1 节中讲到，有对现有宿主机系统调用进行包装和完全库实现仿真两种模式。QEMU 采用的是对现有系统调用进行封装，这样可以减少很大部分的工作量，牺牲了模拟器的兼容性。但是由于 Linux 系统中系统调用接口极少改变，所以模拟器与宿主机的兼容性受到的影响可以忽略不计。

QEMU 在源代码机器中遇到系统调用时，就结束当前的基本块，并将系统调用的下一条指令的地址作为后一个基本块的首地址；然后把控制转移到模拟器模拟的系统调用，通过模拟的系统调用来仿真源机器的系统调用服务；在系统调用完成之后，控制器获得控制权，然后转回 QEMU 的主循环，从系统调用的下一条指令开始继续翻译和执行源二进制程序。

对于应用程序中调用的库文件，一般来说，其实际都是通过系统调用来实现的，因为在应用程序加载初始阶段，已经将应用程序所需要的库文件加载进入了模拟器内存空间，并当作源二进制程序一并翻译执行的。所以，如果被调用的库函数里面没有使用系统调用则和普通代码一样翻译执行，如果库函数里面有系统

调用，则先被翻译执行，直到系统调用处利用宿主机系统调用封装模拟实现系统调用。

3.2 程序历史状态的记录

在第二章已经讲到我们将采用二进制翻译和二进制植入技术对程序植入用于记录程序执行状态的代码，使得程序在被执行的过程中这些被植入的记录状态的代码可以自动执行，从而将此时的程序状态记录到日志文件中。一般来说，程序的历史执行状态包括所有模拟的寄存器、内存以及系统调用和系统资源句柄等。在这一节，在讲述这些状态的记录的同时，重点是介绍不同情况下的两种内存记录模式。因为对二进制翻译进行不同的改进和优化，产生了两种针对不同情形的二进制植入模式，针对不同指令集架构的多平台模拟的 Binary-Translate 二进制植入模式和针对相同指令集架构平台的 Binary-Copy 模式的二进制植入模式。在介绍这两种植入模式及其区别之前，先探讨一下对于程序植入点的选取。

3.2.1 植入和检查点的策略

在 2.4.2 节中讲到二进制植入的几种方式，静态植入不能解决代码中含有库调用问题；而基于探针式（Probe-based）的动态二进制植入在 X86 等变长指令系统上存在覆写的控制指令长度大于即将被改变的指令的长度，从而导致植入不能成功。从另一个意义上说，基于探针式的动态二进制植入在植入粒度不能够更细到每一条指令。对于那些需要对某些超小特殊指令的植入（可能只有 1 或 2 个字节，但是控制流跳转指令 JMP 有 5 个字节，如果采用基于探针技术的植入技术，JMP 指令可能会覆写多条指令）就没办法做到这一点。所以基于动态翻译的二进制植入技术是一个相对比较好的选择。在本文后面，不作说明的翻译器或模拟器都是基于动态即时翻译技术的，而实际上我们所实现的逆向调试器核心部分就是基于动态二进制翻译器 QEMU 实现的。

3.2.1.1 代码植入策略

本文的逆向调试器采用的是基于程序执行状态的记录和重放技术逆向执行技术，所以需要采取办法对程序执行状态进行记录。通常，在模拟器中，有两种方式可以实现程序状态的记录，一是修改模拟器的翻译后的代码部分，使得程序在模拟器上执行的时候，能够通过模拟器自动记录状态日志，而被模拟程序不做任何额外的改动；二是对程序进行标志植入，使得程序在执行过程中通过被植入代

码的运行而实现自动记录状态日志。而在本文中基于动态二进制翻译系统 QEMU 的逆向执行技术是将这两种状态记录的方法结合起来去记录程序执行状态中的不同部分，例如修改模拟器代码以记录模拟 CPU 的寄存器值，在翻译代码块中植入内存跟踪代码等。

因为一般基于动态翻译技术的模拟器都是采用一个基本块（Basic Block）或多个连续执行的基本块作为基本的翻译-执行单位，所以在开始执行每个基本块之前，我们可以将此时的寄存器和异常中断标志记录到日志中。但是仅仅这样是不够的，因为程序执行栈、堆以及打开的文件句柄、套接字、管道等都是需要记录的。因为程序执行栈、堆的操作都是内存的操作，简化为保存(Store)和读取(Read)两项动作，而对于那些打开的文件句柄，只能在系统调用的封装内对文件句柄的信息进行保存。

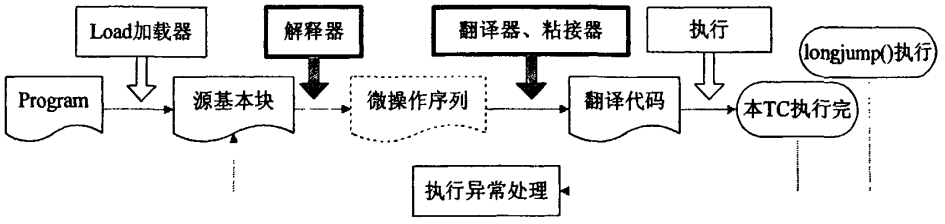

图 3.6 植入点的选取

如图 3.6 所示，因为模拟器的虚拟执行的过程就是解释器翻译源二进制代码流为微操作（Micro-OP）序列的过程，翻译器将这些微操作实现的对应的 C 代码拷贝出来，通过粘接器对其进行拼接成为可以在宿主机上执行的二进制代码，并直接在宿主机上执行这些翻译好之后的代码，从而模拟源二进制程序的执行过程的。由此可见，我们只能选择在解释器、翻译器以及执行翻译基本块这几处做修改使得程序的历史执行状态能够保存下来。

3.2.1.2 检查点的内容

在本节的前面讲到，不可能对每条指令都做跟踪，所以本文选择了做检查点（Checkpoint）。所谓的检查点对于软件来说就是跟踪一定的间隔内程序的所有异动，并把结果写入相应的 Undo 日志和 Redo 日志。一般来说比较常用的检查点策略是定时的间歇性的对所有状态甚至时整个程序的所有有关机器状态都做一个“剖面”快照（snapshot）[39]。这种做法对于非常大的长期运行的程序来说是一个比较有效的一种方法，例如对于操作系统的 Record/Replay。但是对于应用程序的用户级模拟来说，这样做是不够的，因为很多程序在一瞬间就能够执行完毕，对程序状态的长时间间隔设置检查点就没有意义。所以我们并不选择以时间为检查点的间隔的度量，而是选择以每个翻译-执行的单位作为检查点的间隔度量。这

样做有两个好处，一是，对于执行时间比较短的应用程序来说，这样做可以有效的保证有足够的检查点来记录程序执行状态；二是，因为每个翻译-执行单元的执行流程是“单入单出的”，也就是说，执行了这个单元的第一条指令，必然会执行这个单元内的所有指令而不会被打断。这样，我们就可以通过这个检查点记录的历史状态，跟踪到这个翻译-执行单元的每一条指令。

同时，对于内存的变化的跟踪，我们不会采用对所有内存页进行标记记录的方式，因为如果一旦某个翻译-执行单元修改了某内存页的一个字节，那么我们就必须将这个内存页的所有数据都保存下来，这样是不合适的。我们选择了检查点的间隔单位是一个翻译-执行单位，而每个翻译-执行单位所包含的代码都是非常少的，一般都是几条指令，甚至有的只有一条指令，那么如此所分的翻译-执行基本单位的数量是非常多的，而每个翻译-执行单位中修改内存的数据是非常少的，这样就导致大量的重复数据的保存，造成大量存储空间和时间的浪费。所以我们选取增量式的日志记录方式，非常精确地记录内存变量的变化。也就是说只记录变化的内存变量单元，或以 4 字节为单位，或以 2 字节、1 字节为单位记录其内存内容的变化。这样就需要在程序中跟踪内存操作的指令，对读或写内存的指令进行植入跟踪，记录下历史的内存状态值。

同时因为在一个基本块内，同一个内存单元有可能被改变多次，如果对于每一次改变都要记录下这些地址的历史值，那么势必得不偿失。所以，同一个内存单元在同一个翻译-执行单元内被多次改变，则只记录第一次被改变前的内存地址、内容和类型。因为是在一个“单入单出”的翻译-执行单元里面，只要内存在进入这个单元执行代码之前，能够恢复出入口处的内存镜像，那么后面的其他内存变化都是以此为基础的，都是可以通过代码执行重现的。

所以本文结合了通过模拟器在每次执行基本块的翻译代码之前记录程序的寄存器数据和在翻译代码中植入跟踪内存数据的变化两种方式，以每个基本块将要被执行的位置为检查点（Checkpoint），将两检查点之间的内存变化记录在前一个检查点内，作为前一个检查点的内容。在本文中解决的问题暂时只能是单核单进程单线程的程序逆向研究，对于多线程、多核等并发并行程序的逆向执行的研究将是今后的进一步研究重点方向。

本文在每个翻译-执行单元的代码被执行前，生成一个检查点，并对其进行编号，同时将此时模拟器虚拟的 CPU 中用于全组寄存器恢复的结构变量保存起来。检查点的编号将是按照执行程序的模块的顺序依次递增的。保存的检查点信息结构按照顺序保存在顺序数组内。如图 3.7 所示，在顺序数组内保存确定的状态的信息，例如寄存器当时的 Snapshot，同时因为有可能包含系统调用以及信号的处理，所以也将这些信息保存在数组中。因为每个翻译-执行单元内修改内存的数量不确定，所以，所有被修改的内存单元按照链表的形式链在数组上，仅仅在数组

结构中保存被修改的内存的单元数。

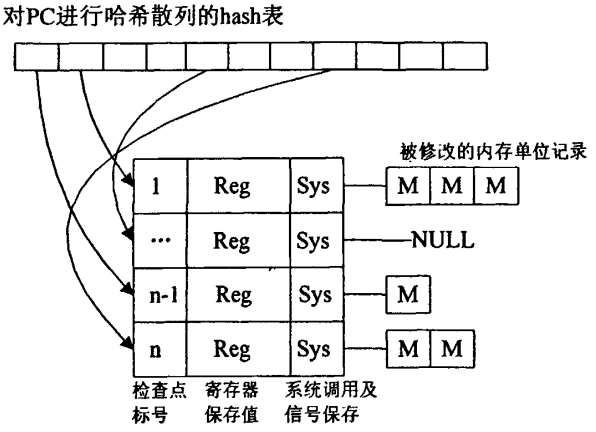

图 3.7 检查点数据保存结构图

同时，所有的检查点将被加入到以源二进制指令的 PC 为哈希的哈希查找表中，便于在获得逆向命令的时候，可以快速通过 hash 表查找到将要恢复的检查点。

3.2.2 寄存器状态的记录

寄存器是整个程序的执行过程中一个极其重要的部分，寄存器是中央处理器中有限存贮容量的高速存贮部件，它们可用来暂存指令、数据和地址。以下是 X86 系统的基本的 CPU 模拟结构。

```
typedef struct CPUXState{
    target_ulong regs[CPU_NB_REGS];
    target_ulong eip;
    target_ulong eflags;
    SegmentCache segs[6];
    SegmentCache ldt;
    unsigned int fpstt;
    unsigned int fpus;
    unsigned int fpuc;
    uint8_t fptags[8];
    union {
        CPU86_LDouble d;
        MMXReg mmx;
        } fpregs[8];
    uint32_t mxcsr;
    XMMReg xmm_regs[CPU_NB_REGS];
}savedReg[MAX];
```

本文在上述寄存器保存的数据存储结构的基础上设计了寄存器保存算法Record_Reg()函数来实现保存每个 TB 执行之前的寄存器状态。对于寄存器保存的详细算法如下。

```
Record_Reg(tb_idx, env)
BEGIN
    for i = 0 to CPU_NB_REGS -1
        savedReg[tb_idx].regs[i] = env->regs[i]
    endfor
    savedReg[tb_idx].eip = env->eip
    savedReg[tb_idx].eflags = end->eip;
    for i = 1 to CPU_NB_REGS
        Copy_seg(&saveReg[tb_idx].segs[i],&env->segs[i])
    endfor
    Copy_seg(&savedReg[tb_idx].ldt, &env->ldt);
    savedReg[tb_idx].fpstt = env->fpstt;
    savedReg[tb_idx].fpus = env->fpus;
    savedReg[tb_idx].fpuc = env->fpuc;
    for I = 1 to 8
        savedReg[tb_idx].fptags[i] = env->fptags[i];
    endfor
    for I = 1 to 8
        SavedReg[tb_idx].fpregs[i] = env->fpregs[i];
    enffor
    savedReg[tb_idx].mxcsr = env->mxcsr;
    for I = to CPU_NB_REGS
        Copy_xmmreg(&savedReg[tb_idx].xmm_regs[i],
         &env->regs[i])
    endfor
END
```

3.2.3 内存状态的记录

在程序执行过程中，内存是程序执行的极其重要的核心部分，用来缓存整个计算的中间结果，它通过与中央处理器（CPU）进行结果的交换，完成整个计算过程。可以说，没有内部存储，计算机的快速计算是不可能的。为了能够在程序逆向的过程中，完整的逆向整个计算机的计算过程，所以内存的状态的保存是必不可少的。前面在 3.3.1 节中讲到，我们选取翻译-执行单位作为记录的的检查点间隔，同时，我们只非常详细地记录每个被改变的内存单元，所以需要跟踪每一个内存修改/读取指令，在修改内存单元之前，将原始的内存保存进入检查点结构中。

在此基础上，根据代码翻译环境和方式的不同，我们做出了两种不同的翻译植入模式，Binary-Translate 模式和 Binary-Copy 模式的植入跟踪。前者应用在源应用程序和目标运行平台具有不同的指令集架构的情况下，具有跨平台性质；后者只能应用在源应用程序和目标运行平台具有相同的指令集架构的情况下，提供快速的接近本地执行速度的逆向执行功能。这两种模式都是跟踪内存操作指令，进行相应的代码动态植入，使得在翻译好的代码被执行的时候，相应的内存状态就会自动被记录下来。

3.2.3.1 Binary-Translate 模式

在 3.1 节中讲到 QEMU 的系统的运行是动态翻译的，先将源二进制程序的每一条指令解释为更小的微操作，并组成序列，然后翻译器根据这些微操作序列，将这些微操作对应的已经编译好的 C 实现代码拼接起来组成在宿主机上可执行的二进制代码。为了跟踪内存修改指令，我们有两种代码植入方式以达到这个目标：一是修改解释器，在内存操作指令的解释过程中，当源二进制程序翻译成微操作序列时，增加一个保存即将修改内存的微操作，也就是说，在最后得到的微操作序列中插入一个保存内存的微操作。同时，我们必须自己实现这个保存内存的微操作的 C 代码实现。这是一个比较直观的想法，但是因为一条指令所翻译的几个微操作之间具有一定的耦合性，存在指令间的“全局性”（就是存在变量在这一条指令所翻译的几个微操作间都是可见的），如果在中间加入一个“突然的”微操作，可能会破坏这条指令的翻译的正确性。二是修改已经存在的微操作的 C 代码实现，这种方式只要保证在不修改微操作实现的相应的变量即可。经过对 QEMU 中解释和翻译部分代码的分析得知，所有内存操作读、写有关的指令解释的微操作中都会包含以下读/写两类微操作，（1）写内存：4 字节内存写 OP_STL_XX_YY，2 字节内存写 OP_STW_XX_YY，字节内存写 OP_STB_XX_ YY；（2）读内存：4 字节内存读 OP_LDL_XX_YY，2 字节内存读 OP_LDW_XX_YY，字节内存读 OP_LDB_XX_YY。基于这一点，本文中提出的逆向执行技术采用的是修改微操作 C 实现代码的方式。

以下将给出在 3.3.2.1 节中给出的指令翻译示例：

```
stl %eax, 0xfffffffc4(%ebp)        ; 将 eax 值存入 ebp-3b 这个内存地址中
```

它将被翻译为指令：

```
1> MOVL_A0_EBP        2> ADDL_A0_IM(-3b)
3> MOVL_T0_EAX        4> STL_RAW_T0_A0
```

四个微操作，其中微操作(4)*STL_RAW_T0_A0*是保存内存值的实际操作，它未修改前的C代码实现如下：

```
1:  void op_stl_T0_A0(void)
2:  {
3:      *(unsigned long*)A0 = T0;
4:      __asm__ __volatile__("" : : : "memory");
5:  }
```

其实际内容就是将T0值存入内存地址A0处，为了跟踪内存A0处的修改，需要将此处的内存原值保存下来。被植入后的伪码如下：

```
1:  void op_stl_raw_T0_A0(void)
2:  {
3:      Store the type of Memory(A0);
4:      Store the address of Memory(A0);
5:      Store the content of Memory(A0);
6:      *((unsigned *)A0)=T0;
7:      _asm_volatile(“”:::”memory”);
8:  }
```

通过在内存A0被修改之前，保存A0地址原来的类型、地址和内容，能在需要恢复的时候读取A0地址的历史内容<A0，A0内容，类型>。这样，当微操作OP_STL_RAW_T0_A0的C实现代码被拼接成可执行二进制代码片段时，保存内存的操作就被加入到了代码中，随着代码的执行，内存变化就自动记录下来了。同时，因为是通过修改微操作实现代码来植入微操作的，而且这些微操作已经离线编译成了二进制可执行码，这样，虽然植入的灵活性降低了，但是在满足植入要求的前提下植入的效率就提高了很多。

Binary-Translate 模式是一种跨平台的翻译模式，具有很高的灵活性，但是牺牲了代码的执行效率，因为产生更多的冗余操作，代码膨胀了好几倍，所以产生了代码执行时间上的很多额外开销。

3.2.3.2 Binary-Copy 模式

Binary-Translate 模式提供了一种不同指令集平台间的程序执行状态的记录方式，当然具有相同指令集架构的应用程序和宿主机平台在这种模式下也可以工作。但是，因为它们具有相同的指令集架构，所以应用程序代码可以直接在宿主机上

执行，如果还利用原来解释成微指令，然后再拼接代码这种方式，必然是不可取的，因为这些二进制代码是经过编译器优化之后的最精简的可以在本地 CPU 上直接执行的，对于那些可以直接拷贝过来在本地 CPU 上直接执行的指令，可以直接整块的拷贝，省却了翻译器的拼接和由代码的膨胀带来的执行时间的额外开销。

当然，并不是每一条指令都可以直接拷贝过来直接在本地 CPU 上执行，因为有些指令，诸如调用指令、跳转指令、Push/Pop 指令等，涉及到某些模拟器模拟的运算构件，则需要回滚回去重新做解释-翻译步骤。在 Binary-Copy 模式中，因为代码是直接在本地 CPU 上执行的，所以需要将虚拟的 CPU 结构中的数据转到本地真实的 CPU 上，在执行完通过拷贝模式获得的基本块代码之后，再将本地 CPU 的相关数据拷贝到虚拟的 CPU 数据结构中缓存下来，等待下一个基本块的代码的执行。

Binary-Copy 模式能够为相同的指令集架构程序的运行提供接近本地执行的速度。但是因为跳过了解释-翻译这个步骤，那么在前一节中的 Binary-Translate 模式不再适用。因此本文根据不同情况，设计实现了针对相同指令集架构的平台的更为优化的 Binary-Copy 模式，直接将内存记录的代码以二进制的形式植入到执行缓存代码区。

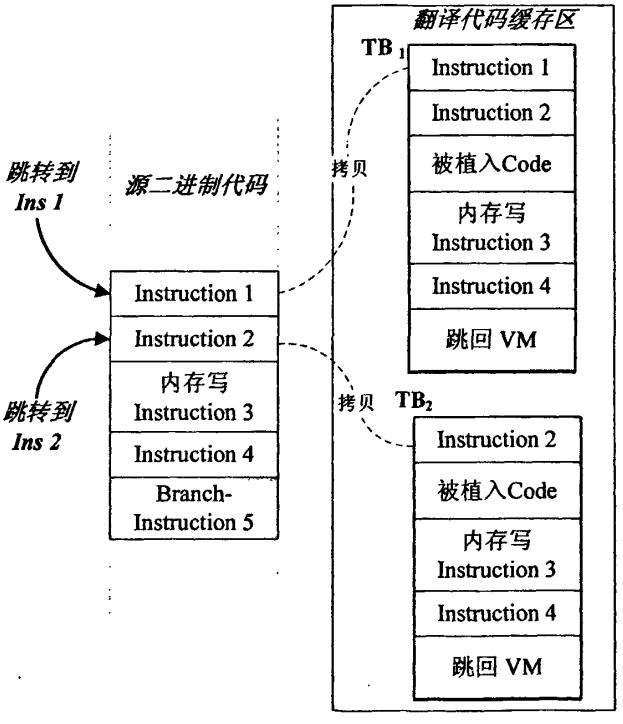

图 3.8 Binary-Copy 模式的代码植入

同时，在构造翻译代码的时候，在同一段代码内以不同 PC 地址为首指令的基本块将会产生不同的翻译代码块。如图 3.8 所示，有某个跳转到 Ins_1，那么以 Ins_1

为首将会产生一个基本块，并为这个基本块产生一个翻译代码块 TB_1，而另外有一个指令跳转 Ins_2，Ins_2 是以 Ins_1 为首的基本块内部的指令，但是为了简单处理，还需要产生一个新的以 Ins_2 为首指令的基本块 TB_2。

为了记录即将被修改的内存，在内存写指令前需要植入代码来记录原始的内存，当然，因为翻译的代码是直接在本地 CPU 上执行的，被植入的代码肯定会影响本地 CPU 的寄存器值，从而导致这些植入的代码干扰了原来的代码的执行甚至破坏了原来的控制流。为了避免这些情况的发生，在植入前，需要对当前寄存器的值做一个备份，在完成整个操作之后再利用备份恢复原始寄存器的值。那么被植入的代码就需要完成包括前期的准备工作和后期的恢复工作，所以被植入的代码将按顺序完成的以下工作：

1) 保存当前 CPU 的所有寄存器相关值到 CPU 备份中；
2) 保存即将被下一内存修改指令修改的内存状态到日志文件中；
3) 恢复保存在 CPU 备份中的寄存器值；
4) 跳转到原始的下一条指令处继续执行；

这样，程序的执行流程就不会被植入的代码所干扰了。

3.2.3.3 基本块内存保存策略

当然对于每个基本块来说，在能够把程序状态完全恢复到某个 TB 起点的前提下，保存越少的内存的副本，越能提高时间和空间上的效率。为了达到这个目的，首先引入第一次写引用指令(First Writing Reference Instruction)的概念[34]。所谓第一次写引用指令是指在一个基本块中，所有向同一内存地址的写指令中的第一条写指令。我们只需将第一条写引用指令中将要修改的内存原值保存下来即可。

```
1. Read  A;    //第一次引用——读
2. Write B;    //第一次引用——写
3. Write A;    //第一次引用——写
4. Read  B;    //第一次引用——读
5. Write A;    //第二次引用——写
6. Read  A;    //
```

图 3.9 一个基本块内的内存访问记录

如图 3.9 所示，在一个基本块内，A 被写访问 2 次，分别在第 3、5 行，那么只需要在第 3 行的写 A 内存前对内存 A 进行一次保存即可，对于第 5 行的写 A 内存不需要做任何操作。因为在恢复的时候，只需要恢复第 3 行以前的内存 A 的值，而第 3 行到第 5 行间的值是由第 3 行的写内存指令所决定的，根本不需要再

划分出空间用来记录第 3 行到第 5 行间内存 A 的值。为了便于判断某内存是否是本基本块的第一次写引用，设计了一个查找哈希表。在每次需要保存内存地址前，先进行哈希查找，判断是否在哈希表中已经存在这个内存值，存在则不用植入代码进行保存此内存值。否则，将这个内存地址 hash 映射到哈希表中保存下来。每当执行完一个基本块的翻译代码，就立即清空表示内存存储的哈希表。因为对于每个执行单元来说，内存修改的地方非常少，所以可以适当的缩小哈希表的容量，保证较少的冲突。

3.2.4 系统调用的记录

本文中所设计的逆向执行技术是基于用户级的应用程序模拟执行，所以系统调用和信号实现都是由宿主机操作系统提供的，模拟器仅仅是对这些已有的系统调用进行了封装。系统调用是操作系统提供的功能，是可以信赖的，无需追踪。在调试中，只要保证每次执行有完全一致的结果，即对内存和寄存器的改变是相同的。在翻译代码中出现系统调用，则跳出翻译-执行大循环进行系统调用中断处理，根据相应的系统调用号和参数构建宿主机的系统调用入口，利用宿主机的系统调用实现源二进制代码的系统调用效果。为了模拟系统调用的效果，需要通过在模拟系统调用的封装函数内将这些系统调用的系统调用号、调用参数和调用后的返回参数记录进入日志文件，在重建历史状态的时候，根据这些已经保存下来的参数，进行系统调用重现。但是对于某些复杂的系统调用，因为涉及到很多复杂的操作系统资源，并不是所有的系统调用都可以通过上面的方式重现。例如，文本写操作，可能需要对整个文本做一个日志的增量式保存，通过模拟给出系统调用后的结果，而不是实际的去执行已经执行过的系统调用。

操作系统发送给应用程序的进程的中断信号都是通过操作系统的系统调用来实现的，所以对于这一类信号，可以通过记录系统调用来处理；但是另外在应用程序内通过 signal 类的信号函数激发信号非结构化的调用流程，在模拟器中也是通过跳出翻译-执行大循环，通过操作系统模拟操作实现的，现在只是有一个初步的构想，通过保存信号处理表和模拟发送信号实现程序中信号的发生和处理。正因为同某些系统调用一样，可能向进程或内核的地址空间中写入新值，但是简单的使用相同的参数调用同一个系统调用并不能在逆向调试中重现上次该系统调用的行为。这些都有待于进一步的研究。

3.3 程序历史状态的重建

通过 3.2 节中对逆向执行技术中记录程序历史执行状态上的设计的详细描述，

为了能够完全的逆向执行，还需要读取这些已经记录下来的信息，根据这些信息重建历史执行点的程序状态，使得程序能够返回到某历史执行点。

3.3.1 逆向执行策略

根据逆向的时机和执行的状况的不同，调试过程中程序的执行有三种情形：

1. 用户希望程序停止的断点在当前执行状态之后，且当前点和目标断点之间的基本块没有被执行过，则顺序执行，计算并保存路径上所有基本块的初始状态信息。
2. 用户希望程序停止的断点在当前执行状态之前，则利用已保存的检查点恢复信息，把CPU状态和必要的内存状态恢复到断点所在基本块的开始，顺序执行到断点处。
3. 用户希望程序停止的断点在当前执行状态之后，但是当前点和目标断点之间的基本块已经执行过，则顺序执行，且每执行到一个新的基本块，需要先恢复本基本块所在的内存状态，执行直至到达目标断点。

通过读取已经记录下来的程序状态来重建历史执行点的所有程序状态。当然在调试的过程中，程序员需要通过外部输入命令使程序逆向到那些历史执行点上。所以对于逆向执行的虚拟机部分集成了调试器的服务端，用来解析调试器远程发送的命令字符串，并根据解析出的命令和数据进行相应的操作。逆向调试器可以在传统调试器的基础上，对其远程调试协议进行修改，增加逆向命令的数据包处理，同时在虚拟机中增加逆向命令的数据包解析及其动作处理。

在服务端接收到的由远程调试端发送的命令中，需要包含将要逆向过去的源指令的PC，因为需要通过PC来确认需要返回去的位置。在解析了返回的指令的PC后，需要根据此时PC的值，查找到将要逆回去的检查点的编号，然后恢复在此检查点时的程序状态，包括寄存器状态值和内存的值，同时，在逆向的过程中，需要对系统调用进行模拟回滚。因为对于一个CPU来说，只有一套寄存器，所以，只需要恢复对应编号的检查点的寄存器保存记录就可以了，而对于内存来说，为了节省内存存储空间可以采用增量式的日志记录方式。那么对于内存的恢复需要从最后一个检查点依次恢复到相应编号的检查点内存保存记录。下面将详细的解释内存状态的恢复流程。

3.3.2 进程空间内存的重建

为了恢复某一时刻的内存镜像，需要利用保存下来的内存记录去覆写现在的内存，不同于常用的内存页监控的保存恢复模式，本文采用了精确的内存细粒度单元值保存方式，为了节省保存空间，采用的是增量式的记录保存方式，这就决定了某时刻的内存空间状态并不是由一个检查点来决定的，而是由当前程序执行的状态和所有从即将返回去的检查点记录到最后一个检查点记录所决定的。因此，为了恢复某检查点处的内存状态，需要从最后执行状态一直逆着程序执行时保存的检查点的顺序覆写回去。如图 3.10 所示，在运行程序的过程中，执行内存保存，对于基本块 TB_i 来说，需要记录被修改的 A，B 和 C 三处不同的内存，而对于基本块 TB_{i+1}，D 内存被写两次，但是根据在 3.2.3 节中的基本块内存保存策略，则只需要保存内存 D 第一次写引用的原内存值，所以对于 TB_{i+1} 来说，只需要按顺序保存内存 D 和 E 即可，TB_n 同理，只需保存内存 E 即可。当然，对于内存的恢复来说，就需要从当前程序执行的状态作为基准点，逆着内存检查点保存的顺序从 $TB_n, TB_{n-1} ... TB_{i+1}, TB_i$（i 为需要返回的检查点的编号），将每个检查点内的 Undo 内存记录都覆写回原内存，例如，图 3.10 中的内存覆写顺序即为，$TB_n : E, ..., TB_{i+1} : DE, TB_i : ABC$，同时，在内存恢复的过程中，需要记录其 Redo 日志，也就是说便于程序可以返回当前的正常执行基准点。同 Undo 日志记录内存单元被修改前的值相反，Redo 日志记录的是程序在执行内存修改之后内存单元的值。

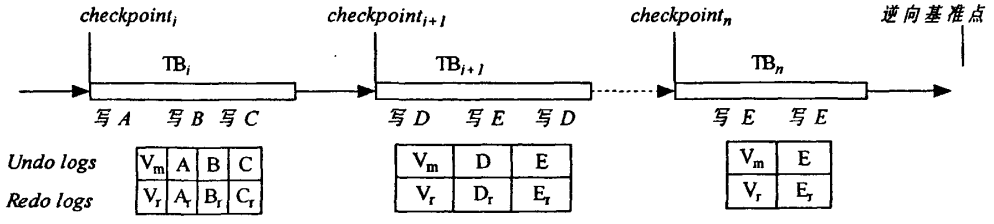

图 3.10 内存检查点的恢复

3.4 本章小结

本章通过对动态二进制翻译机制的分析，并基于动态二进制翻译器 QEMU 探讨了逆向执行技术的方案。通过记录和重建历史执行状态来实现程序的逆向执行，不同于以往的根据时间间隔做检查点记录程序执行状态的方法，本文以基本的翻译-执行单位基本块为检查点的记录间隔，在执行基本块的翻译代码之前，保存当前模拟 CPU 的寄存器状态信息，同时跟踪和植入源指令中内存写的指令，在执行

这些指令之前，保存即将修改的内存状态信息到相应的检查点中。对于二进制代码的植入，根据应用程序和宿主机指令集架构平台的异同，分别设计了不同的植入方式：Binary-Translate 模式和 Binary-Copy 模式。对于历史执行状态的重现，主要是读取已经保存的程序执行状态日志文件，然后逆着保存这些检查点的顺序恢复历史执行点的程序状态。本章讨论的逆向执行仅适用于单核单线程上确定性的应用程序的逆向执行，对于并发或并行的应用程序还应该考虑线程间信号同步互斥以及检查点的保存顺序等一系列非确定性的问题，这将是今后的一个很重要的研究方向。

第四章 逆向调试器 PORD 的设计与实现

逆向调试器 PORD 在传统调试器的基础上增加了逆向执行的功能，可以根据程序员的需要逆向执行到任意的历史执行点。本章主要是在第三章的基于 QEMU 的逆向执行技术的基础上介绍逆向调试器 PORD 的设计和实现。PORD 主要由两部分模块构成，一是逆向调试器的虚拟执行环境模块，二是为用户提供调试接口的 R-GDB（Reversible GDB）模块部分，这两部分之间通过修改的 GDB 远程调试协议通信。

4.1 逆向调试器架构

在第一章，本文已经简要介绍了 PORD 构成的两个模块，虚拟执行环境模块，我们称之为虚拟机，和用户调试会话接口界面，称之为 R-GDB。本文的思想是，被调试的程序在虚拟机上模拟执行，在执行过程中，记录程序执行状态，程序员通过调试器 R-GDB 控制虚拟机的执行，通知虚拟机逆向的目标地址，以及一系列传统的调试器功能。为了实现这一目标，本文设计了基于动态二进制翻译器 QEMU 的逆向调试器 PORD。其框架如图 4.1 所示。

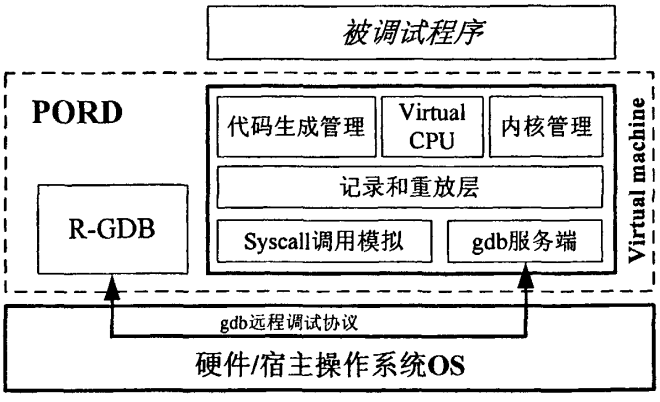

图 4.1 逆向调试器 PORD 框架

逆向调试器 PORD 在宿主机的操作系统上以应用程序进程的方式运行，而被调试程序被 PORD 以数据的形式加载，翻译并模拟执行。所以在虚拟机模块中包含系统调用的模拟层，利用宿主机操作系统已经提供的系统调用来模拟上层被调试程序中的系统调用。

在虚拟机模块中，包含了代码生成管理、虚拟 CPU、内核管理、程序状态记录和重放层，及系统调用模拟等模块。

- 记录和重放层：为了便于方便的控制程序动态植入和记录重放，在 PORD

的虚拟机模块中包含了一个记录和重放层。利用这一记录和重放层，可以根据实际情况，决定是否对程序进行植入和记录，同时在接到逆向命令后，执行主要的寄存器和内存的恢复操作；

- 代码生成管理模块：根据不同的环境和植入选项选择植入模式，Binary-Translate 模式和 Binary-Copy 模式，对程序的内存操作指令进行跟踪植入；
- 虚拟 CPU：模拟源机器的核心 CPU，以及一系列重要的执行时信息；
- 内核管理：主要是负责虚拟系统中的初始化，上层内存映射管理，翻译代码的缓存区管理等；
- 系统调用模拟层：对宿主机的系统调用进行模拟，在正常执行的时候对系统调用的调用号和参数进行记录和分析。

同时，在逆向调试器的用户调试会话界面模块中，主要是对 GNU 的传统调试工具 GDB 做一定的修改，使其能够具备发送逆向调试命令的功能。主要的修改部分有两处，一是在虚拟机内部集成的 GDB 服务端，二是通过 GDB 解析命令并发送远程命令数据包部分。因为在第三章中讲到基于 QEMU 的逆向执行技术，其中涉及到一些执行和重放的模块设计部分。所以本章的重点将放在虚拟机部分的其它模块和 R-GDB 部分的设计和实现。

4.2 虚拟执行模块

虚拟执行模块是程序执行的核心部分，被调试程序被完全的控制在虚拟机中执行，利用虚拟机可以任意修改程序执行的状态，所以利用对虚拟机的控制来恢复历史状态并控制程序的执行流程，以达到逆向执行程序的目的。整个虚拟机部分是在动态二进制翻译器 QEMU 的基础上进行修改的，

4.2.1 代码生成管理模块

为了更好的使被调试程序灵活的被执行，在代码生成管理模块中设置了一个标志变量 bInject，来表明是否对被调试程序进行植入。如果 bInject 为 false，则不对程序进行植入，程序将按照 QEMU 原流程进行执行，且不具备可逆向调试的功能。如果 bInject 为 true，则还需要根据当前 PORD 执行的命令参数来做是否对程序进行植入的判断，默认情况下是不进行代码植入，根据不同的情况调用不同的解释器和翻译器函数。表 4.1 和表 4.2 将给出这些函数及其解释。

表 4.1 Binary-Translate 模式下解释器和翻译器函数列表

Binary-Translate 模式的函数	描述

static int disas_insn(DisasContext *s)	不带植入的源二进制指令解释器，产生微操作序列
static int disas_insn_i(DisasContext *s)	源二进制指令翻译器，产生带植入的微操作序列
int dyngen_code(uint8_t *gen_code_buf, uint16_t *label_offsets,....)	根据微操作序列和参数列表生成宿主机可执行的二进制代码

表 4.2 Binary-Copy 模式下解释器和翻译器函数列表

Binary-Copy 模式的函数	描述
static int disas_insn_copy(DisasContext *s)	不带植入的指令解释器，直接产生可执行的二进制代码
static int disas_insn_copy_i (DisasContext *s)	带植入的源二进制指令解释器，直接产生宿主机可执行的二进制代码，其中植入了保存状态代码

另外还有一个重要的函数:

```
int cpu_gen_code(CPUState *env, TranslationBlock *tb,int max_code_size, int *gen_code_size_ptr);
```

是用来翻译源二进制程序并最终产生源机器上一个基本块对应的翻译代码片段的。也就是说，所有的代码均是通过 cpu_gen_code 来产生，并缓存到翻译代码缓存区中的，参数 env 中的 PC 指定下一条即将执行的指令位置。如图 4.2 所示 cpu_gen_code 根据不同的情形调用表 4.1 和 4.2 中的解释器和翻译器函数生成宿主机相关的二进制可执行代码。

```
//代码生成模块伪码：
Int cpu_gen_code() {
    Switch（代码生成模式）{
    Case  Binary-Translate 模式且不植入：
        For(;;) { 不带植入的 disas_insn(); }
        For(;;) { dyngen_code();//根据生成的微指令序列产生宿主机代码 }
        Break;
    Case  Binary-Translate 模式且植入：
        For(;;) { 带植入的 disas_insn_i(); }
        For(;;) { dyngen_code();//根据生成的微指令序列产生宿主机代码 }
        Break;
    Case  Binary-Copy 模式且不植入：
        For(;;) { 直接拷贝源二进制代码 disas_insn_copy();}
        Break;
    Case  Binary-Copy 模式且植入：
        For(;;) { 直接拷贝源二进制代码 disas_insn_copy_i();}
        Break;
    Default：Error;
    }
}
```

图 4.2 代码生成伪代码示意

4.2.2 记录和重放层

整个 QEMU 系统的翻译-执行过程的核心控制函数是：int cpu_exec(CPUState *env1)；在这个函数中参数 env1 是当前状态下虚拟 CPU 的状态，其返回值是异常或中断的类型，也就是说，如果在执行过程中出现中断或调用，则跳出 cpu_exec 函数中的翻译-执行大循环，做异常处理，之后再循环执行 cpu_exec 函数，直到源程序被模拟执行完。记录和重放层就植根于 cpu_exec 这个函数内，可以根据用户对虚拟机的控制命令确定是否继续翻译-执行，还是重建历史状态。

在记录和重放层，存在几种不同的状态：正常执行状态、重建历史状态和根据历史状态记录正向执行。起始状态是正常执行状态，接到返回历史执行点的命令之后转移为重建历史状态，在完成重建历史执行点之后，状态切换到根据历史记录正向执行状态，当正向执行完成所有历史状态的正向执行之后切换到正常执行，继续进行历史状态的记录。

在正常执行过程中，保存寄存器状态到检查点中的函数原型如下：

```
void save(CPUX86State *env,ulong start_code, ulong end_code){
    if((start_code < env->eip)&&( env->eip < end_code )){
        checkpoint_id++;//检查点编号自增
        ckpoint[checkpoint_id].mark = 1;//检查点有效
        mem_last = 0; //初始化检查点内存链
        save_cp(env,checkpoint_id); //将 cpu 状态存入检查点
    }else { error("overflow checkpoint"); }
}
```

参数 env 是当前虚拟 CPU 的状态，主要由 save_cp()函数做处理，将寄存器的状态存入检查点中。对于重建历史的寄存器状态来说刚好相反，直接读取已经记录下的寄存器值，修改虚拟 CPU 的状态，在这里就不再赘述。

内存状态就比寄存器状态的记录和恢复要复杂的多，对于内存的记录，是通过代码生成模块来管理的，在 4.2.1 节中已经介绍过了，这里把焦点放在内存状态的恢复上面。由于内存在记录的时候是按照基本块的执行顺序来增量式保存在检查点中的，所以在重建历史内存状态的过程中，需要逆着检查点的编号的顺序，对应每一个保存下来的内存单元进行恢复。当然，在恢复之前，需要先确定要恢复那些检查点内保存的内存状态历史数据。需要恢复的历史的检查点通过需要逆向回去的点的 PC 来决定，并通过 Search_CP() 函数来确定此 PC 对应的检查点的编号，设定为 I。程序当前保存的最后一个检查点的编号设定为 M，则恢复内

存状态的时候，按照编号 M，M-1，...，I+1，I 的顺序依次将内存读取出来覆写到原地址中。

4.2.3 系统调用模拟

QEMU 模拟器的核心部分是循环翻译和执行的过程，在翻译代码的执行过程中，如果有系统调用，则跳出翻译执行循环，执行系统调用和信号异常处理。在 QEMU 系统中的系统调用是对宿主机系统调用的封装，将系统调用的调用号、调用参数及其操作记录下来，便于逆向执行过程中对系统调用进行模拟。但是这样仅仅只能处理一些简单的系统调用。

系统调用的模拟函数为 do_syscall，其原型是：

```
env->regs[R_EAX] =  do_syscall( env, env->regs[R_EAX], \
                                env->regs[R_EBX],env->regs[R_ECX], \
                                env->regs[R_EDX],env->regs[R_ESI], \
                                env->regs[R_EDI],env->regs[R_EBP]);
```

模拟器 CPU 的 EAX，EBX，ECX，EDX，ESI，EDI 和 EBP 分别作为系统调用的调用号以及一系列调用参数模拟传入 do_syscall，在 do_syscall 中对这些系统调用进行模拟，并将系统调用的返回值赋给模拟器 CPU 的 EAX 寄存器。那么可以在进入 do_syscall 函数的开始，对这些系统调用号和参数进行保存，并将最后返回的值进行保存。同时，对所有的系统调用进行逐一分析其内部行为特征，并做相应仿真。但是，并非所有的系统调用都能够获得与原来一致的状态，所以对于某些复杂的系统调用，利用这种方式还是没有办法解决的，这将是今后的研究的一个重点问题。

4.3 调试器后端 R-GDB

为了使得底层的逆向执行实现对上层的调试的“透明”化，PORD 将传统调试器 GDB 和虚拟执行部分结合起来。将 GDB 进行修改，增加逆向调试的命令解析和命令数据包的发送构成 R-GDB 部分，同时修改 R-GDB 和虚拟机通信的远程调试协议，使得可以通过 R-GDB 控制虚拟机的运行。由此可见，PORD 的逆向调试只能是远程调试模式。因为 PORD 毕竟还是调试工具，所以不能舍弃原来已经存在的调试的功能，只能在保持原有功能的基础上增加额外的逆向执行功能。所以，大部分的 GDB 的调试功能在 PORD 中得到了保留。

4.3.1 逆向调试命令及流程

为了支持逆向调试，增加了一系列逆向调试的命令，详细命令解释见表 4.3。

表 4.3 逆向调试相关命令

命令行	功能
reverse <line_number>	逆向回源 C 代码中第 line_number 行
reverse <source_pc>	逆向回源二进制代码中的 PC 地址处
rebase	在逆向的过程中直接返回到最后正常执行的位置

对于以上命令中 reverse <line_number>和 reverse <source_pc>，从根本上来说，是实现相同的功能，使得程序回滚到某历史的执行点，只不过，前者是通过源程序中 C 代码的行号来确定返回历史执行点的，而后者是通过二进制代码的指令地址来确定的。命令 rebase 是为了使程序在逆向的过程中，能够快速返回到正常执行的最后执行点。

一般来说，程序员首先需要启动 PORD 的虚拟机端，运行被执行程序，并开启虚拟机的 R-GDB 的远程调试桩，用于接收远程的调试命令数据包。此时 PORD 处于阻塞状态，等待远程的调试命令。然后程序员开启 R-GDB 命令解析端通过 UDP 端口连上虚拟机内的远程调试桩。这个时候就可以通过 R-GDB 对虚拟机进行命令控制了。首先对程序下断点 A，然后启动被调程序运行，直到断点 A，此时可以通过 R-GDB 类似传统的调试器一样输入查看和 watchpoint 命令查看程序执行状态，如果发现此时程序状态跟预期不一致，那么可以通过 reverse <line_number>或者 reverse <source_pc>指令返回到可疑的指令，再检查程序状态，可以反复回滚直到找到程序错误的根源。如果在回滚多次之后，没有发现错误，想继续执行下去，则执行 rebase 命令，此时程序将返回到逆向时的断点 A 处，继续往下执行。

4.3.2 R-GDB 的实现

为了向 GDB 中增加逆向调试相关命令，则需要对原 GDB 的命令解释部分和命令动作部分进行修改，使得能够解析命令，并根据相应的命令执行相应的操作。

GDB 中有自己的命令表，存储各种命令的信息。在用户键入命令时，GDB 会首先查找命令表，找到相关信息然后才会执行相关的命令。命令表用于存储目前 GDB 支持的所有用户命令。每一个命令有其命令名、类型、相应内部函数的指针等属性，这些被封装在 cmd_list_element 结构中（参见附录 A cmd_list_elemet 结构）。GDB 中有许多命令表，所有命令表作为树中的结点，构成树状结构，作

为“根”的命令表是主命令表(头指针是 cmdlist)。树中任何命令都可以有其子命令表，指向子命令表的指针存储于该命令的 prefixlist 域中，例如 info 命令存储在 cmdlist 表中，而各种 info 子命令(如后文定义的 info processes 等)则存储在由 info 命令的 prefixlist 域指向的子命令表 infolist 中。一般地，命令行中的命令名存放在主命令表中。如果该命令还包含着子命令表，则由第一个参数区分的各子命令存储于第一级子命令表中，依次类推，可以有第二、三级子命令表等。执行 info 命令时，首先在主命令表 cmdlist 中找到 info 命令，然后在子命令表中按 info 命令的第一个参数继续查找匹配的子命令，找到并执行它。

基于 cmdlist 列表，R-GDB 利用 addcom 函数向 cmdlist 中增加表 4.3 中的逆向相关命令，并将命令和相关的命令解析函数结合起来，实现逆向命令的响应动作。向 cmdlist 表中增加逆向命令 reverse <Line_number>和解析函数 undo_command() 的详细函数如下：

```
/*发送逆向命令数据包函数*/
static void send_reverse_packet(unsigned int addr,char* line);
static void undo_command (char *cmd, int tty){
    unsigned long num = 0; /**/
    unsigned int addr = -1;
    if(cmd == NULL) {
        printf_filtered("undo have one argument as line number\n"); return;
    }
    num = strtol(cmd, NULL, 10); /*命令参数读取，Line_number 或 PC*/
    if(num <= 0)  { /*检查参数合法性*/
        printf_filtered("line number must bigger than 0\n"); return;
    }
    get_pc_from_line(num, &addr); /*获得行号 num 对应的 PC 地址 addr*/
    if(addr == -1) {    /*地址验证*/
        printf_filtered("undo address error\n");        return;
    }
    /*Send pakage to remote*/
    send_reverse_packet(addr,cmd);   /*将命令打包并发送到 gdb 服务桩*/
}

static void init_remote_reverse (void){
.... //
/*增加命令 reverese，并将此命令和函数 undo_command 结合起来，并加入
到 cmdlist 命令列表*/
add_cmd ("reverse", class_obscure, undo_command, _("\
Restore program run to addr given in reverse."),&cmdlist);
... //
}
```

4.3.3 远程服务桩的实现

在远程调试中，所有的 GDB 的命令以及命令的响应都将被以包的形式发送。一个包将以字符“$”为前导，然后是实际的包数据，接着是字符“#”，最后是两位数字的校验和（checksum）。校验和是在字符“$”和字符“#”之间的所有字符和模 256 的值。其形式为 $packet-data\#checksum$，当主机或者目标桩接收到数据包之后，将会返回一个回应，其字符串是“+”（代表正确接收）或者“－”（要求重发一次）。对于实际包数据中包含的字符序列，只要不包含“#”、“$”和“X”都可以。数据包中的数据域之间通过“,”“;”或者“:”来分开。对于逆向调试命令的数据包，设计形式如下：$u,0000\#checksum$，u 代表逆向，0000 是数据位，表明将要逆向过去的源二进制的代码地址 PC。

作为 GDB 的远程服务桩必须响应‘g’，‘G’，‘m’，‘M’，‘c’和‘s’数据包，其他的数据包命令是可选的。它们分别是读/写寄存器、读/写内存、continue 运行和 step 单步命令的数据包类型区分符。除了在远程服务端响应这些数据包以外，还需要响应我们的逆向调试数据包‘u’。

当然，为了响应逆向调试命令数据包，虚拟机部分的程序控制流需要进入到 GDB 的远程服务桩 gdbstub。由此可知，为了逆向，必须先使程序进入调试中断，即在某处下断点。这样，解释器在生成微指令序列的时候会先对 gdbstub 中断点列表进行分析，如果当前被翻译代码 PC 在断点列表中，则在此代码之前产生一个调试微操作 OP_DEBUG，与此相对应的，翻译器将产生调试中断代码，在执行的过程中，将产生调试中断，跳出翻译-执行代码大循环，从而进入中断处理循环，开始处理调试中断，即远程调试服务桩 gdbstub 中。在 gdbstub 中最核心的数据包处理函数是 static int gdb_handle_packet(GDBState *s, CPUState *env, const char *line_buf)函数。它是一个根据数据包 line_buf 解析数据，并执行相应的调试动作的函数。为了使得 gdbstub 可以响应逆向调试数据包，则在此函数内增加‘u’数据包的处理部分，下面是部分代码列表：

```
/*switch 结构中数据包‘u’的处理*/
    case 'u':
        unsigned long pc = strtoul(p,(char **)&p,16); /*获得 u 数据包传过来的 PC 地址*/
        if(pc != 0){
            //then we must find the first pc of tb
            unsigned long kk=0;
            /*从源 PC 和翻译代码 PC 映射的查找树上获取对应的翻译代码块首 PC*/
            unsigned long first_pc = search_BST(pcTBTree, pc,&kk);
            char buf[256];
            sprintf(buf,"OK+%0x",first_pc); /*构造发送数据包*/
```

```
            put_packet(s,buf);
            global_first_pc = first_pc; /*通过 global_first_pc 来控制逆向的 PC*/
            /*we will insert bp again*/
            append_bp_pc = pc; /*在此即将回滚的地址上下断点*/
            rev_state = RESTORE; /*修改此时虚拟机逆向的运行模式*/
        }
        break;
```

以上代码中 Search_BST()函数的原型是

unsigned long search_BST(BiTree T, unsigned long pc,unsigned long *reserved);

参数 pc 为源程序的 PC，而参数 reserved 返回对应的保存在查找树 T 上的检查点的编号。查找树 T 保存了已经缓存的翻译代码块（TB）的首地址 PC 和其对应的源二进制代码 PC 地址映射列表以及这个基本块的代码大小。这样，通过 R-GDB 中的源程序的 PC 就可以得到此 PC 对应的基本块首 PC，进而得到对应的翻译代码块信息和其对应的检查点编号，这样就可以通过编号对程序历史状态进行恢复了。

在获得需要逆向的翻译代码 PC 地址之后，通过 rev_state 修改虚拟机执行状态为 RESTORE，即逆向重建历史执行点状态，这样，在退出远程服务桩 gdbstub 的循环处理，进入翻译-执行循环后，虚拟机会根据 RESTORE 状态标志，根据 global_first_pc 来恢复历史执行点。在历史状态重建完毕后，虚拟机的状态将转为 REVERSE_RUN，在 REVERSE_RUN 状态下，程序并不进行记录状态操作，仅仅只是根据日志文件重建状态。

4.4 本章小结

本章在第三章的基于 QEMU 的逆向执行的基础上设计和实现了逆向调试器 PORD。根据逆向执行中所采取的利用程序状态日志文件的记录和重放技术，PORD 修改了动态二进制翻译器 QEMU，使其具有执行时动态记录程序状态到日志文件的功能，并修改传统的调试器 GDB，构成 R-GDB，使其与 QEMU 结合起来，实现程序员透明地可逆向调试程序。在虚拟机执行部分，动态的利用检查点技术来记录程序的状态，控制程序的执行控制流，并将调试命令的输出结果发送到 R-GDB 端；而在 R-GDB 中，解析用户输入命令，并向虚拟机部分发送命令相关数据包，同时接收虚拟机部分对于调试命令的响应，显示命令输出结果给用户。

第五章 实验结果与分析

在第三章中对基于 QEMU 的逆向执行技术进行了相关探讨，并在此基础上在第四章中设计了基于这一技术的逆向调试器 PORD。在本章中，将从功能性、执行性能、适用性等方面对 PORD 进行实验验证。实验结果表明本文提出的基于 QEMU 的逆向调试器技术是可行的，并且基于这一技术实现的逆向调试器 PORD 能够为跨平台的软件开发提供一个性能较好的调试平台，并且在具有相同指令集的平台下，PORD 能够以接近本地 CPU 执行速度执行被调试程序。

5.1 实验选取

本文选取了经典的快速排序（quicksort）算法 C 程序来验证逆向调试器 PORD 的可逆向执行功能。在本文的 quicksort 程序中以 20 个整数数组为输入，并对每一次扫描迭代后的数组进行检测，然后返回到程序的任意历史执行点，为了便于更好的验证程序的正确性，在编译被调试的 quicksort 程序时，开启了-g 选项，在程序中加入了调试信息，这样便于确认程序的执行过程。

在验证 PORD 的逆向调试功能的同时，为了获得其他程序在 PORD 上的执行性能和效率分析，本文还选取了开源的 BYTEmark[35]程序组作为程序运行时间和内存开销的基准程序来验证不同间隔检测点粒度对时间、空间的开销。详细的 BYTEmark 程序组清单见表 5.1 所示。

表 5.1 BYTEmark 程序描述

程序	功能描述	性能描述
Number Sort	32 位整形数组排序	一般的整数性能
String Sort	任意长字符串的数组排序	内存移动性能
Bitfield	一系列的位操作	位操作性能
FP Emulation	浮点数计算小软件	
Assignment	任务分配算法	大量的横向和纵向的整数数组移动
Fourier	数值分析函数	FPU 的性能测试
IDEA	密码学算法	大块数据移动
Huffman	文本和图压缩算法	字节、位和整数混合操作
LU Decomposit	解线性方程算法	浮点数检测

同时本文针对实际应用系统中的一系列应用软件也进行了一定的验证，例如，linux 系统中常用的命令 ls，tar 等应用程序。本章只是对这些程序进行了功能性的实验验证，并进行了相关的讨论。

本文所有实验的验证平台为 Intel(R) Pentium(R) 4 1.80GHz CPU 512k Cache，512M 内存，操作系统 Redhat9、linux 内核 2.4.20，Gcc 版本为 3.2.2。

5.2 实验验证与结果分析

在本节中我们将对本文中提出的逆向调试技术以及逆向调试器 PORD 进行实验验证。首先我们对 PORD 的逆向功能进行验证，然后针对不同的翻译模式以及不同的检查点粒度的时间和空间的开销进行实验和分析。启动 R-GDB 界面如图 5.1 所示。

```
[root@hdc bin]# ./gdb /home/lt/test
R_GDB v 0.1 modified on GNU gdb 6.5
Copyright (C) 2006 XiDian University.
=1= GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
There is absolutely no warranty for GDB.
=2= R_GDB is free software, Modified by XDU, which make gdb have ability of
debugging in reverse.   Modified by Mu 2007.5 .

  ########             ######    ########   ########
  ##     ##           ##    ##   ##     ##  ##     ##
  ##     ##           ##         ##     ##  ##     ##
  ########   #######  ##   #### ##     ##  ########
  ##   ##             ##    ##   ##     ##  ##     ##
  ##    ##            ##    ##   ##     ##  ##     ##
  ##     ##            ######    ########   ########
 ..Using host libthread_db library "/lib/libthread_db.so.1".

(R_GDB) l 5
1       #include <stdio.h>
2
3       int main()
4       {
```

图 5.1 R-GDB 启动界面

被调试程序快速排序程序 quicksort 编译时，被加上-g 选项，加入了调试符号信息。输入的整数数组 array 被初始化为从大到小的全序列数组，数组大小定为 10 个，其内容为 10~1。经过一次快速排序后数组应为 1，9，8，7，6，5，4，3，2，10，执行程序到最后，可以得到从小到大排序后的数组 1，2，3，4，5，6，7，8，9，10。然后逆向到第一次排序后的的位置，可以得出数组 array 跟预期数据一致，同时，也可根据 GDB 的查看寄存器和内存命令来查看程序的相关状态。实验表明 PORD 满足逆向执行的功能，能够忠实地完全地重现历史的执行点的状态。

5.2.1 针对不同翻译模式的负载分析

PORD在代码的植入中有两种不同的方式，一是针对跨平台的Binary-Translate模式，二是只能用在具有相同指令集的源程序和宿主机平台上的Binary-Copy模式。本文实验验证时，选取的源程序和宿主机的指令集架构均是X86的32位平台，执行的验证程序是BYTEmark程序组的Numeric Sort，String Sort，Bitfield，FP Emulation，Assignment,Fourier，IDEA，Huffman，LU Decomposit程序。将每个程序在本地CPU、Binary-Translate植入模式下和Binary-Copy植入模式下执行10次，然后取平均值，得出表5.2的实验数据，单位是迭代次数每秒，代表执行BYTEmark程序的速度。

表5.2 BYTEmark程序分别在本地执行、Binary-Translate模式和Binary-Copy模式下的时间负载 （迭代次数/秒）

程序	本地执行	Binary-Translate 模式	Binary-Copy 模式
Number Sort	0.20	0.05	0.18
String Sort	0.03	0.01	0.03
Bitfield	46264	6385	45858
FP Emulation	0.03	0.01	0.01
Assignment	0.03	0.01	0.02
Fourier	9.95	1.01	2.28
IDEA	0.42	0.06	0.18
Huffman	0.37	0.04	0.10
LU Decomposit	0.10	0.01	0.01
Number Sort	0.20	0.05	0.18

由表5.2中可知，本地执行速度是最快的，当然，对于Number Sort，String Sort，Bitfield，Number Sort这几个测试程序来说，Binary-Copy拥有和本地执行非常接近的速度，远远大于Binary-Translate模式下的程序植入。而对于其他几个BYTEmark的程序来说，Binary-Copy模式的速度却跟Binary-Translate模式很接近，那是因为有些源程序的浮点指令不支持直接利用Binary-Copy模式进行本地直接执行，反而转向利用Binary-Translate模式下的翻译执行程序的缘故。实验对于BYTEmark测试程序组的几个程序的不同执行时间效率如图5.2所示。

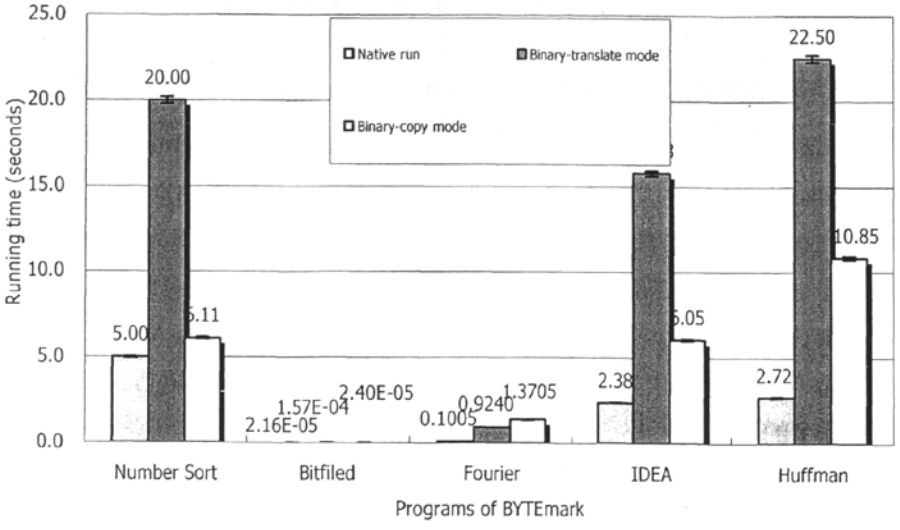

图 5.2 三种模式下 BYTEmark 程序的时间开销

5.2.2 针对不同检查点粒度的负载分析

逆向调试器对程序的默认检查点的间隔粒度是一个基本块，也就是说每翻译-执行一个基本块就需要保存一个检查点。本节中将对不同检查点间隔粒度的时间和空间上的开销进行实验和分析，以获得更佳的检查点的间隔粒度。实验方式：从 BYTEmark 的程序中任意选取 3 个程序，以 4 个基本块为单位递增程序状态检查点的粒度，即以 1，4，8，12....为检查点的间隔，对每一种检查点的间隔粒度来说，每个程序执行 10 次，获得每次程序运行的执行时间和内存耗费峰值，最后取平均值，绘制成不同间隔点程序的时间开销趋势图如图 5.3 所示。

由图 5.3 可以看出，Bitfield，String Sort 和 Fourier 为被运行程序，在 PORD 的记录检查点的间隔粒度越大，PORD 耗费在执行这些程序的时间上的开销越小，同时间隔粒度在 1~8 基本块之间时，时间的开销的下降速度比较快，而更大的间隔粒度的时间开销的下降并不是非常的明显，因为在需要对不同基本块内的内存数据进行合并会耗费一定的内存地址状态的查询和处理时间。

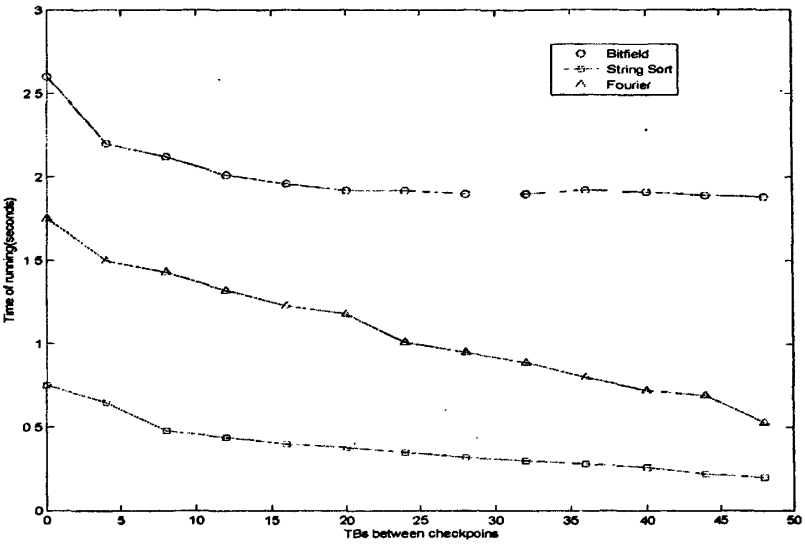

图 5.3 不同检查点间隔粒度的时间开销

图 5.4 给出了不同检查点间隔粒度的内存空间的耗费。由图中可以看出 LU Decomposit，String Sort 和 Fourier 三个被运行程序，随着 PORD 记录检查点见间隔的增大，PORD 的内存空间的耗费随之变小，整个过程呈下降趋势。分析其原因，在于增大检查点间的间隔，那么这些间隔中间的寄存器的保存将大大减少，同时，因为检查点间隔之间的多个基本块之间会存在对一个内存的多次写操作，对于每个基本块来说，应该都记录下来的，但是，因为增大了检查点的间隔粒度，这样相应地，重复内存将只会被保存一次，也大大减少了内存空间的开销。

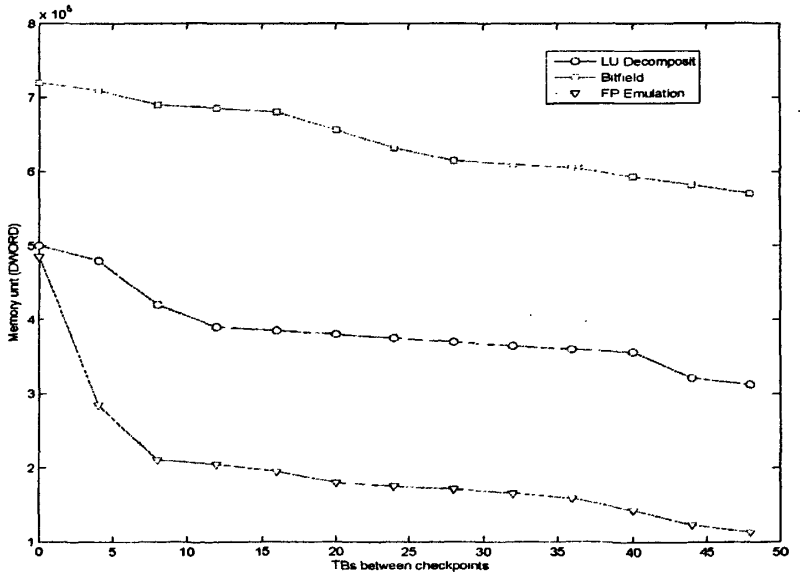

图 5.4 不同检查点的间隔粒度的内存开销

由上可知，对于记录程序执行状态的检查点的间隔单位来说，并非间隔粒度越大就越好。从另一方面来说，大粒度的检查点间隔，则程序状态恢复到需要回

滚的历史执行点的时候，将会耗费更多的正向执行时间。而且，本文仅仅是针对单核单线程的确定性程序，在以后的研究中可能会涉及到程序并发以及并行的同步的问题，这样检查点的间隔粒度就应该是更为灵活的，而不是固定的基本块执行单位。

5.2.3 整体性能分析和讨论

最后，通过对 BYTEmark 程序在本地 CPU 直接执行、Binary-Translate 模式的执行和 Binary-Copy 模式的执行的时间对比发现，程序在 Binary-Translate 模式下执行时间开销是本地 CPU 直接执行时间开销的 8~12 倍，而在 Binary-Copy 模式下仅仅只有 1.3~2 倍的时间开销。同时，对比其它的可动态二进制植入工具 LVM 和 Pin（均未做执行程序的状态记录），LVM 是本地 CPU 执行时间的 80 多倍，Pin 约是本地 CPU 执行时间的 11~14 倍。由此可以看出，PORD 的 Binary-Translate 模式在可提供跨平台的前提下，仍然保持了非常不错的执行速度；PORD 的 Binary-Copy 模式提供了非常接近本地执行速度的程序调试环境。图 5.5 给出了这几项的平均执行开销（LVM 和 Pin 执行时间来自文献[14]）。

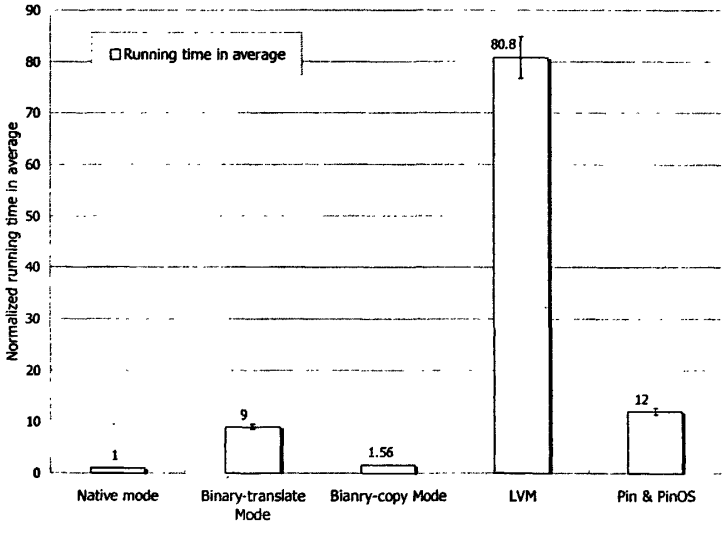

图 5.5 平均运行时间

在对 ls、tar 等 linux 系统下的应用程序的逆向调试功能性实验表明，本文实现的逆向调试其 PORD 在一定程度上能够满足一般应用程序的逆向调试需求，但是对于那些具有多线程、多输入的大量非确定性因素的应用程序，例如 Mozilia，Xserver 等具有可视界面的应用程序。这是因为逆向执行的虚拟机暂时并不支持某些中断和系统调用的模拟所造成的，将是今后进一步的研究所要解决的问题，需要尽更大的努力做更多的研究工作。

5.3 本章小结

本章针对第三章和第四章提出的逆向调试技术以及逆向调试器 PORD 做了一定的实验验证和分析。利用快速排序程序在 PORD 上的逆向回滚，检查了 PORD 逆行执行的功能性实现。同时，通过 BYTEmark 程序对 PORD 的各方面的性能进行验证。分析了不同检查点间隔粒度对时间和空间的开销的影响，得出检查点间隔粒度大小适中为佳的结论。同时也将 LVM 和 Pin 等动态二进制植入工具的运行时间同本地执行、Binary-Translate 模式和 Binary-Copy 模式下的程序运行时间开销做了比较，并得出结论，PORD 可以为程序员提供执行快速的跨平台调试环境，并且 PORD 可以在非常有限的影响下增加传统调试器的逆向调试功能，为广大的程序员提供强大便捷的功能。

第六章 总结与进一步工作

6.1 本文工作总结

随着软件系统的日益复杂，程序的错误诊断工作变得越来越难。逆向执行和逆向调试作为一种错误诊断和程序理解很重要的手段已经初步进入嵌入式软件开发的应用领域。从上世纪七八十年代就开始了这方面工作的研究，有从语言设计方面入手的逆向执行研究，也有从程序植入后状态保存方向开展的研究等等。随着近年来硬件价格的降低和运算速度的提高，虚拟执行也愈加发展起来。由此也带动了利用虚拟机来控制程序流程的程序逆向执行的研究。

本文就是基于动态二进制翻译器 QEMU 的程序逆向执行的研究。本文在分析了逆向执行方面的发展和程序逆向执行实现方式之后，选取基于动态二进制翻译技术为二进制程序进行植入保存程序执行状态的代码，使得程序在翻译器被虚拟执行的过程中，程序的执行状态就被自动地保存下来，在需要的时候，虚拟机读取这些程序状态的日志文件重建程序的历史执行状态。在分析了基于动态二进制翻译器 QEMU 的逆向执行的基础上，本文设计和实现了基于此技术的逆向调试器 PORD。通过修改 QEMU 支持程序的逆向执行，并且增加 GDB 的命令解析和远程调试协议，构成 R-GDB，在 QEMU 中加入逆向执行的远程调试服务桩，控制虚拟机部分的程序执行动作包括逆向执行，最终将 R-GDB 和虚拟机部分结合起来实现“透明的”逆向调试。

最后的实验验证表明本文提出的基于 QEMU 的逆向调试技术是可行的，并且基于此技术实现的逆向调试器 PORD 在跨平台下为程序员提供较快速度的具有回滚功能的调试环境。而对于具有相同指令集的源程序和宿主机平台，程序在 PORD 上调试执行可以获得接近本地执行的快速执行速度。

6.2 进一步研究方向

尽管逆向执行概念的提出非常早，直到今天，逆向执行在程序的执行方面的应用还是非常的少，并且现在为了实现逆向执行的许多问题也并非都能非常好的被解决。就程序的逆向调试而言，现今只有 Simics 和 GDB 在它们的最新版本中加入了逆向调试的试验性功能，远远没有达到可以实用的阶段，还有很多的研究问题没有解决，而更多的问题也随着计算机的发展被提出来。多核多线程可以更大容量更快速度地进行计算，但是多核多线程的并行和并发特性，也带来许多问

题，诸如程序中线程的同步、关键数据的互斥等，当然也给逆向执行带来很多的问题。同时，现存的大量的程序都有中断和异常处理的身影，对于中断和异常的精确处理，还是程序逆向执行方面一个重要的承待解决的问题。另外，对于 QEMU 动态翻译系统的翻译机制，可以有更进一步的优化。

6.2.1 多线程并发程序的逆向执行

本文仅仅处理了单核单线程程序的逆向执行，与实际软件工程应用中的逆向调试还有很大距离，但是因为本文基于的动态二进制翻译器 QEMU 是单线程的模拟器，其中的代码翻译和执行过程并不是线程安全的。现在已经有多线程方面的研究[12]，另外，因为从微观上看多线程单核程序的并发其实也是线性的执行流程，只是代码段的执行先后顺序是非确定的，如果同样需要基于 QEMU 进一步的实现多线程并发的逆向执行，有两点问题需要解决：一是改造模拟器 QEMU，使其代码的翻译和执行段是线程安全的，这样，当源程序上生成一个新的线程的时候，翻译器 QEMU 上同样也生成一个新的线程，翻译和执行新的线程中的代码，当然，每个线程需要有自己的模拟线程栈；二是需要重新设计检查点的保存和恢复算法，因为并发程序执行的非确定性，检查点的保存和恢复对于忠实的重现历史执行点状态绝对是极其重要的，本文的检查点保存策略是基于确定性程序执行这一特性，对于多线程并发是不合适的。

6.2.2 动态二进制翻译机制的进一步优化

本文讨论的逆向执行技术是基于动态二进制翻译器 QEMU 的，虽然模拟器 QEMU 已经非常优秀，但是还有很多的优化余地，这样可以加快被调试程序的虚拟执行的速度。例如，动态二进制翻译系统的标志位处理的延迟优化，异常处理优化等。重要的是，需要改变 QEMU 模拟器的代码生成方式，因为利用 QEMU 模拟器生成的代码数量比源二进制代码大了 5~8 倍，这样势必造成较低的执行效率的翻译代码，进而影响程序模拟的速度。

致谢

本文是我在硕士研究生期间学习和工作的一个总结，在此谨向所有关心，帮助和支持过我的老师，同学，朋友和亲人致以最衷心和最真挚的谢意！

首先，向我的导师刘西洋教授表示深深的谢意，感谢刘老师三年多来对我的学习和生活的关心和支持。正是刘老师对我的深切的关怀和谆谆的教诲使我真正有机会了解研究生阶段应该做什么以及怎样去达到自己的目标。刘老师严谨的治学态度，孜孜以求的工作作风，对研究透彻的理解无时无刻不在感染着我。刘老师深厚的理论功底，丰富的项目经验和源源不断的新思想新想法，永远都是我学习的榜样！在今后的工作和生活中，我将牢记刘老师教给我的求学做人的精髓，一定不辜负刘老师的殷切期望！

同时也非常感谢课题组各位同学对我的帮助和支持。感谢各位师兄师姐——刘鹤辉，雷宁，曹国栋，李航，丁军，王黎明，王磊，刘洋，方晓琴，孙军国，朱修彬等对我在研究生期间生活和学习上的无私帮助。感谢实验室里的各位同门——张苗，柏志文，杨雷，张中宝，薛鹏，金文辉，徐亮，能够和他们在一起学习一起研究，使我的生活更加充满乐趣。还要特别感谢王艳，陈敬林，杜文，陈晓萍，李晋，章华蔚，穆浩英，李春香，吴桂花，秦英，宋爱龙等师弟师妹们，正是他们的支持与帮助才使我的论文得以顺利完成。尤其是同一课题组的柏志文，王艳，穆浩英，李春香，没有他们无私的帮助和支持，我的研究无法顺利进行，

最后，将我最真诚的发自内心的谢意送给我的家人和苗苗，感谢我的父母，感谢我的祖父母，感谢我的弟弟妹妹等亲人，是他们使我懂得如何才能成为一个对社会，对家庭有用的人。二十余年无私的爱与支持将铭记在心，我将加倍努力，不辜负他们的深厚期望！感谢苗苗，没有她的陪伴，没有她的鼓励，我的生活将索然无味，黯然失色。我一定尽我最大的努力，不辜负苗苗的殷切期望！

再一次对我的导师刘西洋教授和所有关心过、帮助过和支持过我的老师、同学、朋友、亲人致以崇高的感谢和敬意！

参考文献

[1] Andreas Zeller, Why Programs Fail: A Guide to Systematic Debugging, Morgan Kaufmann Publishers Inc., San Francisco, CA, 2005

[2] M.V. Zelkowitz, Reversible Execution, Comm. ACM, vol. 16,no. 9, p. 566, Sept. 1973.

[3] C. Lutz. Janus: a time-reversible language. A letter to Landauer. http://www.cise.ufl.edu/~mpf/rc/janus.html, 1986.

[4] Tetsuo Yokoyama, Robert Glück: A reversible programming language and its invertible self-interpreter. PEPM 2007: 144-153

[5] S.I. Feldman and C.B. Brown, IGOR: A System for Program Debugging via Reversible Execution, Proc. SIGPLAN Notice, pp. 112±123, Jan. 1989.

[6] T. Moher. Provide: A process visualization and debugging environment. IEEE Trans. on Software Engineering, 14(6):849–857, 1988.

[7] H. Lieberman and C. Fly. Bridging the gulf between code and behavior in programming. In Proc. of Conference on Human Factors in Computing Systems, pages 480–486, 1995.

[8] S. Chen, W. K. Fuchs, and J. Chung. Reversible debugging using program instrumentation. IEEE Trans. on Software Engineering, 27(8):715–727, 2001.

[9] T. Akgul and V. J. Mooney. Instruction-level reverse execution for debugging. In Proc. of Workshop on Program Analysis for Software Tools and Engineering, pages 18–25, 2002.

[10] T. Akgul, V. J. Mooney, and S. Pande. A fast assembly level reverse execution method via dynamic slicing. In Proc. of 26th International Conference on Software Engineering, pages 522–531, 2004.

[11] C. Demetrescu and I. Finocchi. A portable virtual machine for program debugging and directing. In Proc. of Symposium on Applied Computing, pages 1524–1530, 2004.

[12] Michiel Ronsse, Koenraad De Bosschere: RecPlay: A Fully Integrated Practical Record/Replay System. ACM Trans. Comput. Syst. 17(2): 133-152 (1999)

[13] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operatingsystems with time-traveling virtual machines. In Proceedings ofUSENIX 2005 Annual Technical Conference, 2005.

[14] Toshihiko Koju, Shingo Takada, Norihisa Doi: An efficient and generic reversible debugger using the virtual machine based approach. VEE 2005: 79-88

[15] Virtutech. one commercial product virtutech simics. It is a generalpurposedevelopment tool for reversible execution.

[16] GNU. Gdb: The gnu project debugger. www.gnu.org/gdb

[17] F. Bellard. Qemu, a fast and portable dynamic translator. InProceedings of USENIX 2005 Annual Technical Conference, 2005.

[18] Shyh-Kwei Chen, W. Kent Fuchs, Jen-Yao Chung: Reversible Debugging Using Program Instrumentation. IEEE Trans. Software Eng. 27(8): 715-727 (2001)

[19] Jikes RVM (Research Virtual Machine), http://jikesrvm.org/

[20] Bitan Biswas, Rajib Mall: Reverse Execution of Programs. SIGPLAN Notices 34(4): 61-69 (1999)

[21] Erik R.Altman, David Kaeli and Yaron Sheffer, "Welcome to the Opportunities of Binary Translation," Computer, Vol 33, No 3, March 2000, IEEE Computer Society Press, pp 40-45.

[22] C.Cifuentes and V.Malhotra, "Binary Translation: Static, Dynamic, Retargetable?," Proceedings International Conference on Software Maintenance. Monterey, CA, Nov 4-8 1996. IEEE-CS Press. pp 340-349.

[23] C.Cifuentes and M.Van Emmerik, "Recovery of Jump Table Case Statements from Binary Code," Proceedings of the International Workshop on Program Comprehension, Pittsburgh, USA, May 1999, IEEE-CS Press, pp 192-199.

[24] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation, pages 196–205, 1994.

[25] J. Larus and E. Schnarr. EEL: Machine-independent executable editing. In Proceedings of the ACM SIGPLAN 95 Conference on Programming Language Design and Implementation, pages 291–300, June 1995.

[26] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of win32/intel executables using Etch. In Proceedings of the USENIX Windows NT Workshop, pages 1–7, August 1997.

[27] B. R. Buck and J. Hollingsworth. An api for runtime code patching. Journal of High Performance Computing Applications, 14(4):317–329, 2000.

[28] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. Proceedings of the 3rd USENIX Windows NT Symposium, pp. 135-143. Seattle, WA, July 1999. USENIX.

[29] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, "Dynamo: A Transparent Dynamic Optimization System", In Proceedings of the ACM SIGPLAN '2000 conference on Programming language design and implementation, PLDI'2000, June, 2000

[30] Timothy Garnett. Dynamic Optimization of IA-32 Applications Under DynamoRIO. M.Eng. Thesis, MIT, June 2003.

[31] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney,S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customizedprogram analysis tools with dynamic instrumentation. In Proceedingsof the 2005 ACM SIGPLAN Conference on Programming LanguageDesign and Implementation, Chicago, IL, USA, June 2005.

[32] Prashanth P.Bungale , Chi-Keung Luk. Pinos: A programmableframework for whole-system dynamic instrumentation. In Proceedingsof the 3th International Conference on Virtual ExecutionEnvironments, June 2007.

[33] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation.Proceedings of ACM SIGPLAN 2007 PLDI 2007, San Diego, California, USA, June 2007.

[34] 一种基于检查点的并行程序调试器的设计与实现 刘建,汪东升,沈美明,郑纬民 清华大学计算机科学与技术系高性能计算技术研究所

[35] BYTE. a benchmark tool. It is designed to expose the capabilities of a system's CPU, FPU, and memory system.www.byte.org

[36] C. Demetrescu and I. Finocchi. A portable virtual machine forprogram debugging and directing. In Proceedings of Symposium onApplied Computing, 2004.

[37] a. E. S. H. Agrawal, R.A. DeMillo. Debugging with dynamic slicingand backtracking. SOFTWARE-PRACTICE AND EXPERIENCE,23(6):589 616, JUNE 1993.

[38] E. H. S. Hiralal Agrawal, Richard A. DeMillo. An execution back tracking approach to

debugging. Source IEEE Software, 8(3):21C 26, May 1991.

[39] B. Lewis. Debugging backwards in time. In Proceedings of theFifth International Workshop on Automated Debugging (AADEBUG2003), October 2003.

[40] B. C. N. Kumar and M. Soffa. Low overhead program monitoring andprofiling. In The 6th ACM SIGPLAN-SIGSOFT workshop on Programanalysis for software tools and engineering table of contents, 2005.

[41] Scott W. Devine , Palo Alto , Edouard Bugnion, Menlo Park, MendelRosenblum. Virtualization system including a virtual machinemonitor for a computer with a segmented architecture. In In United States Patent 6,397,242, May 2002.

[42] S. J. A. E. J. R. M. D. M. S. Bhansali, W. Chen and M. Drinic.Framework for instruction-level tracing and analysis of programexecutions. In Proceedings of the second international conference onVirtual execution environments table of contents, 2006.

[43] C. A. S. Srinivasan, S. Kandula and Y. Zhou. Flashback: A light weight rollback and deterministic replay extension for software debugging.In Proceedings of the 2004 USENIX Technical Conference,June 2004.

[44] Y.-M. W. C. V. H. J. W. Samuel T. King, Peter M. Chen and J. R.Lorch. Subvirt: Implementing malware with virtual machines. InProceedings of the 2006 IEEE Symposium on Security and Privacy,May 2006.

[45] H. Sutter and J. Larus. Software and the concurrency revolution.ACM Queue, 3(7), September 2005.

作者在读研期间的研究成果

在硕士研究生期间取得的研究成果如下：

一、参加科研情况

1．十一五”国家某项目

二、发表论文

[1] 刘涛，刘西洋. 基于动态二进制植入技术的可逆调试技术研究. 西安电子科技大学 2007 年研究生学术年会. 2007 年 10 月.

[2] Xiyang Liu, Tao Liu, Zhiwen Bai, Yan Wang, Haoying Mu, Chunxiang Li," PORD: a Reversible Debugging Tool using Dynamic Binary Translation" in Proceedings of the 14th IEEE Asia-Pacific Software Engineering Conference (APSEC 2007). Nagoya, Japan. December 2007.

附录

附录 A cmd_list_elemet 结构

```
struct cmd_list_element
{
    struct cmd_list_element *next;    /*指向表中下一个命令*/
    char *name;   /* 命令名称   */
    enum command_class class;/*类型*/

    /* 命令的函数定义*/
    void (*func) (struct cmd_list_element *c, char *args, int from_tty)

    union{
        cmd_cfunc_ftype *cfunc;   /* 不是 set 命令，调用函数 */
        cmd_sfunc_ftype *sfunc;   /* 是 set 或者 show 命令，首先定义变量，
                                  然后调用函数*/
    }function;
    void *context; /*命令的当前状态，可以为任意类型   */
    char *doc;
    show_value_ftype *show_value_func; /*为 set/show 命令设置输出函数 */
    int flags;                    /*标志位,存储是否命令是否存在是否老化 */
    char *replacement;
    struct cmd_list_element *hook_pre;     /*执行前要执行的其他命令*/
    struct cmd_list_element *hook_post;    /*执行后要执行的其他命令   */
    int hook_in;                           /*标示是否已执行   */
    struct cmd_list_element **prefixlist;  /*指向子命令表   */
    char  *prefixname;          /*如果此命令是前缀命令，这此为前缀命令名
*/
    char allow_unknown;    /* 如果是前缀命令，标示子命令是否已知.  */
    char abbrev_flag;       /* 如果是缩写则非零，tab 键可补齐 */
    char **(*completer) (char *text, char *word);       /*一个设置函数，主要
                                          完成函数名的补全工作 */
    cmd_types type;                 /*set 或 show 的类型 */
    void *var;                      /* 由 type 指向的变量指针*/
    var_types var_type;             /* VAR 的类型   */
    const char **enums;             /*指向枚举型列表 */
    struct command_line *user_commands;  /*指向用户定义命令字符串*/
    struct cmd_list_element *hookee_pre;   /*指向连接到此命令的命令 */
    struct cmd_list_element *hookee_post;  /*指向被此命令连接到的命令 */
    struct cmd_list_element *cmd_pointer;  /*如果是别名，连到原始命令 */
};
```

基于动态二进制翻译的逆向调试器的设计与实现

作者：刘涛
学位授予单位：西安电子科技大学