华中科技大学

硕士学位论文

一种改进QEMU精确异常处理机制的研究

姓名：余璐

申请学位级别：硕士

专业：计算机应用技术

指导教师：章勤

20080602

摘 要

QEMU 是一款非常流行的使用动态二进制翻译技术并支持多源多目标翻译的二进制翻译器。但是 QEMU 在支持精确异常方面仍存在不足，具体表现在当异常发生时，无法完全恢复到异常发生前 CPU 的正确状态，无法定位异常指令的具体位置。

针对 QEMU 支持精确异常不足的缺点，异常处理系统给出了一种对异常进行处理的机制，异常处理系统将异常分为可预测性异常和不可预测性异常，可预测性异常是指在异常未发生前就能定位异常位置的一类异常，不可预测性异常是指在异常未发生前无法确定异常位置，只有到异常发生时才能确定异常位置的一类异常，因此使用设置异常点法对可预测性异常进行处理，使用计算异常点法对不可预测性异常进行处理。当异常发生时异常处理系统实现对 QEMU 用户模式下异常指令准确定位。

异常处理系统集成 QEMU 远程调试功能，能够在与 GDB 调试器链接时，响应 GDB 命令请求，并能够在调试过程中监控 CPU 状态。当用户在执行过程中出现异常时，以网页方式显示异常处 CPU 状态，具有与用户交互强，操作简单的特点，因此具备很好的实用价值。

实验表明：异常处理系统能够对QEMU用户模式下异常指令实现准确定位。同时与GDB(GNU Debugger)相比，异常处理系统能够定位边界溢出指令位置而GDB无法定位该类异常。另一方面当异常发生时异常处理系统能给出异常的具体原因，而GDB仅能提供除法异常原因。最后异常处理系统能够给出异常点之前通用寄存器状态和标志位寄存器状态，而GDB则必须通过多次设置程序断点方式才能得到此类信息。因此在发生异常情况下，异常处理系统提供了比GDB更全面的信息，而这些信息有助于程序员提高调试程序的效率。

关键词：动态二进制翻译器，精确异常，QEMU，设置异常点法，计算异常点法

Abstract

QEMU is a dynamic binary translator, but there are some deficiencies in supporting the precise exception. When exception happens, it does not completely restore the CPU state, does not locate the position of exception.

Against the lack of QEMU support the precise exception, common exception handling system gives a way to improve the mechanism of the precise exception of QEMU. Common exception handling system divides the exceptions into two parts, one part is called predictable exception and the other part is called unpredictable exception. Using set exception point method to deal with predictable exception and using compute exception point method to deal with unpredictable exception. In the end common exception handling system realizes to locate the position of exception of user mode of QEMU.

Precise Exception handling system integrates the remote debugging of QEMU， it will respond to the requests of the GDB commands if it connect to the GDB and monitor the state of the process of debugging. It will return the result to the web if the exception happens and it has the features of strong user interaction, simplifies the operations, therefore it is very useful.

Result shows common exception handling system realizes to locate the position of exception instruction. Compared with GDB, common exception handling system could locate the position of BOUND exception but GDB could not do it. On the other hand, common exception handling system could give the reasons of exception but GDB could not do it. In the end, common could give the state of general registers and EFLAGS once, but GDB must spend more time to get this information by setting break points. Therefore, common exception handling exception provides faster and more information than GDB, it raises the efficiency when programmers debug the programs.

Key words: dynamic binary translation, precise exception, QEMU, set exception point method, compute exception point method

独创性声明

本人声明所呈交的学位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，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到，本声明的法律结果由本人承担。

学位论文作者签名：

日期：　年　月　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本论文属于　保密□，在_____年解密后适用本授权书。
　　不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　年　月　日　　日期：　年　月　日

1 绪论

1.1 选题的背景与意义

现在大多数的商业化微处理器为了兼容过去使用的软件，仍然保留着上一代产品的所有指令。尽管其中有的指令也许已经有几十年的历史，并且已经很少使用，但是硬件开发商们仍然不愿意去开发一套全新的 ISA(Instruction System Architecture)。因为如果处理器开发商这么做，一方面，基于原来 ISA 的软件就不能够再使用，客户在原来 ISA 上的软件投资就会浪费；另一方面，软件开发商觉得如果为一个全新的体系结构编写应用软件会存在一定的难度，不仅费时而且开销巨大。如果这种体系结构没有赢得足够的市场份额，那么软件开发商为其开发软件将会面临很大风险。基于这两方面的原因，限制了芯片设计的改革，同时也限制了新的竞争者进入芯片设计领域。为了改变这种局面，很多研究人员提出了代码移植的想法，而二进制翻译使得代码移植成为可能。

二进制翻译也是一种编译技术，它与传统编译器的差别在于其编译处理对象不同。传统编译器处理的是某一种高级语言，经过编译处理生成某种机器的目标代码。而二进制翻译处理的是某种机器的二进制代码，该二进制代码是经过传统编译器生成的，经过二进制翻译处理后生成另一种机器的二进制代码。

为了实现不同硬件平台间的代码移植，二进制翻译要兼顾高效性和正确性，在二进制翻译过程中要保证二进制翻译器的高效性的方法往往是采取动态优化手段,例如在动态二进制翻译过程中根据基本块交互的频繁程度将基本块链接形成更大的翻译块，以便减少翻译块频繁访问时间，提高翻译效率或者对翻译的指令进行重新排序提高流水线程度的方式来提高翻译的效率。除了保证翻译时的高效性以外也要保证翻译过程的正确性，动态二进制翻译器必须忠实地模拟源机器上的寄存器与内存状态并能对这些状态进行更新。在任一时刻,如应用软件产生异常或在响应外部中断时,动态二进制翻译器必须提交一个与在源机器执行时完全一致的程序状态。精确异常要求在一条指令发生异常时,该指令之前的任何指令已经执行完毕,而该指令之后的任何指令尚未执行。在执行翻译码时,如果某条目的指令发生同步异常,翻译器需要保证与该目标指令对应的源指令维持精确异常的特性[1]。

当翻译码中的指令发生异常时,如何找到源机器码中的指令,如何恢复出与之对

应的源体系结构的状态,都是二进制翻译器必须解决的问题[2]。

1.2 国内外研究概况

二进制翻译器按照实现分类，分为解释执行，静态翻译，动态翻译三种。解释执行对程序中的每条指令实时解释执行，解释一条执行一条，系统不保存也不缓存解释过的指令，不需要用户干涉，也不进行任何优化。解释器相对容易开发，不用花多大力气就可以与老的体系结构高度兼容，但效率很差，通常一条指令被解释为十几条指令甚至更多[3,4]。但解释执行容易保证程序的正确性，常常被用于辅助翻译。静态翻译是在程序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。一次翻译的结果可以多次使用。静态翻译器属于离线翻译程序，因而比起动态翻译器来说，有足够的时间进行更完整、更细致的优化，故效率较高。而且，静态翻译器还可以利用程序以往执行的记录进行优化，即基于 profiling 的优化，获取更好的优化结果。然而，静态翻译器因为无法得知间接控制转移的目标地址，因此可能无法完整地翻译一个程序，需要依赖解释器的支持，才能完成全部的翻译。解释器要求翻译的程序运行时必须能够支持老机器的全部运行环境，并且需要支持操作系统调用的翻译和映射，这些都是巨大的开销。动态翻译基于动态编译技术，在程序运行时对执行到的片段进行翻译，即输入的源二进制文件经动态分析，在运行时被翻译成目标机器码。它克服了静态翻译的一些缺点，比如说静态翻译时由于不能知道控制流中某点的寄存器或内存的值，从而不能得知控制流的流向。而且动态翻译还可以解决大部分实际情况中的自修改代码问题，而这对于静态翻译是不可能解决的。动态翻译可以执行一些静态翻译无法应用的优化，例如 dynamically-dispatched calls 和过程内嵌。动态翻译器对用户可以做到完全透明，无需用户干预[5-7]。虽然动态翻译器拥有上述诸多优点，翻译过程却由于受到动态执行的限制而不能进行更全面细致的优化，使得翻译生成的代码效率比静态翻译器较差[8]。在表 1.1 中表示的是解释执行，静态翻译，动态翻译这三种实现方法的优缺点对比。

同时二进制翻译根据全系统模式和用户模式又可以分为系统级翻译和应用程序级翻译。系统级翻译器，模拟整个硬件平台，翻译运行在模拟硬件上的所有程序包括操作系统。与系统级翻译相对应，应用程序级的翻译是指翻译器不翻译操作系统，只翻译基于操作系统的应用程序，而且翻译器也使用操作系统提供的库函数和系统

调用。在 1.2.1 和 1.2.2 节将分别讲解系统级翻译的精确异常处理以及应用程序级翻译的精确异常处理。

表 1.1 三种执行方式的比较

	优点	缺点
解释执行	容易开发，不需要用户干涉，高度兼容。	效率很差。
静态翻译	离线翻译，可以进行更好的优化，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，给用户的使用造成不便。
动态翻译	无需解释器和运行环境的支持，无需用户参与，利用动态信息发掘优化	翻译的代码效率不如静态翻译高，对目标机器有额外的空间开销。

1.2.1 系统级翻译精确异常处理

对于系统级的翻译器，要求在发生异常之后，能够把异常之前的寄存器以及内存内容恢复出来。

Daisy[9]系统是IBM公司于1999年开发的，它是用于仿真现存体系结构的二进制翻译系统[10]，以使旧体系结构上现存的软件(包括操作系统内核)可以在超常指令字(VLIW)体系结构下运行。VLIW结构设计简单而且指令发射率高，但却与现在的软件不兼容，使VLIW得不到真正的使用。Daisy正是要解决这个问题。每当一段新的指令第一次执行，这些代码就被驻留在只读内存中的虚拟机监视器翻译成VLIW原语，并行化且保存在旧的体系结构看不到的内存中，以后在执行这段代码时就无需翻译。Daisy实现了对于PowerPC体系结构的动态并行化算法，以较低的翻译开销，获得了较高的指令级并行度。另外Daisy还采用了一定的方法，动态解决了包括自代码修改，精确中断，内存一致性问题。Daisy中解决精确异常处理的方法[11]是在将结果放入寄存器之前先放入非物理(non-architected)的寄存器中，然后在原来的顺序点(即指令调度之前该指令的位置)插入拷贝指令把非物理的寄存器放到物理的寄存器中。当然这种方法要求内存存储操作是不能调度的。这样在产生异常时，定位到哪一条指令之后，就可以从寄存器直接得到POWERPC的状态。

Transmeta公司推出的Crusoe[12,13]处理系统是一种动态二进制翻译器与硬件协同设计的系统，它利用动态二进制翻译器CMS(code morphing soft-ware)来简化底层硬件

的设计，以达到降低处理器能耗的目的。CMS的源体系结构为X86指令集体系结构，目标机为Transmeta开发的VLIW处理器，与IA-32 EL 不同，Crusoe上的一切软件都运行在CMS之上，包括操作系统，BIOS，设备驱动程序等等。与IA-32 EL相比,CMS与硬件结合得更紧密，可以获得更多的硬件支持；同时，运行操作系统等程序也给CMS带来了新的问题和挑战。Crusoe利用硬件的推测执行能力来支持精确异常。在Crusoe处理器中,对每一个映射寄存器都配有一个临时寄存器,同时还有一个受控内存缓存区。在执行指令时,它更新临时寄存器或者写入受控内存缓存区。当它执行完一个翻译码块之后,它会执行一个提交操作，提交指令将这些临时寄存器上的值更新至映射寄存器,将受控内存缓存区的值写入映射内存。如果在该指令执行之前,有一条指令发生了异常,或者有外来中断需要处理,那么这些临时寄存器和受控内存缓存区中的值将被作废,将状态恢复至上一条提交指令后将控制转移至CMS程序。在发生异常的情况下,Crusoe会从上一个提交点开始逐条指令解释执行直到发生异常的指令,然后调用X86异常处理程序。有了这种硬件支持,精确异常对该翻译码块的调度基本上不再构成约束。

BOA 动态翻译器系统[14,15]的目标是简化硬件，通过结合二进制翻译和动态优化，填补 PowerPC RISC 指令集和更简单的硬件原语之间的语义差别。BOA 系统关心的不是每条指令的周期数目（CPI）最小化，而是希望通过简化的硬件指令，可以极大地提高处理器频率。BOA 系统动态地解决了二进制翻译中存在的精确中断和自修改代码问题，并且通过在解释过程中收集 profiling 信息，帮助生成热路径，将一条热路径上的代码放于内存连续位置，提高了指令 cache 命中率，有助于迅速取址。BOA 还进行了优化调度，从而提高了程序并行性，并解决了由调度产生的访存一致性问题。BOA 为取得最大的指令集并行（ILP）调度，同时进行乱序调度、优化和寄存器分配。BOA[14]在 trace 转化的边界保持精确的检测点（precise checkpoint），当发生异常的时候，可以回溯到最近的一个检测点。在 trace 中的指令为了支持投机执行而被乱序调度，寄存器重命名。对于存储操作还是需要按原来的程序执行顺序执行，只是标记它们为未决定的（pending）,如果产生异常就可以撤销对内存的操作。只有当 trace 正常退出的时候（无论是 side exit 还是 end-of-trance exit）,所有的 PowerＰＣ寄存器提交到检测点寄存器，未决定的内存存储被标记为确定的,然后执行下一个 trace。这时，在基本块的边界，每一个映射 PowerPC 寄存器的 BOA 的寄存器都是真正的值。

1.2.2 应用程序级精确异常处理

应用程序级翻译的整体效率高于系统级翻译，所以大多数商业实用的系统都是应用级的。应用程序级翻译，按照对库函数是否翻译，有两种翻译层面：对库函数的包装和对系统调用的包装。如果翻译器是对库函数进行包装，那么在翻译中碰到库函数，就不对库函数本身进行翻译了，而是用相同功能的本地库函数进行替代。翻译器对系统调用的包装，则比库函数包装更加底层，即遇到库函数时，将进行库函数的翻译，只有碰到了系统调用，才用功能相同的系统调用进行模拟程序级的。Intel公司的IA32 EL, Transitive公司的Quick Transit 都是对于系统调用的包装，所以它们只有当硬件发生异常，给操作系统发出 signal 时才启动异常处理。

Intel公司2003年开发的IA-32 Execution Layer(EL)[16-18]软件，通过软件的方法在IPF(Itanium Process Family)上执行IA32的应用程序，实现兼容，从而简化硬件的复杂度。IA-32 EL是一个应用程序级的翻译器，它运行在本地的64位OS之上，能够支持windows和Linux系统。IA-32 EL 利用优化调度后的翻译码中的“ 提交点 ”和“ 异常点 ”支持精确异常。异常点是指翻译码中可能发生异常的指令。每一个提交点可以对应多个异常点。与异常点不同，提交点不是某一条具体的Itanium指令，而是在翻译码中一个无形的“ 障碍提交点 ”。对应于源机器码中一条指令以及该源指令执行前的源机器状态。在翻译码中，映射寄存器状态的更新在提交点这一点提交。所以，在提交点之后，在下一个提交点到来之前，如果存在对这些寄存器的更新，需要对这些寄存器进行备份。这样，在与该提交点对应的某异常点发生异常时，IA-32 EL 可以从这些映射寄存器或专用保存寄存器中恢复出从该提交点所对应的源码状态。故此，本提交点与下一个提交点之间的指令可以比较自由地调度。为了使翻译码有更大的调度空间，IA-32 EL 将尽可能多的异常点对应于一个提交点。第1个提交点在块首设置，然后每当遇见一个无法调度的异常点(写内存和跳转指令)，或者当保存寄存器被用尽无法保证提交点完整的源寄存器状态之时设置一个新的提交点。IA-32 EL 利用纯软件的方法巧妙地实现了优化调度和精确异常[19]。IA32 EL 为了减小异常处理的开销，采用了还原检测点机制，即在每隔一段间隔，插入一个机器的副本状态，一旦发生异常，就从最近的还原点把机器状态还原出来，从还原点开始重新执行。在这种机制下，检测还原点的插入关系到翻译性能。如果检测点间隔太短，那么加入的额外代码太多；间隔太长的话，发生异常之后的代价就会大了

DEC公司的FX!32[20-22]系统在Alpha处理器上实现了IA-32体系结构。FX! 32是一

个解释器与静态二进制翻译器相结合的系统。严格地说，它不是一个动态二进制翻译系统。它在程序第一次运行时对程序进行解释执行，同时收集程序的动态运行信息 在程序第一次运行完以后，由系统中另一个后台进程对运行过的源程序片段进行静态二进制翻译。在程序被多次运行的情况下，后台程序会对曾经翻译过的代码进行进一步的优化翻译。FX!32是在 windows NT 平台下从 x86 到 alpha 的应用程序级翻译器，它对windows的 API 以及 com 组件进行包装，并通过支持SHE (structured handling exception)来对异常进行处理。如图1.1所示。

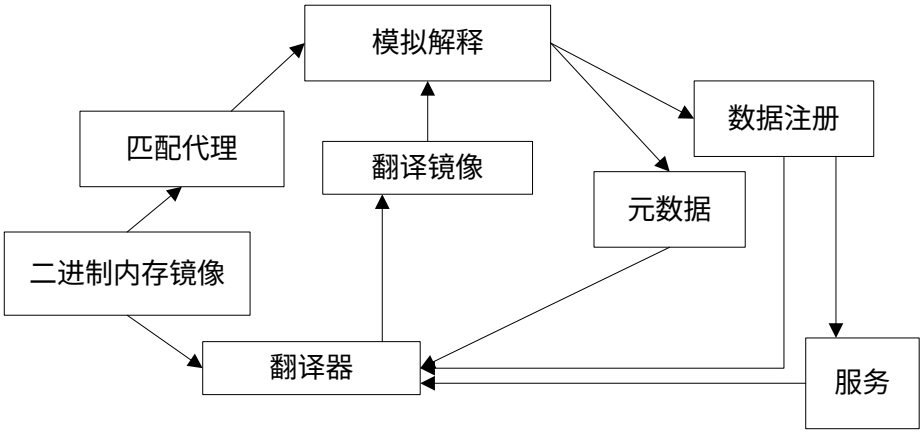

图 1.1 FX! 32 主要功能模块图

QEMU是一款使用动态可移植翻译技术的机器模拟器[23-25]。它在(x86, PowerPC, ARM, SPARC, Alpha和MIPS)平台上模拟了许多目标CPU(x86, PowerPC, ARM和SPARC)。QEMU支持全系统模拟同时也支持Linux用户模式模拟，允许一种被目标机编译后的进程在另一种CPU上执行。

QEMU能在不需要做任何修改的目标操作系统上运行并且它集成一个Linux 特有的用户模式模拟。这种运行模式可以用来测试交叉编译的结果或者可以用于去测试CPU 模拟器而不需要去启动一个完全的虚拟机。

QEMU由控制核心，解释器，翻译器，编译器和翻译缓存等几部分组成。QEMU实现了对异常的处理，但是它并没有准确的定位到异常的位置。

1.3 研究内容

QEMU 是一款非常流行的多源多目标 CPU 模拟器。但是 QEMU 在支持精确异常方面仍存在不足，具体表现在当异常发生时，无法完全恢复到异常发生前 CPU 的精确状态，无法定位异常指令的具体位置。

针对 QEMU 支持精确异常不足的缺点，设计与实现了改进 QEMU 精确异常处理

机制的异常处理系统。

1. 异常处理系统给出了一种对异常进行处理的机制，并将异常分为可预测性异常和不可预测性异常[26]。异常处理系统使用设置异常点法对可预测性异常处理，使用计算异常点法对不可预测性异常进行处理，最终实现对 QEMU 用户模式下异常指令的准确定位。

2. 集成 QEMU 远程调试功能，当与 GDB 调试器连接时，响应 GDB 命令请求，并实现调试过程中状态监控。当用户在执行过程中出现异常时，以 B/S 方式显示异常处 CPU 状态，该功能具有与用户交互强和用户操作简单的特点，因此具备很好的实用价值。

1.4 章节安排

本文分为五章，内容安排如下：

第 1 章概述了课题的选题背景与意义，介绍了二进制翻译系统的发展状况以及相关技术的研究现状，然后列举比较了国内外几个具有代表性的二进制翻译系统并介绍了这些二进制翻译器处理精确异常的方法，最后说明了本文的研究重点。

第 2 章介绍首先介绍了精确异常处理系统 PEHS 的系统结构，异常处理系统是 PEHS 的核心子系统，它实现了改进 QEMU 精确异常处理的机制，实现对异常指令准确定位的功能。同时第二章也将对 PEHS 包括其层次结构和各个组成模块的主要功能以及各模块工作流程进行介绍，最后对异常处理系统各子模块进行了介绍。

第 3 章首先对异常特性进行描述，其次介绍异常处理机制并对其中一些关键技术进行了详细的讲解。

第 4 章详细讲解异常处理机制的实现以及 GDB(GNU 调试器)远程调试应用，详细讲解设置异常处理模块与计算异常处理模块的实现过程。最后介绍了 GDB 远程调试的相关知识，并详尽讲解了 PEHS 监控 GDB 的整体流程以及具体实现过程。

第 5 章图解说明异常处理系统的功能测试，并对测试结果进行分析。实验表明异常处理系统能准确定位异常指令。同时在检测到异常并进行异常指令定位方面，随着指令数的增加设置异常点法执行时间也在增加，而由于翻译器初始化以及指令匹配执行需要消耗时间，所以计算异常点法总的执行时间并不随指令数的增加而增加。在性能测试方面，当异常发生时异常处理系统提供了比 GDB 更全面更准确的信息，对提高程序员调试程序的效率有很好的帮助。

第 6 章对全文进行总结并展望未来工作。最后是致谢和参考文献。

2 PEHS 中异常处理机制的设计

异常处理机制是本文研究的重点和核心，而异常处理系统作为精确异常处理系统(Precise Exception Handling System)PEHS 的子系统实现了这一机制，完成对异常的处理并实现异常指令的精确定位。PEHS 集成了 QEMU 远程调试功能，实现对远程调试过程中调试状态的监控，增强了用户与 PEHS 平台的交互性。因此在本章中，首先介绍 PEHS 的总体结构并对 PEHS 各个模块以及模块间的相互关系进行阐述，最后讲解异常处理系统的组成部分以及与动态二进制翻译器的交互关系。

2.1 PEHS 概述

PEHS 是一个程序异常检测平台，其中子系统异常处理系统针对 QEMU 无法定位异常指令位置的缺点，实现对异常指令的准确定位，同时 PEHS 也集成了远程调试功能，实现对远程调试状态的监控，当程序在运行过程中产生异常时，系统将返回程序发生异常处异常指令的位置，异常类型，相应通用寄存器以及标志寄存器的值。当程序员在调试程序时，这些信息有助于提高他们调试程序的效率，因此具有很好的使用价值。

PEHS 处理的对象是 Linux 下可执行 ELF 文件。用户可以提交 ELF 文件如果程序需要参数，用户必须正确提交程序使用时所需的参数。用户也可以将 C 源程序通过 FTP 方式提交给 PEHS，通过 PEHS 所在机器的 GCC 编译器对其编译执行生成可执行的二进制可执行文件，然后通过平台进行提交执行。平台在执行过程中，给出程序执行所消耗的时间。用户可能希望增强与平台的交互性，为了提高调试程序的效率，希望知道程序在执行过程中 CPU 状态，而传统的 Linux 调试工具 GDB(GNU Debugger)虽然功能强大，但是由于复杂的 GDB 命令以及界面简陋，很难提高用户的交互性，而 PEHS 由于集成远程调试功能后，实现对这些信息监控与显示，且界面友好，大大增强了与用户的交互性。用户直接通过网页查看需要的内容，简化了用户使用 GDB 命令的操作。

2.2 PEHS 系统架构

系统从逻辑至上而下共分为 3 层如图 2.1 所示。最上层为平台层，它是整个系统和用户交互的接口。中间一层就是请求池，主要负责接受平台层提交下来的请求，

并将交给后续模块进行处理。最底层从左到右是 QEMU 动态二进制翻译器，异常处理模块，通用寄存器存取单元，EFLAGS 寄存器存取单元。 QEMU 需要从请求池接收可执行 ELF 文件及其参数，或者是 GDB 远程连接请求，同时，QEMU 在翻译和执行阶段均要和异常处理系统进行相应信息的交互。这里先给出 PEHS 的整体架构图，然后在后面几节将分别对各模块的功能，执行流程，以及相关技术进行阐明。

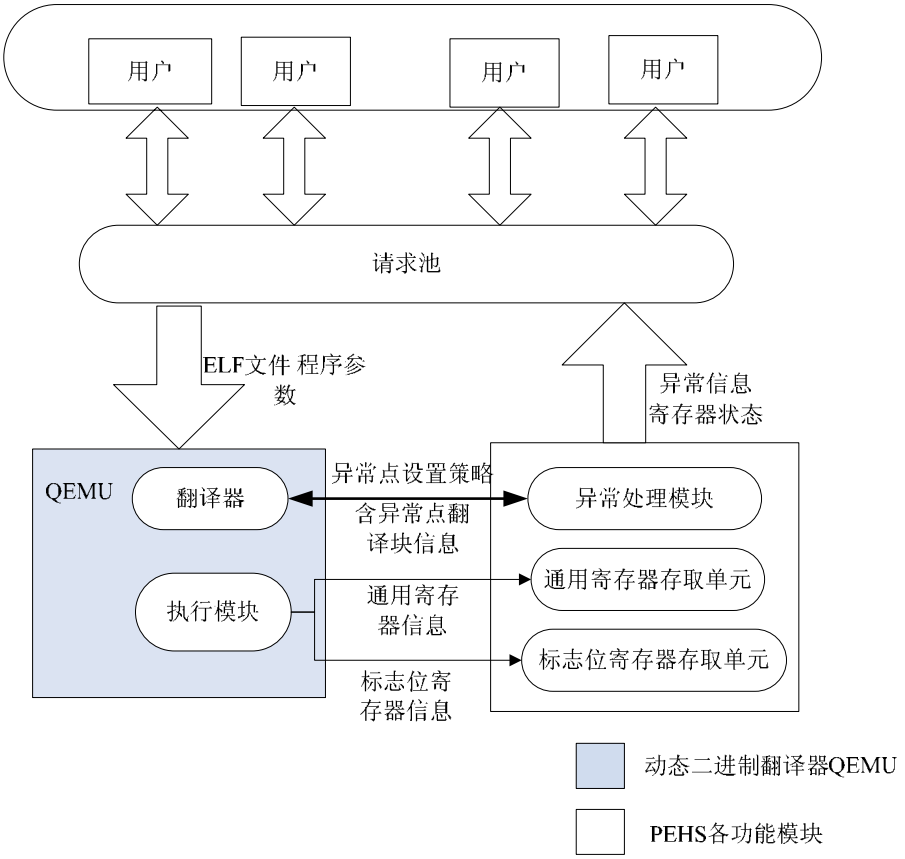

图 2.1 PEHS 整体架构图

首先用户通过平台层提交二进制 ELF 文件以及程序执行过程中所需的参数。在 QEMU 接受到这些相关信息以后，将开始启动翻译器开始执行，如果程序执行正确，将把程序正确执行结果反馈给平台层，否则将通过异常处理模块将程序产生异常的相关信息反馈给平台层。

2.2.1 平台层

平台层作为整个系统与外界的交互，相当于 PEHS 的门户。它上面集成了预处理，任务注册，任务监控和显示信息共 4 个大的功能。

1 预处理是指当用户提交 ELF 文件以及参数时，需要将网页上提交的内容整理

成一个统一的格式然后传递给请求池进行后续处理，同时当后台处理结束以后当有结果生成时，预处理模块也要负责将传递过来的结果进行解析处理才能在 WEB 上显示内容。

2 任务注册就是当用户提交的内容被预处理模块处理并生成统一格式以后，就会提交给请求池，并为其生成一个全局唯一的作业号。

3 监控模块将使用这个作业号对执行请求执行的状态进行查询。任务监控主要负责对请求执行情况的轮询，根据轮询结果判断结果是否产生异常并对控制其中结果进行相应显示。

4 显示模块主要是读取这些通过预处理加工后的数据并将其在平台上显示，要求每隔 3 秒钟刷新一次。可以简化为如图 2.2。

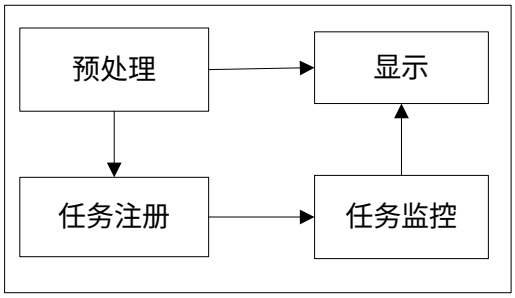

图 2.2 平台层功能图

2.2.2 请求池

请求池位于平台层和执行层之间，一方面请求池要接受来自平台层的各种请求，请求包括执行二进制 ELF 文件请求或者是用户需要提交 GDB 连接请求。同时在另一方面请求池根据提交请求从线程池为提交请求选择空闲工作线程，工作线程在获得请求包含的参数以后，执行请求。请求池在其中扮演了很重要的角色，请求池实际上是一个守护进程。它在每隔 3 秒就将提交作业请求插入请求队列中，然后按照先来先服务策略将刚插入的请求的状态设置为就绪态，同时向线程池提交请求，线程池检查有无可使用工作线程，如果没有可使用工作线程就进入阻塞状态，否则选择空闲工作线程提交作业执行。当程序执行结束以后，如果程序在执行过程中产生异常将会把请求设置为执行异常状态，反之如果程序执行正确，则将请求设置为执行正确状态。平台层的监控模块正是根据这些状态信息在通过显示模块显示这些类型的信息的。平台层的目的就是为了实现用户提交操作与后台处理的透明化，出于安全的考虑，不应该将对后台的处理直接放到平台层，让平台层既要处理用户数据

又要提交给后台程序，现在通过使用请求池将两者分开，使整个系统具有很好的扩展性，可靠性，安全性。这里给出了请求池处理的整个流程如图 2.3。

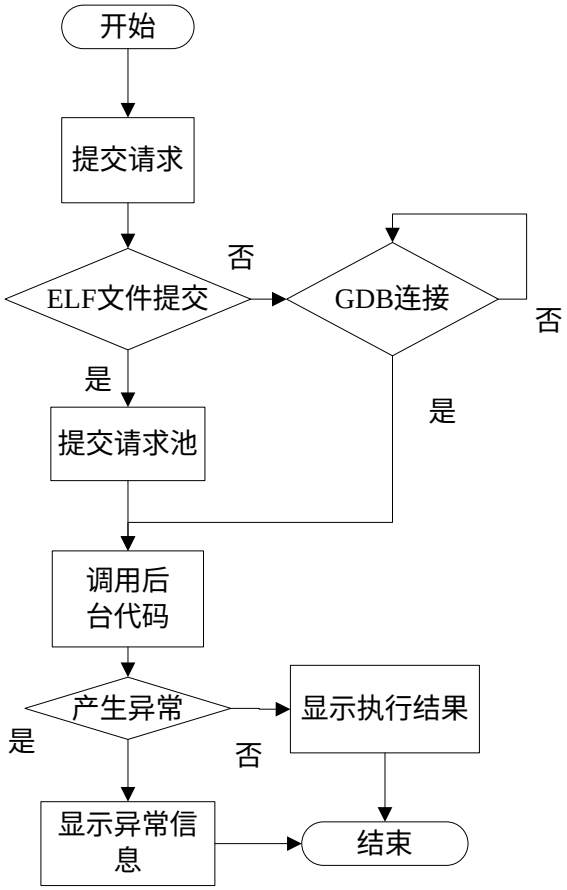

图 2.3 请求池处理流程

2.2.3 通用寄存器存储单元

通用寄存器存储单元是异常处理模块的存储缓存区，它主要负责当翻译器进行翻译执行过程时，对翻译器修改的通用寄存器内容的记录收集，以便当异常发生时，由通用寄存器单元把异常前的通用寄存器的状态返回给平台模块。通用寄存器的工作流程具体说明如下：

(1) 系统控制核心根据当前PC值在Cache中查找是否有已经翻译好的翻译块，如果存在跳到步骤(3)，否则继续。

(2) 调用翻译器，执行相应的指令翻译生成翻译块。

(3) 找到翻译块，并执行翻译块，把寄存器在执行过程的值存放在通用寄存器存储单元中。

2.2.4 EFLAGS 存储单元

EFLAGS存储单元作为异常处理模块的存储缓存区域，主要负责当翻译器进行翻译执行过程时，对EFLAGS寄存器修改时，缓存器将记录下相应修改标志位的状态值。EFLAGS状态寄存器的工作流程具体说明如下：

(1) 执行翻译块,如果未执行完，则继续(2)，否则转到(3)。

(2) 如果在执行过程中，EFLAGS 某个标志位状态发生改变，则把改变的位存放在 EFLAGS 状态寄存器里面。否则状态没发生改变，转到(1)。

(3)执行结束。

2.3 异常处理机制设计思想

2.3.1 异常处理子系统整体设计

异常处理系统是整个 PEHS 系统中负责处理异常事件的子系统。它主要完成以下功能。

1. 根据异常原因，在指令翻译阶段对翻译块中异常指令进行设置，并记录异常点相关信息。

2. 接受执行阶段时产生的异常信号，并根据具体的异常信息，将异常出现以前通用寄存器以及 EFLAGS 寄存器状态回送给平台。

3. 如果异常无法定位，通过宿主机的指令 pc 计算相应的源机器相应异常的 pc 值。

异常处理系统是本研究的重点，它实现了异常处理机制，能够对 QEMU 用户模式异常类型指令的定位。为了能够将异常处理系统各模块关系进行更好更详细地描述，在这里将异常处理系统进一步细化为若干子模块，其中包括设置异常点模块，计算异常点模块，TB 信息队列，异常表等一系列子功能模块。里面详细的介绍了异常处理模块相互交互的关系图，如图 2.4 所示图。并对其中所有子模块进行了详尽的介绍。

1.异常表。完成动态翻译器执行过程中信号的接受，当接受到异常信号时，判断异常信号类型，如果接受到的异常信号是 page-fault 内存访问出错异常，则将异常处理交给计算异常点模块进行处理，如果不是 page-fault 异常处理模块则将异常处理，交给设置异常点模块进行处理。

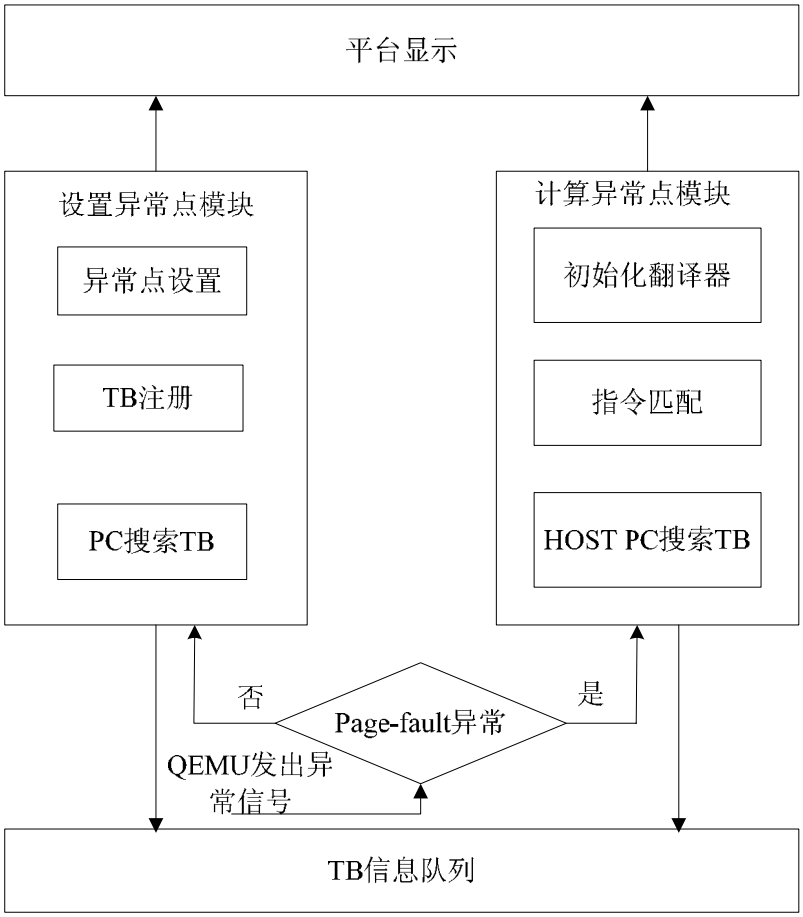

图 2.4 异常处理模块各子模块交互图

2.异常点设置。把异常分为可预测异常和不可预测异常。可预测异常是指在该指令执行前便可预测的一类异常，比如说浮点溢出异常，如果判断它的 CF 标志位为 1，这说明发生了浮点溢出，因此可以在程序执行之前就记录可能出现异常点位置。通过设置异常点完成此功能。

3.TB 注册。动态二进制翻译块都是以块为单位进行翻译执行，而块中包含的块起始地址，块大小等信息都是一些非常重要的信息。因此在每次动态二进制翻译块结束翻译以后，都应该将这些相应的翻译块信息进行注册存放在 TB 信息队列中。

4.PC 搜索 TB。当程序在执行过程中触发了确定性异常以后，采用设置异常点法获得了异常指令的 PC 值，而平台还需要具体 PC 所在翻译块中的信息，因此使用异常指令 PC 值在 TB 信息队列中检索包含 pc 的翻译块。

5.host pc 搜索 TB。当异常信号是 page-fault 异常时，动态翻译器将向异常处理模块发出宿主机产生的异常指令并不是目标机的异常指令，因此需要通过计算异常点法找到相应的异常位置，计算异常点法为三个阶段，host pc 搜索是其中的第一个

阶段，它将使用顺序查找的方式在表中进行查找，找到包含宿主机指令的翻译块。

6.初始化翻译器。当通过 host pc 搜索 TB 找到包含宿主机指令的翻译块时，需要利用该翻译块去初始化翻译器，目的是为得到一些翻译过程中的信息，为后面指令匹配算法做好准备。

7.指令匹配。指令匹配是指计算异常点法的第三个阶段当完成初始化翻译器后得到了一些重要信息以后，这些信息包括微操作表，以及每个微操作对应字节映射表，在翻译块中目标指令序列表，指令匹配将使用这些信息查找相信的微操作位置，然后通过该微操作并根据目标指令序列表计算出目标机的异常指令。

8.TB 信息队列。TB 信息队列存放了翻译过程中每次生成的翻译块信息，TB 信息队列为 pc 搜索 TB 模块和 host pc 搜索 TB 模块提供检索 TB 信息的物质基础。

2.3.2 异常处理系统与动态二进制翻译器交互关系

这里给出异常处理模块与动态翻译器的交互图 2.5。

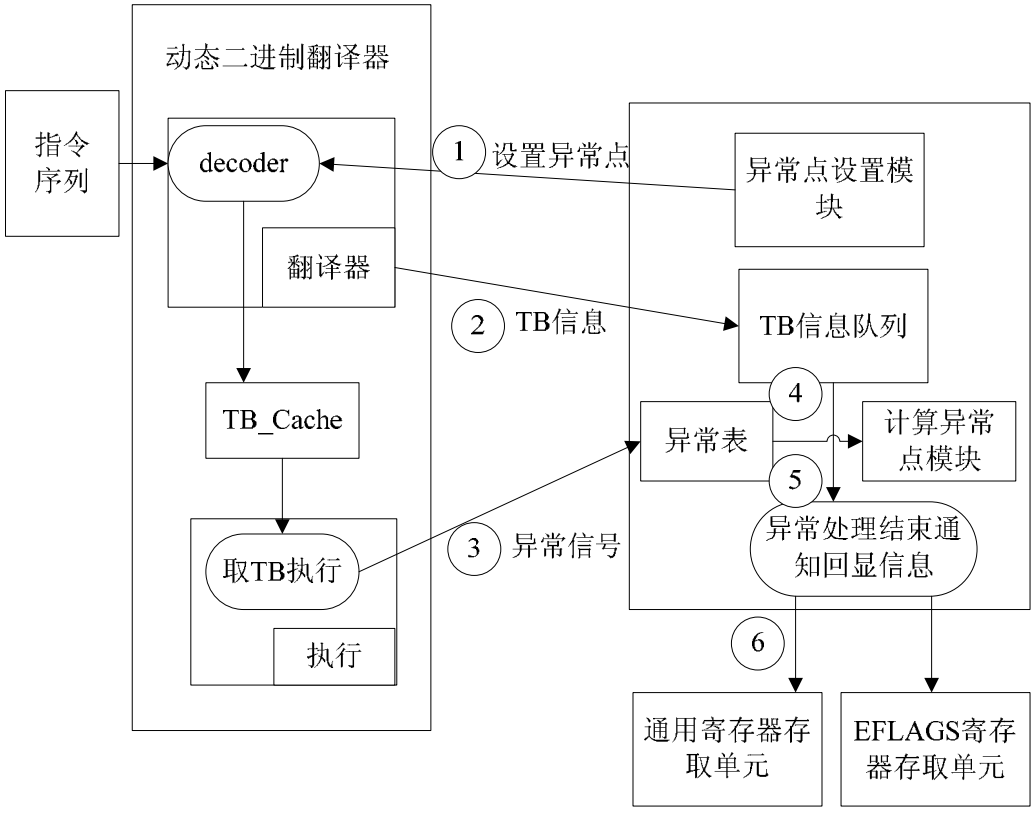

图2.5 异常处理系统与动态翻译器交互图

异常处理模块的具体流程说明如下：

(1) 在指令翻译译码阶段，获得异常指令信息，通过异常处理系统在可能出现的

异常指令处设置异常点，当翻译执行结束以后将形成的翻译块的信息传递给异常处理系统。

(2) 在执行宿主机代码时若遇见异常发生时，将发出异常信号给异常处理系统，系统通过信号匹配表查看如果异常类型如果不是page-fault异常转到(5)，否则执行(4)。

(3) 查看异常处理系统信息，返回异常指令位置，以及相应TB信息，跳到(5)。

(4) 如果异常类型为page-fault，调用宿主到目标机地址映射模块得到原机器相关信息，继续往下执行。

(5) 通知通用寄存器存储单元和 EFLAGS 存储单元将存放信息传送给平台模块用于显示，执行结束。

2.4 系统特色

PEHS与QEMU相比，在其基础上添加了异常处理子系统，完成了指令在翻译块中异常指令的精确定位，并实现与GDB的联合调试，带来的好处如下：

1. 异常指令精确定位

QEMU是一款动态可移植的二进制翻译器，它是以翻译块为执行单位，而现在实现在翻译块中异常指令的定位，迅速寻找异常指令的位置，相比较现在很多的调式器如GDB通过设置多次断点方式寻找异常位置来讲效率提高不少，提高了程序员查错以及分析错误的效率。

2. 实现平台与GDB联合调试

GDB(GNU Debugger)是Linux下一个功能十分强大的debug调试器，主要面向Linux下C/C++ JAVA程序员，但是GDB本身缺乏良好的用户交互界面，并且命令多而复杂，PEHS的开发正好在这方面弥补了GDB的不足，为用户提供了基于WEB的良好界面，并支持单步跟踪，为用户在断点处提供相应通用寄存器以及EFLAGS状态寄存器的正确信息，避免程序员记住复杂的GDB命令。

2.5 小结

本章首先对 PEHS 进行了简单概述并对其总体架构进行介绍，然后分析了系统各个模块的功能以及工作流程，并对其中一些相关技术进行了详细分析，最后讲解异常处理模块的组成部分以及与动态二进制翻译器的交互关系以及具体流程。

PEHS主要由平台模块，请求池，异常处理模块，通用寄存器存取单元，EFLAGS寄存器存取单元五个模块组成。平台模块主要负责用户请求的提交以及处理状态的回显；请求池主要负责多任务请求的处理调用系统后台程序；异常处理系统主要用于根据不同异常情况对异常点进行设置，对page-fault这种无法定位的异常通过宿主机指令，求得目标机异常的相关信息。通用寄存器存取单元主要负责在执行过程中收集寄存器的变化信息。EFLAGS寄存器存取单元主要在执行过程中收集标志位信息。异常处理系统是整个PEHS系统的核心模块，而其它模块又为异常处理系统的实现提供了帮助。

3 异常处理机制的关键技术

异常处理是二进制翻译中不可回避的问题，也是影响翻译性能的一个重要因素。异常处理，通常需要回滚[27-29]机制，即在发生异常之后，能够把在产生异常之前的机器状态还原回去，机器状态主要指CPU的状态(包括通用寄存器以及一些控制、状态寄存器)以及内存的状态。本章根据异常类型和原因不同，将异常分为可预测性异常和不可预测性异常，可预测性异常是指可以在异常未发生时就定位可能异常位置，通过使用设置异常点法进行处理，不可预测性异常是指在异常未发生前无法确定异常位置，只有到异常发生时才能确定异常位置，通过使用计算异常点法进行处理。本章首先介绍异常特性进行描述，其次介绍异常处理机制，并详细讲解其中一些关键技术。

3.1 异常特性描述

异常是一种异步事件，是当处理器在执行一条指令时检测到一个或多于一个的未定义事件。异常分为3大类，fault，trap，abort[30]。

fault异常可以被纠正，一旦纠正，允许程序重新启动，而不失程序执行过程中的一致性。

trap异常当遇见trap指令时异常被报告，但是trap指令允许程序继续去执行不会失去程序在执行过程中的一致性。

abort异常是程序在执行过程中无法定位程序出错的原因，也无法使程序重新启动。程序重新执行依然会导致程序异常出错。

异常处理系统一共处理了7类异常，除法异常，DEBUG异常，断点异常，浮点溢出异常，边界溢出异常，一般性保护异常，页面出错异常。其中除法异常，边界溢出异常，一般性保护异常，页面出错异常属于错误，而DEBUG异常，断点异常，浮点异常属于trap。

3.1.1 异常检测依据

精确异常处理要经历异常检测与异常处理两个阶段，异常检测阶段主要是指当异常发生时，根据异常产生的原因，发出异常信号，这里给出异常处理系统能够处理的七类异常并对这七类异常产生原因进行解释。

1. 除法异常就是作为DIV或IDIV操作的指令的操作数为0，导致结果不能以数值的形势表示而引起的一类异常。

2. Debug异常指暗示由DEBUG调试器设置的一个或多个调试异常的条件被检测，而触发相应异常。

3. 断点异常是一个breakpoint指令被执行，导致一个断点陷入被产生。

4. 浮点溢出异常暗示当一个INTO指令在执行时，这INTO指令检查EFLAGS标志位状态寄存器的OF位的状态。如果OF位被设置，一个OVERFLOW陷入将会产生。许多算术指令(如ADD和SUB)执行符号算术运算和无符号算术运算。这些指令设置OF和CF位在EFLAGS寄存器中,用于暗示无符号数溢出和有符号数溢出。一旦异常测试条件被检测出，异常发生。

5. 边界溢出异常用于检测数组索引是否在数组范围之类，如果超过了数组的上界和下界就报告异常发生。

6. 一般性保护异常是指处理器检测到一些保护性违规条件这里称为“一般性保护违规”。这些条件导致异常产生，其中包含了很多条件，具体可以参考Intel程序员手册。

7. 页面出错异常是指当操作系统允许分页时，处理器检测到在使用分页翻译机制时翻译从一个线形地址到一个物理地址时产生的异常错误，如没有足够的权利去访问一个特定权限的页时产生异常。

3.2 异常处理机制

3.2.1 二进制翻译中间表示形式

中间语言的应用最初来自于编译器，编译器一般首先将程序源代码转换成一种更适合于优化分析的形式，即在生成目标机器代码前的中间步骤。中间语言的设计有很多经验可以借鉴，甚至可以直接使用很多现有的中间语言。如果使用现有的语言，就需要考虑它对新的使用环境的各种适应性问题——既包括前端的语言（源语言）也包括后端的语言（目标语言），还要考虑实现它所产生的移植代价与重用现有设计和代码所固有的开销之间的权衡。现有的中间语言都针对编译器设计，即源语言为一种高级程序设计语言，目标语言为机器语言。这些中间语言并不完全适合于二进制翻译的应用环境。在传统的静态编译器中，可以使用多种中间语言，在编译过程中从一种中间语言转换至另一种形式，以在不同的阶段适合于不同的需求。但增加

中间步骤的级数也意味着增加额外的编译开销，在静态的编译器中这不是问题，但在动态的运行时系统，如动态二进制翻译器或动态编译器中，运行时的编译开销直接影响了程序的运行时性能，因此一般不值得采用多重的中间形式。Queensland 大学先后研究开发了可变源和目标的静态二进制翻译系统框架（UQBT）[31-33]和动态二进制翻译系统框架（UQDBT）[34-36]，它们均是根据中间表示形式实现了多源多目标的指令翻译[37-39]。

QEMU 作为一款支持多源多目标动态二进制翻译器同样也使用了中间变量表示形式[40]。对于单条指令而言，翻译首先从指令解释器开始。指令解释器负责从整个指令流中将每一条指令区分出来。这个过程中需要完成以下工作：

(1)对指令进行长度判断，语义解释，将其中与寄存器相关的指令转化成对于中间变量的操作指令，将一些复杂指令分解成几个简单操作；

(2)识别出跳转，返回等控制程序流的指令，计算其目标地址，然后直接将目标地址写入解释后的指令中。

(3)对系统调用进行特殊处理，将其传给控制核心，进而提交给本地操作系统处理。

之后的工作交由翻译器处理。对于已经分解的汇编指令流，由翻译器找到其对应的 C 语言程序段，这个程序段已经事先被编译成 x86 上的可执行代码，再加入立即数和跳转目的地址，就将指令转化成了可以在本机执行的代码。

最后由连接器将程序段拼接起来，构成一个完整的基本块。

例如对于以下的 arm 指令：

```
add r1 r1 16 # r1 = r1 + 16
```

首先会在指令解释器中被分解成如下三条指令：

```
movl_T0_r1 # T0 = r1
addl_T0_im 16 # T0 = T0 +16
movl_r1_T0 # r1 = T0
```

随后在翻译器中会找出这三条指令的对应 C 函数：

```
void op_movl_T0_r1(void) {
T0 = env->regs[1];
}
void op_addl_T0_im(void) {
T0 = T0 + ((long)(&_op_param1));
```

```
}
void op_movl _r1_T0 (void) {
env->regs[1] = T0;
}
```

而立即数 16 会被送入控制核心中保存立即数的堆栈中已被运行时使用。最后由连接器将这三个 C 函数对应的微操作码连接起来，这样就完成了对上述指令的翻译。

QEMU 采用这些中间变量表示形式大大降低系统设计难度。同时使用中间变量方式实现多源多目标的指令翻译。在整个翻译的过程中有几个关键的变量，这里描述如下：

1. gen_opc_buf[ ] 是一个数组，它存放了在翻译器翻译过程中所有的中间微操作序列， 该序列就是块翻译后生成的中间表示形式。

2. opc_copy_size[ ] 指出对应每一个微操作，对应宿主机函数需要的字节数。

3. gen_opc_pc[ ] 指出在翻译块中目标主机包含的指令序列。

通过上面的介绍，以翻译块为单位，让翻译器对其进行处理以后，那么在 gen_opc_buf 中将存放此翻译块中所有的中间形式。而 gen_opc_pc 中将存放所有翻译块中目标机包含的指令序列，异常处理机制正是基于 QEMU 上述中间表现形式的思想，设计并实现了设置异常点法和计算异常点法。使用这两种方法分别处理可预测的异常和不可预测异常。

3.2.2 基于翻译器中间表示形式设置异常点法

由于异常分为可预测异常和不可预测异常两种。异常处理模块使用设置异常点法对可预测异常进行处理。而这种处理是建立在 QEMU 中间表示形式基础上的。

这里以除法指令为例，当翻译器对一条指令译码时，首先读取该指令第一个字节，通过读取该字节，判断该指令为除法指令后保存该指令 PC 值，接着判断该除法指令是要以字节为单位的进行除法运算，还是要以字为单位进行除法运算。在这里根据设置异常点方法，添加相应的保存异常指令 PC 的语句。如图 3.1 所示。

通过添加 gen_jmp_im 函数，保存了异常点处指令 PC 值。这里 gen_jmp_im 将把微操作 INDEX_op_movl_eip_im 添加到对应于翻译块的微操作表 gen_opc_buf[ ]中，同时添加了对应宿主机代码将对应异常指令存放在 EIP 寄存器中，env->eip = exception_pc。

```
case 6: /* div */
        switch(ot) {
        case OT_BYTE:
          //添加gen_jmp_im函数用来设置异常点
          gen_jmp_im(pc_start-s->cs_base);
          gen_op_divb_AL_T0();
          break;
        case OT_WORD:
          gen_jmp_im(pc_start - s->cs_base);
          gen_op_divw_AX_T0();
          break;
        case OT_LONG:
          gen_jmp_im(pc_start - s->cs_base);
          gen_op_divl_EAX_T0();
          break;
```

图 3.1 异常点设置

tb表示待翻译的翻译块，tb->pc_ptr表示待翻译块的pc首址，pc_ptr是一个用于指向目标机pc值的指针，它会随着在不断译码过程中不断增加，直到翻译块结束pc_ptr停止增加。Gen_eip_submit( )表示提交可能异常点提交，disas_insn( )用于对指令进行译码，submit_tb_queue( )是向TB信息处理队列提交翻译后的TB信息。

算法 异常设置策略算法

算法输入：与翻译相关的翻译块tb，tb->pc_ptr存放了翻译的pc首址。

步骤1：// 进行初始化

pc_start =tb->pc_start;

pc_ptr=pc_start;

步骤2：//进行译码与设置异常提交点

for *pc_ptr* ***to*** *last instruction in TB*

do

if *pc_ptr* ***in*** *expection_list* // 异常点检测

{

gen_eip_jmp(pc_ptr – pc_start); //设置异常提交

}

pc_ptr = disas_insn(dc,pc_ptr); //对指令进行译码新的pc_ptr等于原先pc_ptr加上指令节数

end do

步骤 3： *submit_tb_queue(tb);* //提交给 TB 信息队列

采用设置异常点法获得了异常指令的PC值，而平台还需要具体PC所在翻译块中的信息，因此必须使用异常指令PC值在TB信息队列中检索包含pc的翻译块。

这里首先给出指令搜索定义

定义 3.1 指令搜索是一个三元组，记作 A=(T,Q,R),其中

1. T,Q 都是有限集合，分别表示 TB 信息块中包含指令的集合与 TB 信息队列中包含 TB 信息块的集合。$T_i=\{p_{ij}\mid 1<j<m\}$，其中 i 表示第 i TB 信息块，p_{ij} 表示第 i TB 信息块第 j 条指令。$Q=\{T_1,T_2,T_3\cdots T_n\}$，表示 TB 信息队列。

2. R 是一个 T,Q 的二元关系，即 $R\subseteq T\times Q$，R 描述是否 $\exists p\in T\cup T\in Q$。$p$ 表示某条指令。

定义 3.2 $\exists p\exists T$ 使得 $p\in T$，其中 $f(p,T)=\{R\mid (p,T)\in T\times Q\}$，则有

当且仅当 $f(p,T)=\varnothing$，则表示没能通过指令 p 在 TB 信息队列中找到相应的 TB 信息块。

当且仅当 $f(p,T)\neq\varnothing$，则表示通过指令 p 在 TB 信息队列中找到相应的 TB 信息块。

PC 搜索 TB 流程具体如下：

(1) 构建 TB 信息队列 $Q=\{T_1,T_2,T_3\cdots T_n\}$，$n$ 表示当前 TB 信息队列中 TB 信息块的个数。

(2) i 初始值为 1，u 表示搜索时使用的指令。

(3) 如果 $i<=n$ 从 TB 信息队列中取出第 i 个 TB 信息块 T_i，否则转到(6)。

(4) 如果 $u\in T_i$，表明找到相应 TB 信息块转到(6),否则执行(5)。

(5) 置 $i=i+1$，转到(3)

(6) 返回获得的 T_i，算法中止。

这里分析 PC 搜索 TB 算法时间复杂度。比较查找的次数设 m 表示 TB 信息队列中所有的 TB 信息块个数，n_i 表示第 i 个 TB 信息块的长度。所以总的比较次数为 $=\sum_{i=1}^{m}n_i$，考虑最坏的情况下设 n 表示所有 TB 信息队列中最长的 TB 信息块。所以 $\sum_{i=1}^{m}n_i<m\times n$,所以最坏情况下时间复杂度为 $O(mn)$。考虑最好的情况下当第一次比较

时就找到包含 PC 指令的 TB 信息块。那么比较次数为 $\sum_{i=1}^{n}1$ 那么时间复杂度为 $O(n)$。

通过使用设置异常点法实现了对可预测性异常的处理，下一节将介绍通过使用计算异常点法对不可预测的异常进行处理。

3.2.3 基于翻译器中间表示形式计算异常点法

当异常的原因可以在执行前就确定时，就可以使用设置异常点法准确定位到异常的位置，但是对于有些异常却并不能根据异常原因来设置，比如 page-fault 异常只有当访问到非法内存单元时才会触发相应的异常，因此无法事先确定异常指令的位置，这里把这种异常称为不可预测异常。异常处理模块通过使用计算异常点法对此类异常进行处理。计算异常点法将寻找异常点方法分为三个阶段，第一个阶段通过宿主机操作系统得到宿主机异常指令，然后利用宿主机 pc 搜索到包含宿主机指令的翻译块信息。算法如下。

算法 宿主机*pc*搜索TB算法

算法输入：宿主机*pc*值：*host_pc*

步骤1：

```
for  TB信息队列中第一个信息块 to TB信息队列中最后一个信息块
do
    host_start_pc = tb->tc-ptr;                    //翻译块宿主机代码首地址
  while     host_start_pc!=tb_end_instruction
   if    host_start_pc==host_pc
         转向步骤2
  else
         host_start_pc++;
 enddo
          if    host_start_pc!=tb_end_instruction
    tb=tb->next_tb;
```

步骤2：返回TB信息，结束搜索。

当通过宿主机 *pc* 搜索 TB 算法得到包含异常宿主指令的翻译块以后，需要利用得到的 TB 去初始化翻译器，目的是要得到对应于该翻译块的微操作序列表 gen_opc_buf，对应每个微操作包含多少个字节的映射表 opc_copy_size 以及对应于该翻译块的目标机指令序列表 gen_opc_pc。这里给出初始化翻译器的示意图如图 3.2

所示，具体流程如下所示.

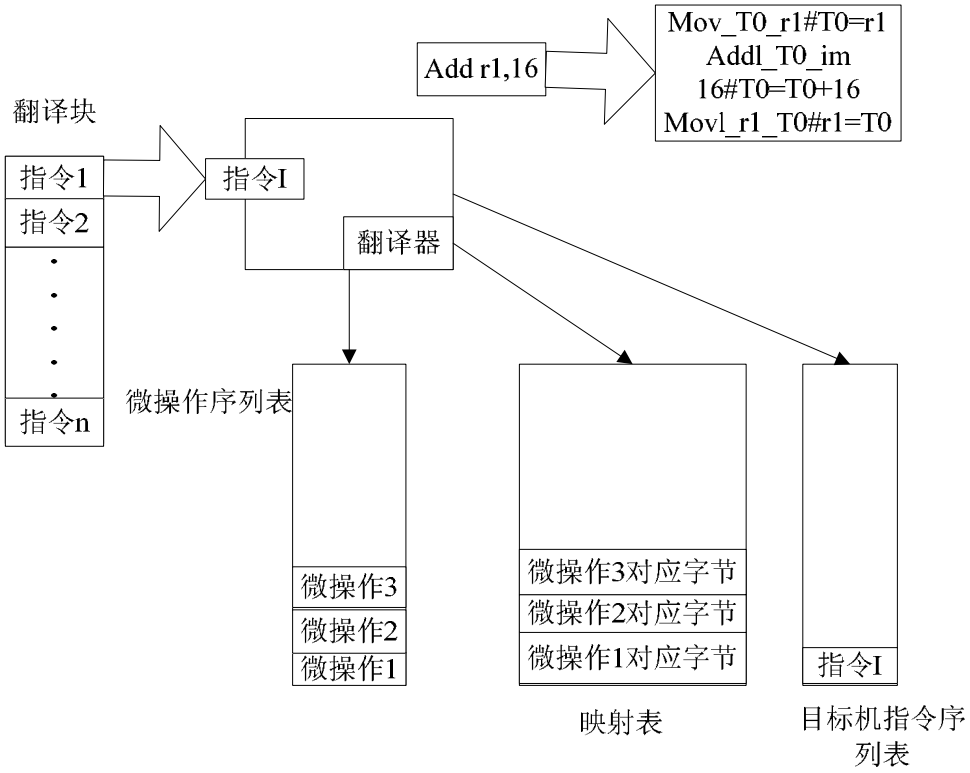

图 3.2 初始化翻译器

（1）翻译器读取翻译块中一条指令。

（2）对指令译码将目标指令分为若干微操作。

（3）将微操作存放入微操作序列表中，将微操作对应字节数信息存放如映射表中，最后将目标机指令存放如对应翻译块的目标机指令序列表中。

（4） 判断是否遇见块结束指令，如果没有遇见块结束指令则跳到(1),否则转到(5)执行。

（5）初始化翻译器工作结束。

3.2.4 指令匹配算法

在计算异常点法中，完成了宿主机指令查找翻译块，然后通过得到的翻译块信息初始化翻译器，这样就得到了对应于该翻译块的微操作序列表 gen_opc_buf，对应每个微操作包含多少个字节的映射表 opc_copy_size 以及对应于该翻译块的目标机指令序列表 gen_opc_pc 这些非常重要的信息。但是还不能得到目标机的异常指令。为了得到求得目标机的异常指令，使用了指令匹配算法。

指令匹配算法的一个基本思想就是先将翻译块宿主机的首地址赋给变量，然后通

过累加每个微操作对应宿主机所占的字节数，如果当累加结果大于宿主机 host pc 值时，记下此时的微操作最后通过查找目标机指令序列表找到发生异常的目标机 pc。如图 3.3 所示。

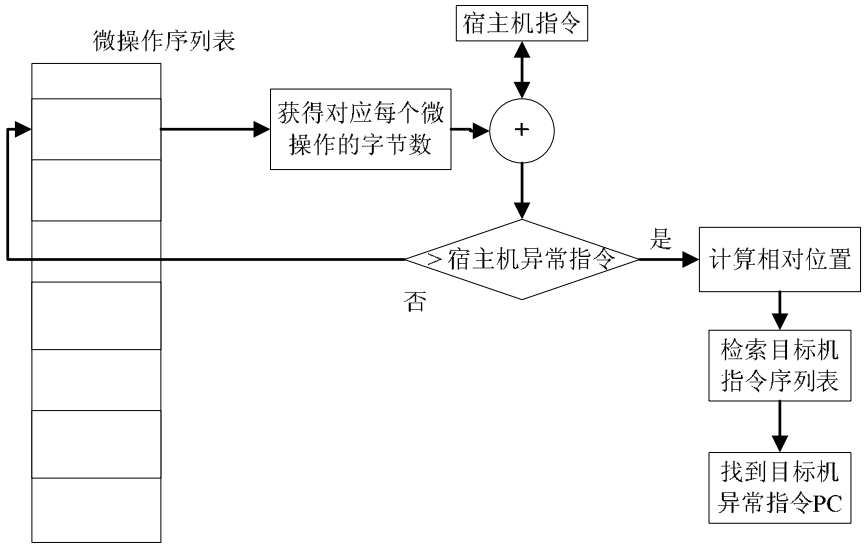

图 3.3 指令匹配算法示意图

这里给出指令匹配算法，必须满足的条件以及算法的具体过程。这里 *host_pc* 表示宿主机异常指令，*base_addr* 表示对应于翻译块宿主机起始地址，*pc_addr* 表示对应翻译块的目标机起始地址，*f(i)*表示微操作，*i* 表示在微操作序列表中的索引值。*g(x)* 表示微操作 *x* 对应宿主机指令需要的字节数，*n* 表示整个微操作序列表中微操作的个数。其中满足 $f(0) = 0, g(0)=0, base\_addr + \sum_{i=0}^{k} g(f(i))$ 表示第 k 条微操作对应宿主机的指令地址。(0<k<*n*)令 S_i 表示第 *i* 条指令对应的微操作个数。P_i 表示目标机指令序列表第 *i* 条指令占用的字节数。

因此给出指令匹配算法的步骤。

(1) 如果 *host_pc* < *base_addr* 表示指令匹配查找失败推出指令查找程序。否则转到(2)。

(2) 从微操作序列表中取出第 k 条微指令 *f(k)*,计算第 k 条微操作所占用的字节数。*g(f(k))*。

(3) 比较 $base\_addr + \sum_{i=0}^{k} g(f(i))$ 是否大于 *host_pc*，如果大于该值，则转到(4),否则 *k=k+1* 转到(2)。

(4) 记下此时 k 的值。

(5) 从目标机指令表中取出第 i 条指令，得到第 i 条指令包含的微操作数为 S_i.

(6) 比较 $S_1+ S_2+ S_3.....+ S_i$ 是否大于 k,如果小于 k,那么 i=i+1 转到(5),否则转到(7)。

(7) 记录下此时的变量 i 的值。

(8) 查找目标机指令序列表，索引值第 i 项存放 i 条指令占用字节数。

(9) 计算出异常指令位置 $exception\_pc = pc\_addr+\sum_{m=1}^{i-1} p_m$

这里对指令匹配算法时间复杂度进行分析。设目标指令序列表长度为 r ,设微操作序列表长度为 n ，当 $base\_addr + \sum_{i=0}^{k} g(f(i)) > host\_pc$ 这是微操作表索引值为 k ($k <= n$),然后比较 $\sum_{i=1}^{m} s_i < k$ 得到比较次数 m ($m <= r$)，最后得到异常指令地址进行累加 $m-1$ 次。所以总的执行次数为 $k+m+m-1=k+2m-1 <= n+2r-1$，所以时间复杂度为 $O(n+r)$ 。

通过指令匹配算法计算得到相应目标机异常指令的位置，然后通过平台层将异常发生时的相关信息显示在网页供用户查阅。

3.3 小结

本章首先介绍了异常处理模块处理的异常类型，并进行了简单的介绍，然后讲解二进制翻译器中间表示形式，分析 QEMU 中间表示形式的具体过程。针对异常原因与特性不同，存在两大类异常可预测异常和不可预测异常。对于可预测异常可以在执行之前，根据产生异常的原因，设置异常点，然后在执行过程中将相关信息，返回给平台模块，对于不可预测的一类异常，如 Page-fault 异常并不能在执行前对其按照设置异常点方法进行处理，因为 Page-fault 异常具有不可预测性，往往是在执行过程中才产生，因此采用了指令匹配算法对异常指令进行精确定位。最后详细讲解和分析了 host pc 搜索 TB 算法 以及指令匹配算法，并对 PC 搜索 TB 算法和指令匹配算法的时间复杂度进行了分析。分析显示 PC 搜索 TB 算法最坏情况下时间复杂度为 $O(mn)$ 。考虑最好的情况下当第一次比较时就找到包含 PC 指令的 TB 信息块。那么比较次数为 $\sum_{i=1}^{n} 1$ 那么时间复杂度为 $O(n)$ 。而指令匹配算法时间复杂度为 $O(n+r)$ 。

4 异常处理机制及其相关技术实现

本章阐述了异常处理机制的具体实现。首先描述了异常处理机制在整个系统中的实现细节。异常分为可预测性异常与不可预测性异常两大类异常。结合设置异常点法与计算异常点法分别设计与实现了设置异常点模块和计算异常点模块，并详细讲解了如何使用 pc 搜索 TB 算法，host pc 搜索 TB 算法和指令匹配算法在异常处理系统中的具体的实现，最后介绍了 GNU 调试器监控的整体流程与具体技术实现。

4.1 异常处理机制实现

4.1.1 设置异常点模块

程序在执行过程中可能因为种种原因而产生异常，由于这些异常的产生原因可以通过分析得出。比如：除法异常如果知道除数为零，即可判断产生除法异常，因此可以对这类可能出现的异常进行提前预测。要实现在翻译块中异常指令的精确定位，首先必须对可能出现的异常指令位置采用一定的策略进行设置，根据不同异常原因不同，设置异常检测点[35-37]，等翻译块翻译完以后还要将其中翻译块的信息全部提交给TB信息队列管理，用于后来异常信号出现时，相关信息的收集以及向平台模块的回显。设置异常点模块包括异常点设置，TB注册，pc搜索 TB共三个模块。

1．设置异常点方法

QEMU 是一款动态二进制翻译器，它包括了翻译模块和执行模块，当 QEMU 控制中心根据目标机指令找到可用的翻译块时，调用相应的翻译块就可以执行。如果不能找到翻译的二进制翻译块时，将对目标机指令进行译码翻译，在翻译过程中 QEMU 为了能达到较好的可移植性，采用了逐条指令翻译的方法，QEMU 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 的技术，省去了函数调用的开销和堆栈的维护开销

根据上面的分析可知 QEMU 是顺序翻译，并以块为单位调度执行。在 QEMU 中从某一次开始，对某一条指令进行译码。这里以除法指令为例，首先将该指令的第一个字节地址保存起来，然后读取除法指令的第二个字节分析其中的操作数，如果发现其中的操作数为零，即除数为零，就可以判定将发生除法异常。因此将刚才保

存的 PC 地址保存下来并且生成相应的代码在 QEMU 中提交保存 PC 的微操作。当程序翻译执行过程中发生异常时，将返回异常处的指令。

2．PC 搜索 TB

PC 搜索 TB 是指当程序在执行过程中触发了可预测性异常以后，采用设置异常点法获得了异常指令的 PC 值，而平台还需要具体 PC 所在翻译块中的信息，因此使用异常指令 PC 值在 TB 信息队列中检索包含 pc 的翻译块。

PC 搜索 TB 的过程是，

(1) 首先从 TB 信息队列中取出第一个翻译块信息，

(2) 然后从翻译块信息中得到翻译块的起始地址以及块长度，因此可以得到翻译块最后一条指令的地址为起始地址加上块地址然后减 1。

(3) 接着将 PC 值与得到的翻译块首址以及最后一条指令进行比较，如果 PC 值落在这两个值的区间范围内就找到相应的翻译块转到(5).

(4) 否则将继续到翻译块信息队列中寻找下一个翻译块，转到(2)。

(5) 执行结束。

2．设置异常点法整体流程图

设置异常点模块实现的整体流程如下所示，其中包括了设置异常点模块实现上的整体流程如图 4.1 所示。

步骤如下：

（1）当控制中心根据目标机的 PC 指令寻找翻译块时，如果翻译块能够找到则转到(6)，否则转到(2)。

（2）翻译模块开始对指令序列进行译码添加微操作，如果发现其中检测到了可能的异常指令时转到(3),否则转到(4)。

（3）提交保存 PC 值的微操作，保存 PC 值。

（4）检测是否遇见跳转指令，如果没有遇见跳转指令则跳到(2),否则转到(5).

（5）生成新的翻译块，并提交给 TB 信息队列。

（6）控制中心执行翻译块，当异常发生时接受异常信号，并得到异常指令执行时的 PC 值，否则转到(1)。

（7）根据 PC 值计算出 PC 所在的翻译块。

（8）将 PC，以及翻译块中信息一起提交给平台进行显示。

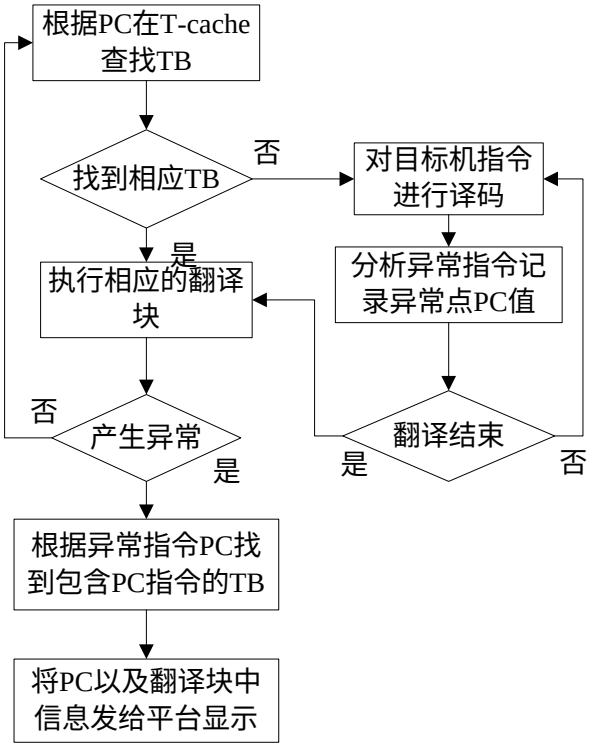

图 4.1 设置异常点流程图

4.1.2 计算异常点模块

异常分为两种，一种是可预测性的异常，另一种是不可预测的异常。可预测性异常由于这些异常的原因都是可以在异常发生前就预测到，具有一定的预见性，比如说除法异常，只要能够判定除数为零，就可以预测到该条指令发生异常。而不需要到程序执行时才去判断。而不可预测异常是指无法直接根据异常原因得到异常指令位置，比如说 page-fault 异常是当非法访问了内存单元时才能触发相应的异常，那么是无法在执行之前就确定位置的。因此需要使用计算异常点模块通过计算方式得到异常指令的位置。

1．计算异常点模块组成

计算异常点模块做为异常处理模块的一个子模块，实现了不可预测异常指令的定位。计算异常点模块一共包含 3 个子模块，分别为 host pc 搜索 TB 模块，初始化翻译器模块，以及指令匹配模块。这三个子模块是计算异常指令时必须经历的三个阶段。这里给出这三个模块的交互图如图 4.2。

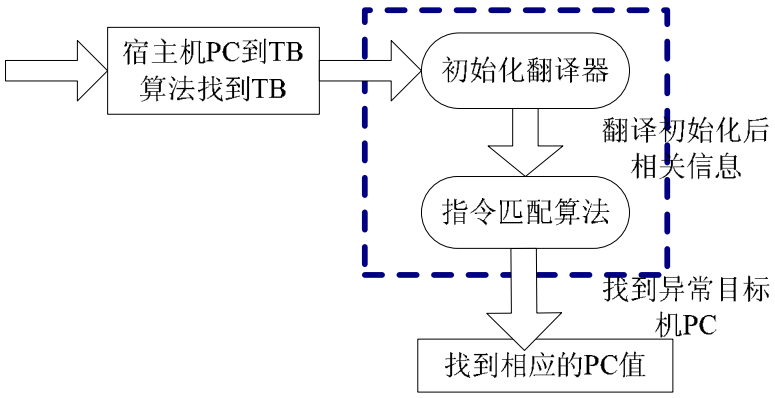

图 4.2 计算异常点子模块交互图

2．计算异常点流程

当遇见不可预测异常时，需要根据计算异常法准确定位到目标机的异常指令。这里给出计算异常点流程步骤。

(1) 接受到异常处理信号，判断如果是 page-fault 异常信号，则转到(2)，否则调用设置异常点模块。

(2) 通过操作系统信号获得宿主机指令,根据 host pc 在 TB 信息队列中查找翻译块 TB 信息。

(3) 在找到翻译块 TB 以后，将翻译块 TB 作为参数去初始化翻译器，目的是为了得到从翻译器译码翻译的过程中得到微指令索引表，以及对应翻译块的目标指令序列表，以及对应每个微操作所需字节的映射表。

(4) 通过所得到的索引表，指令序列表以及映射表计算出相应目标机器的异常指令。

4.2 GDB 远程调试监控

GDB(GNU调试器)是Linux和BSD开发系统的标准组件。是很多专业程序员使用的GNU调试器程序，主要用来调试C/C++应用程序以及查找程序中的错误。PEHS集成了QEMU远程调试功能，将GDB作为平台上的典型应用，实现了GDB与平台的连接，平台负责对GDB的监控，通过使用GDB作为客户端，用户可以键入GDB相关命令，平台根据具体情况进行相应内容的显示。本节首先介绍GDB与远程调试，然后然后介绍QEMU如何实现GDB连接，最后介绍GDB如何实现与平台进行通讯以及其中一些相关细节。

4.2.1GDB 与远程调试

GDB 远程调试是由宿主机 GDB 和目标机调试 stub 共同构成，两者通过串口或 TCP 连接。使用 GDB 标准远程串行协议协同工作，实现对目标机上的系统内核和上层应用的监控和调试功能。调试 stub 是系统中的一段代码，作为宿主机 GDB 和目标机调试程序间的一个媒介而存在。

就目前而言，主要有三种远程调试方法，分别适用于不同场合的调试工作：用 ROM Monitor 调试目标机程序，用 KGDB 调试系统内核和用 GDBSERVER 调试用户空间程序。这三种调试方法的区别主要在于，目标机远程调试 stub 的存在形式不同，而其设计思路和实现方法则是大致相同的。最常用的是调试应用程序。就是采用 GDB+GDBSERVER 的方式进行调试。在很多情况下，用户需要对一个应用程序进行反复调试，特别是复杂的程序，GDBSERVER 在目标系统中运行，GDB 则在宿主机上运行。

在 QEMU 二进制翻译器中，实现了远程调试功能具体实现机制本文将在 4.2.2 节进行介绍，以及详细讲解 PEHS 在此基础上所做的工作。

4.2.2 基于 QEMU 远程调试

QEMU 通过使用 GDB 串行协议建立网络 TCP 连接，实现 GDB 与 QEMU 的连接，因此可以使用常用的 GDB 命令对 QEMU 进行控制，并通过具体的 GDB 命令反馈回相应的结果。本节在这里先介绍 QEMU 与 GDB 进行连接的流程，然后具体介绍 GDB 如何在翻译块中设置端点，因为断点的设置是 PEHS 下一步返回平台正确寄存器的依据和必要条件。具体流程图如下图 4.3 所示。

(1) 通过 QEMU 命令用户输入连接的端口号

(2) 通过网络 socket 建立起 TCP 连接

(3) GDB_SERVER 开始监听 GDB 连接

(4) 如果有 GDB 客户端进行连接，转到(5)，否则转到(3)

(5) 输入需要进行调试的 GDB 命令

(6) 判断程序是否结束，如果程序结束，转到(7)，如果未结束转到(5)

(7) 远程调试结束。

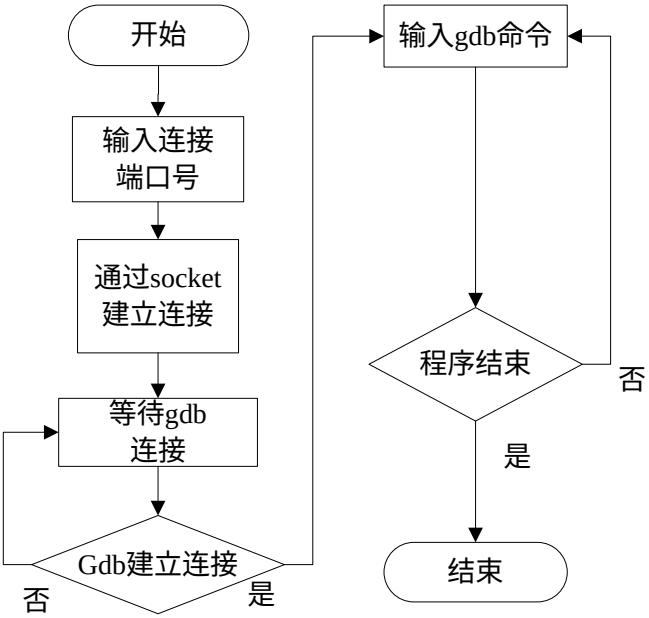

图 4.3 QEMU 与 GDB 连接流程图

以上详细讲解了 QEMU 与 GDB 连接过程的流程图，接下来将详细介绍当用户通过 GDB 客户端执行设置程序断点命令时，QEMU 是如何在翻译块中设置断点的。

首先用户通过 GDB 客户端执行设置断点命令，在设置完以后，通过网络 socket 传递给 QEMU，在 QEMU 中维护了一张断点设置表，每次进行指令翻译时首先检查被翻译的指令是不是存在于断点设置表中如果存在就终止翻译，形成一个翻译块，否则继续翻译直到翻译块翻译结束。其中原理图如下 4.4 所示。

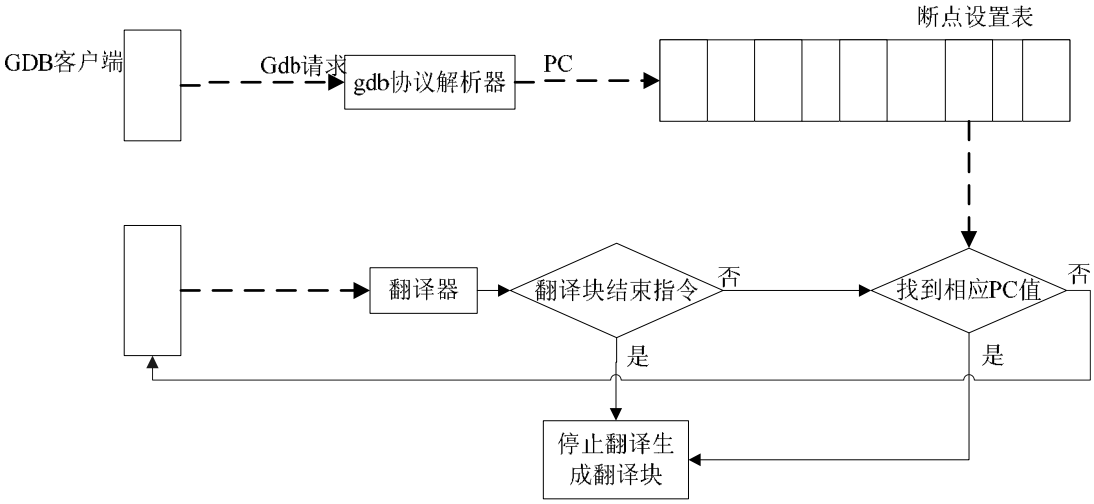

图 4.4 QEMU 设置断点与翻译块关系

4.2.3GDB 远程调试与系统通讯

GDB远程调试与系统通讯具体表现在，由系统平台解析用户从平台提交的请求并根据请求启动QEMU远程调试服务器端，并在平台上给用户提示平台等待连接信息，当有GDB客户端连接，平台将显示连接成功，当用户提交设置断点请求以后，并当程序执行到断点处时，平台将显示异常发生在断点处时的寄存器状态信息。如果程序本身存在其它异常，那么当异常出现时，GDB是无法显示出异常发生前寄存器的状态信息，而平台将给出这一具体寄存器状态信息。具体流程如图4.5所示。

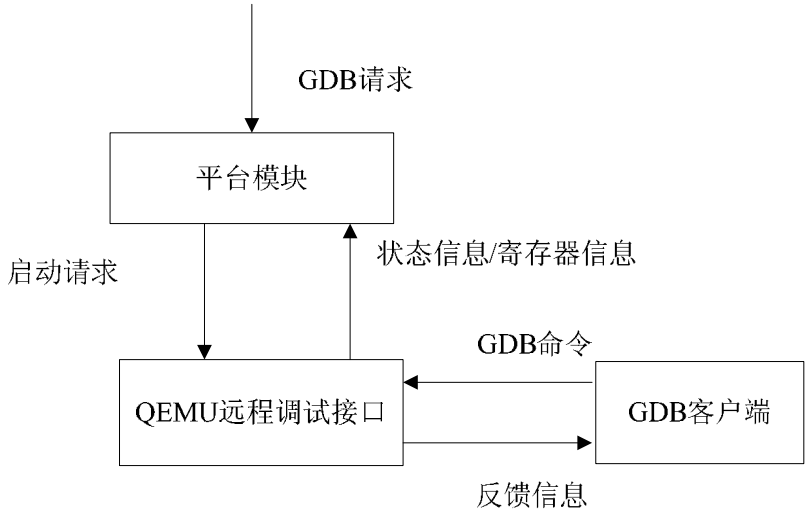

图4.5 平台与GDB远程通讯流程图

平台首先通过WEB接受用户的GDB请求，其中包括提交的GDB处理的ELF文件以及连接的端口号，然后通过平台的预处理模块进行处理后启动QEMU远程调试接口，并等待GDB客户端进行连接。系统平台相当于一个监控模块，会时刻监控QEMU远程调试接口的状态。当启动命令执行，但未和GDB建立连接时，显示等待连接，如果GDB客户建立连接以后，平台将显示建立连接，当用户执行到断点位置时，平台会检测到DEBUG_EXCEPTION异常，并调用通用寄存器存储单元与EFLAGS存储单元内容反馈给平台进行显示。

4.3 小结

本章阐述了异常处理机制的具体实现。首先描述了异常处理机制在整个系统中的实现细节，给出了这个异常处理模块的相互图并对其中子模块的功能进行了一一介绍。结合设置异常点模块和计算异常点模块，介绍了这两大模块的详细流程，并

阐述了 pc 搜索 TB 算法，host pc 搜索 TB 算法和指令匹配算法如何在异常处理系统中得到应用。最后介绍了 GDB 远程调试的相关知识，并详尽讲解了 PEHS 监控 GDB 的整体流程以及具体实现过程。

5 测试与分析

本章主要在上述介绍了系统主要功能与实现技术的基础上，通过实际运行实例，对系统的功能实现，异常处理以及GDB连接等多方面进行全面的测试。最后通过比较计算异常处理模块执行时间对两种异常处理方法进行了分析。并将异常处理系统与GDB进行了比较。

5.1 测试环境

整个测试过程中，系统主要运行在组内的INTEL服务器上，用户只要通过WEB形式即可访问平台。服务器主要软硬件组成如表5.1所示。

表5.1 服务器软硬件列表

CPU	Intel 1.8GHz
内存	512MB
硬盘	80GB
操作系统	RedHat9.0
内核版本	2.4
编译器	GCC3.4.3

个人电脑上采用 IE6.0。控制端与服务器之间的网络连接环境为 100MB 局域网。

5.2 功能测试

功能测试主要处理的对象是 Linux 下可执行的 ELF 文件，要求对存在异常的 ELF 文件进行处理，这里对系统所支持的 7 类异常进行了功能测试，并验证在处理过程中指令是否为异常指令以及验证发生异常时寄存器的正确性，最后对设置异常点法和计算异常点法分别进行了测试。为了验证程序的正确性，在这里选取了一些异常二进制 ELF 文件，在每次测试时均使用不同的异常测试用例。

5.2.1 异常类型检测与指令定位

1．系统测试界面如图5.1所示。

图5.1 系统测试界面

此时用户选择程序执行，选取相应的异常处理程序，同时用户需要提交程序执行过程中的参数。提交以后，程序计算出异常的相关信息。这里给出七类异常处理执行情况。

(1) 除法异常

除法异常就是作为DIV或IDIV操作的指令的操作数为0，导致结果不能以数值的形式表示而引起的一类异常。

图5.2除法异常信息显示

(2) 页面出错异常

页面出错异常是指当操作系统允许分页时，处理器检测到在使用分页翻译机制时翻译从一个线形地址到一个物理地址时产生的异常错误，如没有足够的权利去访问一个特定权限的页时产生异常。

图5.3页面出错异常

(3) 一般性保护异常

一般性保护异常是指处理器检测到一些保护性违规条件这里称为“一般性保护违规”。这些条件导致异常产生，其中包含了很多条件，具体可以参考Intel程序员手册。

图5.4一般保护异常

(4) 浮点溢出异常

浮点溢出异常暗示当一个INTO指令在执行时，这INTO指令检查EFLAGS标志位状态寄存器的OF位的状态。如果OF位被设置，一个OVERFLOW陷入将会产生。许多算术指令(如ADD和SUB)执行符号算术运算和无符号算术运算。这些指令设置OF和CF位在EFLAGS寄存器中,用于暗示无符号数溢出和有符号数溢出。一旦异常测试条件被检测出，异常发生。

图5.5浮点溢出异常

(5) 边界溢出异常

边界溢出异常用于检测数组索引是否在数组范围之类，如果超过了数组的上界和下界就报告异常发生。

图5.6数组越界异常

(6) 断点异常

断点异常是一个breakpoint指令被执行，导致一个断点陷入被产生。

图5.7断点异常

(7) DEBUG异常

Debug异常指暗示由DEBUG调试器设置的一个或多个调试异常的条件被检测，而触发相应异常。

图5.8DEBUG异常

5.2.2 寄存器内容正确性验证

精确异常的要点，就是要保证翻译过程的正确性，当异常出现时我们必须保证程序寄存器各状态的正确性。这里给出了测试方案。

(1) 用户选择 GDB 连接请求，提交需要验证的 ELF 文件，并输入连接平台所需的端口号。

(2) 通过 SSH 登录 192.168.64.62 进入当时 ELF 文件所在目录使用 GDB 与平台连接。

(3) 通过上面已经得到那条异常指令了(如图 5.9) ，然后利用 GDB 设置断点方式进行验证。

图 5.9 除法异常指令 0x0804834f

(1) 在异常指令的前的一条 C 语言设置断点(如 0x8048344)，然后通过 OBJDUMP 查看(如图 5.10)在设置断点和异常点之间有哪些寄存器发生变化（如图 5.11，5.12），观察通过平台数据显示是否与这种预期是否相同通过这种方式进行验证。(如在断点和异常点发现 EAX,ECX,EDX 两个寄存器发生变化，那么平台给用户反馈的信息就是只有两个寄存器内容变化且为 EAX 和 ECX,EDX 其它内容不应该变化)。

```
int main()
{
 8048328:       55                      push   %ebp
 8048329:       89 e5                   mov    %esp,%ebp
 804832b:       83 ec 08                sub    $0x8,%esp
 804832e:       83 e4 f0                and    $0xfffffff0,%esp
 8048331:       b8 00 00 00 00          mov    $0x0,%eax
 8048336:       29 c4                   sub    %eax,%esp
  int i = 5;
 8048338:       c7 45 fc 05 00 00 00    movl   $0x5,0xfffffffc(%ebp)
  i++;
 804833f:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
 8048342:       ff 00                   incl   (%eax)
  int x = i/0;
 8048344:       8b 55 fc                mov    0xfffffffc(%ebp),%edx
 8048347:       89 d0                   mov    %edx,%eax
 8048349:       b9 00 00 00 00          mov    $0x0,%ecx
 804834e:       99                      cltd
 804834f:       f7 f9                   idiv   %ecx
 8048351:       89 45 f8                mov    %eax,0xfffffff8(%ebp)
  printf("the world wonderful!\n");
 8048354:       83 ec 0c                sub    $0xc,%esp
 8048357:       68 18 84 04 08          push   $0x8048418
 804835c:       e8 07 ff ff ff          call   8048268 <_init+0x38>
 8048361:       83 c4 10                add    $0x10,%esp
  return 1;
```

图 5.10 OBJDUMP 显示指令与源码关系

图 5.11 设置断点时各寄存器状态

图 5.12 除法异常产生时各寄存器状态

5.2.3 异常定位方法测试分析

由于异常处理系统能够定位到异常的准确位置与原因，因此与其它系统在准确性方面相比较是最优的。

PEHS采取设计的异常处理系统，在QEMU平台下对系统进行异常处理与测试，而异常处理模块，主要针对两种对异常处理的方式，一种是针对异常可以在执行前

就可以预测的，这里把这种方法称为异常点设置法；另一种方法是针对异常并不能预测，而是在执行过程中才触发的一类异常，这里把这种处理异常的方法称之为计算异常点法。

本次测试，选取了10组数据，因为处理异常是使用异常点设置法处理，所以在这里以除法异常作为处理异常的测试用例。在图5.13横坐标表示异常指令在指令中出现的位置，而纵坐标表示当异常出现在某位置时，从开始执行到检测到异常并退出的时间差，即t(执行时间)=t(执行结束时间)-t(执行开始时间)。结果表明：随着计算机指令的增长，处理异常所耗费的时间基本呈现线性增长。

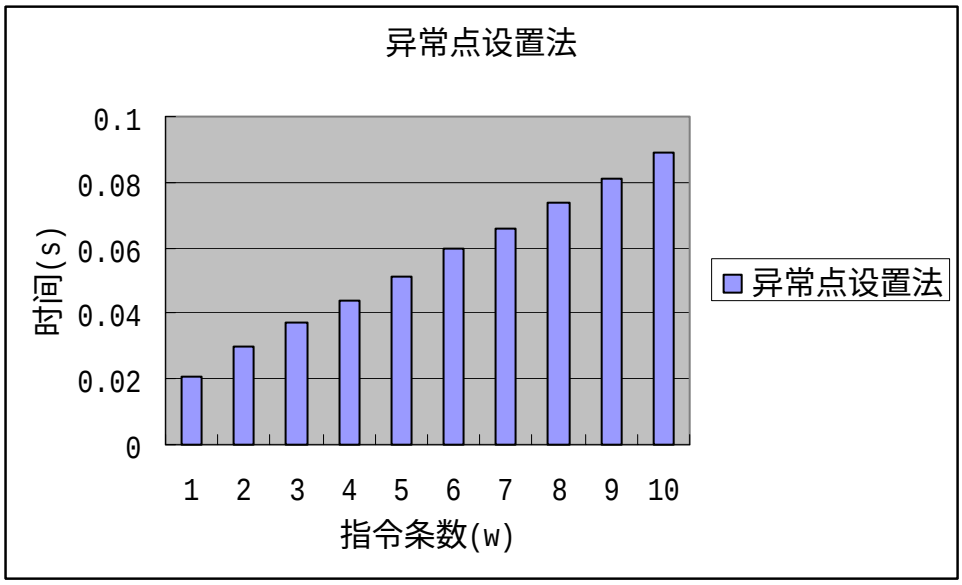

图5.13 异常点设置法性能图

在图5.14本系统测试了计算异常点法，选取的数据与上面完全一样，只是将异常测试用例换成了Page-fault异常，从图中看到，使用计算异常点法并不像设置异常点法那样线形增长，主要原因是因为设置异常点法是在翻译时就已经设置好异常时的具体位置，当异常发生时就返回异常位置即可，在执行过程中并没有计算异常点位置过程，而计算异常点法，需要计算在执行过程中发现异常以后，并要查找TB以及初始化翻译器能一系列活动，找到异常指令位置，而这一过程还涉及到异常指令在原翻译块中位置，块大小不同初始化所耗费的时间也不尽相同，所以这里看见在执行时间上并没有根据指令数的增长，出现线性增长，而是在某些地方出现了一些指令条数增长，反而耗时不如指令较少的情况。

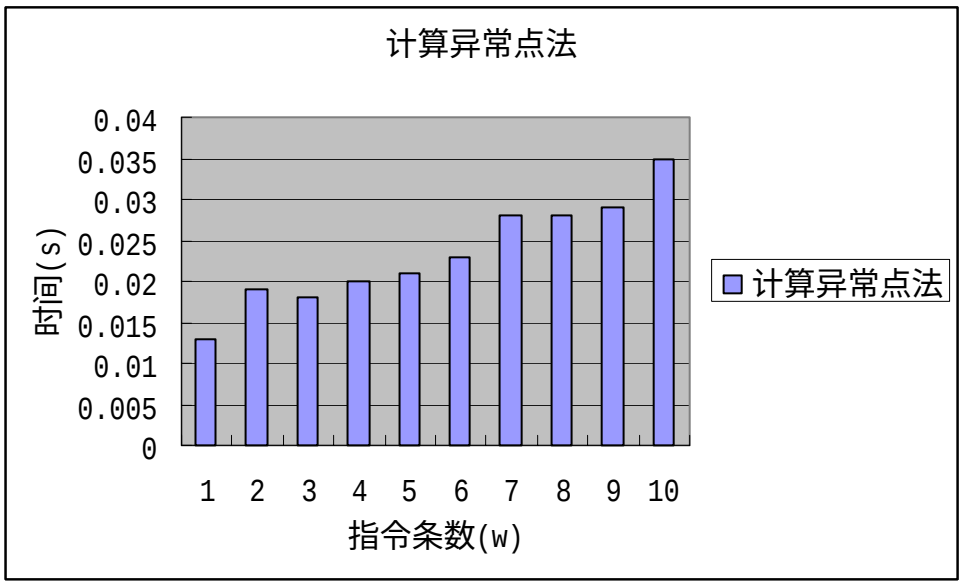

图5.14 计算异常点法性能图

5.3 性能测试

这里将异常处理系统和GDB在发生异常情况下进行比较，如表5.2表示GDB当异常发生时的检测情况，表5.3表示异常处理系统当异常发生时的检测情况。

表5.2 GDB异常检测情况

异常类型	指令位置	异常类型提示	通用寄存器	标志位寄存器
除法异常	是	是	否	否
DEBUG异常	是	否	否	否
浮点溢出异常	是	否	否	否
断点异常	是	否	否	否
边界溢出异常	否	否	否	否
一般性保护异常	是	否	否	否
页面出错异常	是	否	否	否

表5.3 异常处理系统异常检测情况

异常类型	指令位置	异常类型提示	通用寄存器	标志位寄存器
除法异常	是	是	是	是
DEBUG异常	是	是	是	是
浮点溢出异常	是	是	是	是
断点异常	是	是	是	是
边界溢出异常	是	是	是	是
一般性保护异常	是	是	是	是
页面出错异常	是	是	是	是

从表5.2和表5.3分析得出，异常处理系统与GDB相比，在指令定位方面比较，具体表述如下。

1. 异常处理系统能够定位边界溢出指令位置而GDB无法定位这类异常。

2. 异常处理系统在异常发生时能提示异常的原因，而GDB仅提供了除法异常原因。

3. 当异常发生时异常处理模块能够给出异常点之前通用寄存器状态，而GDB无法提供此类信息。

4. 当异常发生时异常处理模块能够给出异常点之前标志位寄存器状态，而GDB无法提供此类信息。

因此在发生异常情况下，异常处理系统提供了比GDB更为详细的信息，而这些信息有助于程序员提高调试程序的效率。

5.5 小结

本章从功能测试与性能测试两方面对系统进行了全面评估。功能测试对系统提供的主要功能进行测试，主要是对各种异常二进制ELF文件的异常检测，并以GDB连接调试，验证了异常处理过程中寄存器状态的正确性。结果证明了系统各部分功能的可用性与正确性。同时从设置异常点法和计算异常点法两方面对异常处理模块进行了全方面的检测，并分别使用除法异常和页面出错异常对系统进行测试，实验表明：设置异常点法，随着指令条数的增长，指令执行时间基本线形增长，而使用计算异常点法由于要考虑此方法是在执行过程中重新去计算异常点方法，而这些时间损耗是与异常点在原翻译块位置，以及翻译块长度有具体关系，所以并不是随指令数增长而出现线性增长。 在性能方面表明，在发生异常情况下，异常处理系统提供了比GDB更为详细的信息，而这些信息有助于程序员提高调试程序的效率。

6 总结与展望

面向二进制翻译的精确异常的研究是当今二进制翻译研究领域的热点问题，同时也是二进制翻译过程中的难点问题，它涉及到操作系统，体系结构，算法设计等一系列与计算机相关的领域相关知识。本文研究面向二进制翻译块翻译过程中的精确异常问题，并设计与实现了改进动态二进制翻译器 QEMU 精确异常处理的机制。二进制翻译的优化技术很好的解决了翻译器在翻译执行过程中的效率问题，而精确异常解决在翻译执行过程中翻译的正确性问题即要保证在翻译过程中。一旦程序出现错误异常时，要恢复到异常出现之前的寄存器状态。

本文主要是以 QEMU 为实验平台，在此基础上开发与实现了一种基于 web 的异常处理调试器，实现了在动态翻译过程中翻译块中异常指令的定位。针对异常产生的原因与特点不同，并提出了一种对异常进行处理的异常处理模型。在开源动态二进制翻译器 QEMU 基础上，对精确异常处理技术进行了相关研究，对全文的工作总结如下：

(1) 介绍了当今二进制翻译器发展的历史发展方向以及研究的热点和难点问题，并对当今二进制翻译器中的若干问题进行了分析和比较。介绍了当今两大类解决精确异常的方法，分析了系统级翻译的精确异常处理以及应用程序级翻译的精确异常处理特点与不同。

(2) 设计与实现了改进 QEMU 精确异常处理的异常处理机制的系统 PEHS(Precise Exception Handling System)，给出了一种对异常进行处理的机制，并将异常分为可预测性异常和不可预测性异常。利用设置异常点法对可预测性异常处理，使用计算异常点法对不可预测性异常进行处理，最终实现对 QEMU 用户模式下异常指令的准确定位。

(3) 集成 QEMU 远程调试功能，当与 GDB 调试器连接时，响应 GDB 命令请求，并实现调试过程中状态监控。当用户在执行过程中出现异常时，以网页方式显示异常处 CPU 状态，具有与用户交互强，简化用户操作的特点，因此具备很好的实用价值。

基于面向二进制翻译的精确异常的研究已经得到了近 10 年的发展，目前依然还是一个十分活跃的研究领域。现在很多公司对二进制翻译技术中关于精确异常问题

都进行了深入研究和探讨，不过所开发的这些系统都是用于商业用途，对外披露很少，本文在上面各方面做了一些有意义的研究成果，但是在这些领域来说还是微不足道的。个人认为论文还可以在以下几个方面得到完善和改进。

1.由用户模式向全系统模式转化

PEHS 是以 QEMU 为实验平台，利用 QEMU 的用户模式，解决了在用户模式下的精确异常的处理，PEHS 共处理了七类异常，但是异常的种类不仅仅如此，还有很多异常需要去处理，而这些异常往往又是在全系统模式下才会出现的，而且全系统模式就是对整个操作系统的模拟，现在用户普遍使用的也是全系统模拟，对其异常的处理，显得就更加有意义。

2.精确异常处理与优化协同机制研究

在二进制翻译中，精确异常处理研究的是翻译的正确性问题，而优化研究的是翻译的效率问题，但是往往优化以后，比如形成 TRACE，SUPERBLOCK 以后，源指令由于优化，导致指令顺序的不一致，或者优化后冗余指令的删除，这样都给我们异常处理指令的定位，带来了很大的困难和挑战。研究精确异常和优化的协同机制成为本系统下一步的工作中心和研究难点。

致 谢

在本文即将结束之际，我要感谢实验室众多才华横溢，学术严谨的老师，是他们热情无私的帮助和指导，在我学习与研究中遇到困难和挫折的时候，给了我最强有力的支持，不仅让我在学术水平有了长足的进步，更让我深深的感受到实验室强大的科研实力，真的很高兴自己能来到实验室这个大家庭。也感谢在实验室和我一起工作学习过的同学们，在两年的日子里大家经常在一起讨论问题交流思想，与大家的交流不仅使自己从他们那里得到了很多好的方法和思想使自己的技术水平得到了提高，更重要的是让我明白了在开发项目的时候协同合作的重要性，一直到现在我都坚信当初选择来我们实验室的决定没有错。

首先感谢我的导师章勤教授，她一丝不苟的治学态度、热情待人的处世态度和忘我的工作态度深深地影响着我。她丰硕的研究实践成果和广博的阅历，是我学习、研究、工作时的灯塔。由于自己是武汉人，一直生活在父母亲身边，父母亲能给与我的更多的是物质上的东西，而她传授给我的是研究问题的方法，处事的态度，做人的道理，面对困境时的冷静。忘不了每当我遇到挫折时，她对我的谆谆教导和鼓励，感谢她在学习研究上给我以悉心认真的指导、鼓励和鞭策，真的感谢她在这两年时间里教导我如何变得成熟，如何去面对未来的人生以及如何把握未来的人生。

感谢实验室主任金海教授，他以自己的研究实践和广博的阅历为大家打开了新的窗口，引导着实验室的研究与发展方向，为实验室营造了一个崇尚创新、充满活力的研究氛围，为实验室的发展壮大尽心尽力，也向我们展示国际最前沿的研究动态。他推崇技术创新，提倡学术自由，营造了一个充满活力的研究氛围；他对事业执着追求并不懈奋斗，是我终生学习的榜样。由于金海教授平时工作繁忙，自己很难有机会直接接受他的指导，但通过开题，暑期年会以及并行高性能计算的课堂上这些宝贵的机会，仍然能够倾听他的演讲和授课，让我深深感受到他敏锐的洞察力和对全局的把握，也促使我在找工作时立志要当一名系统架构师，而不是一名简单的编程人员。

感谢韩宗芬教授和李胜利教授，他们的严格要求、他们的认真执着深深地影响了我，尤其是在开题、演示系统开发等各个阶段，韩教授和李教授都给予了我最热情与最悉心的指导和帮助，真的让我深受感动，每次在实验室遇见他们都特别有亲

切感，同时也使我学会了如何学习、如何工作、如何处世。

感谢喻之斌老师，他思路活跃、学术能力强，通过他对项目组的指导，使项目组的研究在正确的道路上不断发展，也让我接触了很多学术和技术上的创新，开阔了我的思维，他细心的工作帮助项目组成长壮大，平时每天晚上都还能看见他在实验室里努力的工作，这种精神也正是我作为一名即将走上社会的工作者所必需具备的素质，自己会努力向他看齐。

感谢项目组里的胡亚斌博士，刘安战，陈结，郑鸿阳，任菁菁，余璜和他们一起工作学习的日子让我很难忘，也感谢他们给我的帮助和支持，提高了自己技术水平，也从他们身上学到了很多做人做事的道理，再次感谢他们。

感谢郑然老师和孙傲冰博士，在我刚刚进入实验室时给我的指导与帮助。他们对学生的关怀与爱护深深感染了我，他们对工作的严格要求与对技术的不断进取，至今仍在鞭策着我。

感谢师兄虢伟，褚攀，杨文，周清存，师姐王建，史英杰，感谢大家在学习与科研上给我的帮助，使我能够以最快的速度融入到实验室这个大家庭中。忘不了大家在小组讨论时的畅所欲言，忘不了大家对技术的孜孜追求，感谢他们在我挫折时对我的关怀。

感谢与我一同有幸加入实验室大家庭的潘正秋，刘欣，陈巍，习昱鄂，田晶，罗锋，刘安战，张德，江来源，周拥刚，李金虎，乐一帆，廖光贤，陆晓雯，王敬彤，袁金艳，邵颖哲，余洋，项国富，李勇，朱伟，熊贤杰，陈镇光，陈青茶，郭国信，刘三民，王凯，周凡，黄宇华，程岚，杨卫平，廖红艳，陈玉芹，孙肖兴，何汉，竺丽丽，严耀伟还有曾经所有帮助过我的师兄师姐们，感谢你们使我感受到了实验室这个大家庭的温暖，使我在这两年的时间里与你们一同成长。

感谢我的母亲和我的奶奶以及一直都关心着我的人，他们总是在背后默默无私的支持着我，一直记挂我的身体和学业，而我至今仍无以为报，祝福他们安康，快乐！

最后，衷心感谢各位评委的批评和指导!

参考文献

[1] 唐锋.动态二进制翻译优化研究，博士学位论文，中国科学院研究生院，2004.

[2] Michael Gschwind，Eric R Altman. Precise exception semantics in dynamic compilation. The Symp on Compiler Constructions, Grenoble, France, 2002.33~43

[3] Ho-Seop Kim, James E. Smith. Dynamic Binary Translation for Accumulator-Oriented Architectures. In: Proc. of the 1st Int ’l Symp. on Code Generation and Optomization，2003.25~35

[4] R. Altman, D. Kaeli, Y. Sheffer. Welcome to the Opportunities of Binary Translation. Computer, 2000, 33(3): 40~45

[5] K. Ebcioglu, E. Altman. DAISY: Dynamic Compilaton for 100 Percent Architectural Compatibility. In: Proceedings of ISCA24. USA: ACM Press, 1997. 26~37

[6] M. Gschwind, E. Altman. Dynamic and Transparent Binary Translation. Computer, 2000, 33(3): 54~59

[7] A. Klaiber, D. Kaeli. The Technology behind Crusoe Processor. In: Proceedings of Transmeta technology report. USA: ACM Press, 2000. 3~12

[8] C. Zheng, C. Thompson. PA-RISC to IA-64: Transparent Execution, No Recompilation. Computer, 2000, 33(3): 47~52

[9] K. Ebcioglu, E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In: Proceedings of the 24th Annual Intl Symp on Computer Architecture. USA: ACM Press, 1999. 128~141

[10] K.Ebcioglu, C. Thompson. Execution-Based Scheduling for VLIW Architectures. In: Proceedings of Europar99 Lecture Notes in Computer Science. Berlin: IEEE Press, 1999. 1269~1280

[11] K. Ebcioglu, E. Altman. Dynamic Binary Translation and Optimization. IEEE Transactions on Computers, 2001, 50(6): 36~51

[12] 罗金平，俞磊，周兴铭. VLIW 技术的最新发展.计算机工程，2002，28(1):1~4

[13] J. Dehnert, B. Grant, J. Banning, et al. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In: Proceedings of the Intl Symp on Code Generation and

Optimization. USA: ACM Press, 2003. 37~56

[14] M. Gschwind. Dynamic and Transparent Binary Translation. Computer, 2000, 33(3): 52~63

[15] M. Gschwind, E. Altman. Inherently Lower Complexity Architectures using Dynamic Optimization. In: Proceedings of workshop on Complexity Effective Design in conjunction with ISCA-2002. USA: ACM Press, 2002. 133~147

[16] L. Barez, T. Devor, O. Etzion, et al. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium based systems. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Micro architecture (MICRO-36). USA: IEEE-CS Press, 2006.12~23

[17] 李剑慧,马湘宁,朱传琪. 动态二进制翻译优化研究.计算机研究与发展,2007,.44(1):161~168

[18] J. Li, P. Zhang, O. Etzion. Module-aware translation for real-life desktop applications. In: Proceedings of the 1st ACM/USENIX Intl Conf on Virtual Execution Environments. Chicago: ACM Press, 2005. 102~115

[19] J. Li, Q. Zhang, S. Xu, et al. Optimizing dynamic binary translation for SIMD instructions. In: Proceedings of the Intl Symp on Code Generation and Optimization. USA: ACM Press, 2006. 117~129

[20] A. Chernoff, M. Herdeg, R. Hookway, et al. FX!32: a Profile-Directed Binary Translator. IEEE Micro, 1998, 18(2): 2~21

[21] R. Hookway. DIGITAL FX! 32 running 32-Bit x86 applications on alpha. In: Proceedings of IEEE COMPCON 97. USA: IEEE Press, 1997. 26~42

[22] R. Hookway, M. Herdeg. Digital FX! 32: Combining emulation and binary translation. Digital Technical Journal, 1997, 9(1): 3~12

[23] F. Bellard. QEMU, a fast and portable dynamic translator. In: Proceedings of USENIX Annual Technical Conference. USA: ACM Press, 2005. 41~46

[24] "QEMU: The open source processor emulator," http://fabrice.bellard.free.fr/qemu/about.html

[25] 李晓明，吴浩. 二进制翻译器 QEMU 的优化技术，硕士学位论文，上海交通大学，2007

[26] James E.Smith, Ravi Nair.虚拟机-系统与进程的通用平台(英文版)，北京，电子工业出版社，2006.122~136

[27] M. Gschwind. Dynamic and Transparent Binary Translation. Computer, 2000,

33(3): 54~59

[28] A. Klaiber. The Technology behind Crusoe Processor, Transmeta technology report, Jan. 2000. 3~12

[29] D.R. Engler. a Retargetable, Extensible, Very Fast Dynamic Code Generation System. In: SIGPLAN Conference on Programming Language Design and Implementation, 1996.160~170

[30] Intel Corporation. Intel@64 and IA-32 Architectures Software Developer's Manual USA: Intel Corporation，2007.143~168

[31] C. Cifuentes，V. Malhotra. Binary Translation: Static, Dynamic, Retargetable. In: Proceedings International Conference on Software Maintenance. IEEE-CS Press，1996. 340~349

[32] C. Cifuentes, M. Van Emmerik, D. Ung, et al. Preliminary Experiences with the Use of the UQBT Binary Translation Framework. In: Proceedings of the Workshop on Binary Translation. USA: IEEE-CS Press, 1999. 12~22

[33] C. Cifuentes, B. Lewis, D. Ung. Walkabout-A Retargetable Dynamic Binary Translation Framework. In: Proceedings of Fourth Workshop on Binary Translation. USA: IEEE Press, 2002. 76~89

[34] D. Ung, C. Cifuentes. Dynamic re-engineering of binary code with run-time feedbacks. In: Proceedings of Seventh Working Conference on Reverse Engineering. USA: IEEE Press, 2000. 2~10

[35] D. Ung, C. Cifuentes. Machine-Adaptable Dynamic Binary Translation. In: Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization. USA: ACM Press, 2000. 30~40

[36] D. Ung ,C. Cifuentes. Optimising Hot Paths in a Dynamic Binary Translator. In: Proceedings of Second Workshop on Binary Translation. USA: ACM Press, 2000. 23~42

[37] N. Ramsey, M. Fernandez. Specifying Representations of Machine Instructions. ACM Trans.Programming Languages and Systems, 1997, 13(9): 492~524

[38] 白童心. 动态二进制翻译与动态优化相关问题研究. 计算机工程，2004, 23(4):23~32

[39] 马湘宁.二进制翻译关键技术研究，博士学位论文，中国科学院，2004

[40] 冯晓兵，马湘宁，武成岗等. 二进制翻译中的标志位优化技术.计算机研究与发展 ，2005，42 (2) : 329～337

一种改进QEMU精确异常处理机制的研究

作者： 余璐
学位授予单位： 华中科技大学