申请上海交通大学硕士学位论文

二进制翻译系统QEMU的优化技术

Optimization for Binary Translation System QEMU

本论文得到
“国家重大基础研究（973）前期研究专项”支持
项目编号：2004CCA02600

系　　别	信息安全工程学院
学科专业	计算机应用
研究方向	计算机体系结构与嵌入式系统
姓　　名	吴　浩
导　　师	李晓勇

上海交通大学信息安全工程学院

2007 年 1 月

Thesis submitted to

Shanghai Jiao Tong University

for the degree of Master in Computer Science

Optimization for Binary Translation System QEMU

Candidate: Wu Hao

Supervisor: Li Xiaoyong

The School of Information Security

Shanghai Jiao Tong University

January, 2007

上海交通大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，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，同意学校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。本人授权上海交通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。

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘要

当前计算机技术发展受制于软硬件之间的冲突，为改变这种局面研究人员提出了代码移植技术，二进制翻译技术就是实现代码移植的一种方法。由于二进制翻译技术在代码的移植、Virtual IT Shop、虚拟机、计算机安全和硬件开发等方面的重要作用，已经成为现代计算机研究领域的热点之一。本文首先综述了二进制翻译技术的基本原理和方法以及研究现状，然后介绍了我们试验使用的动态二进制翻译系统平台QEMU，最后深入研究了其中的若干关键优化技术。本文提出了三种提高二进制翻译系统运行效率的优化技术：跳转优化、寄存器映射优化和基本块覆盖优化。

本文的主要贡献为研究了目前二进制翻译领域的典型翻译系统，详细研究了动态二进制翻译系统QEMU的翻译机制、运行方式、翻译策略，并使用其用户级系统作为我们的实验平台。QEMU动态二进制翻译系统实验平台上，针对翻译过程中每个基本块运行结束之后过于频繁的状态切换和判断提出跳转控制优化方案。即在明确跳转目标时判断目标基本块是否已经被翻译过，如果当系统发现跳转目标基本块已经被翻译时，可以直接把跳转的源基本块和跳转目标基本块在T-Cache中连接起来，从而降低了动态翻译系统自身的开销。针对QEMU动态二进制翻译系统中将中间变量映射到宿主机寄存器上的翻译机制，对寄存器的不同映射方案进行了性能测试。发现了在目前翻译机制下，中间变量的确是使用最为频繁，最有价值映射的部分。最后提出了取消中间变量的新翻译机制设想。针对QEMU动态二进制翻译系统中每个基本块以头指令pc作为唯一标识的方式，发现了基本块覆盖的存在。即基本块可能是某个基本块的一部分，也有可能包括一些基本块。对此提出了减少基本块覆盖现象的方案，并且将其实现。实验数据表明优化方案的确可以提高系统的整体性能。

关键词：二进制翻译、优化、基本块、跳转、寄存器

Abstract

Now, the development of computer technology is limited by the conflicts between hardware and software. To solve this problem, the researchers have invented the code migration. Binary translation is an important technology to implement code migration. Because of the importance in code migration, virtual IT shop, virtual machine, computer security, hardware design and many other aspects, binary translation become a hot point in modern computer development field.

In this dissertation, the research background, the related concepts, and some means of binary translation are introduced first. Then a binary translation system, QEMU, which we developed our research on, are introduced, followed by deep research on several optimization techniques. This dissertation raises three techniques to improve the performance of binary translation system: direct jump optimization, register mapping optimization, and basic block overlapping optimization.

The main contributions of this dissertation are:

1. We research all existing binary translation systems, and pay special attention to the QEMU system. We research the translating mechanism of QEMU in detail, and use its user level system as our experience plant.
2. Because there are too frequent status swaps and judgments after each basic block be implemented, we design and realize the jump control optimization. The basic point of this optimization is when the target basic block of a jump instrument has been translated and cached in the T-cache, the system will junction the first basic block and the one after it automatically, make it looks like one. So the system will directly implement both basic blocks without status swap and judgment between them.
3. We test different mapping plan for the virtual register, and verified that under the translation mechanism of QEMU, the mapping plan already used is the best choice. But the translation mechanism is not the best design, we

make some improvement.

4. There is a lot of basic block overlapping in QEMU, because QEMU uses the first instrument ‘s PC as the sign of the whole basic block. This situation reduce the utilization of T-cache. To eliminate it, we design a optimization, and implement it. The test shows it can improve the performance of the whole system.

Keywords: Binary Translation, Optimization, Basic Block, Jump, Registar

目录

第一章 绪论

在目前计算机的发展中，微处理器设计面临一个十分棘手的问题。一般来讲微处理器能否在市场上获得成功取决于三个方面的因素:性能、价格和支持的软件。性能主要由所采用的物理器件的速度和体系结构的先进性决定。价格则随着技术水平，生产规模和市场供需关系变化。一般说来，性能价格比越高的处理器越受欢迎，越容易成功。但是，支持软件的多少以及由此引起的用户的惯性常常扭曲了这一推理。一方面，如果硬件开发商推出一种与以往任何一家都不兼容的新指令体系结构的机器，那么就没有现成的系统软件和应用软件可用，因而软件的缺乏使这种机器也就很难在市场上发展壮大。另一方面，软件开发商只愿意为用户数最多，最流行的机器开发软件。

上面二个因素合起来扼制了处理器设计的创新，造成了这样一种现象，商业上获得成功的微处理器都背有老的指令体系结构(Instruction Set Architecture, ISA)这个历史包袱，其中有的已经有几十年的历史。尽管这些ISA有不容置疑的缺陷，但是开发商却不愿从根本上解决问题，开发新的ISA。因为他们担心失去支撑当前产品的软件基础，从而损害其商业利益。最显著的例子就是Intel公司的X86系列的处理器，采用CISC指令集体系结构，由于其在长时间的发展中，积累了大量的软件，为了保持它自身向下兼容性，不能完全摈弃原有的寻址方式、指令模式等过时的技术，影响了其体系结构的创新

在这种情况下，研究不同体系结构之间的软件移植，不仅对软件重用有重大意义，更可以开阔微处理器研发的思路，促进新处理器的创新。二进制翻译作为实现软件移植的一种行之有效的方法，近年来逐渐成为研究的热点。

1.1 二进制翻译的概念

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。由于现有的机器都属于图灵机，所以从原则上说，在一台机器上的所有计算都能够通过二进制翻译技术在另一台机器上模拟运行[1]。总之，二进制翻译技术能够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，它与传统编译器的差别在于其编译处理对象不一样。二进制翻译处理的是某种机器的目标二进制代码，该目标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目标二进制代码；而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。

仿照传统编译器前端、中端、后端的划分，二进制翻译在概念上也可以分为三个阶段。

1.1.1 前端解码器

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。这部分需要准确地对二进制代码进行解码，处理间接跳转/间接调用、自修改代码、数据内容的识别和分析等。解码器的输出是某种形式的抽象中间表示，以便于对其进行分析、优化。这个阶段的功能与传统编译器的前端类似，但接收的对象不同:传统编译器的前端是将用源语言书写的程序进行分析。因此二者的一些核心技术和关键问题也有所不同。

1.1.2 中端分析优化器

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，实现到口标机器的转化，并且对程序进行分析和部分优化。中端 分 析优化器的首要任务是实现两级中间表示的等价变换。为了更好地体现目标平台的特点，在编译系统中，与机器直接对应的中间表示或多或少有着该机器的特点，在二进制翻译系统中，采用两级中间表示可以适当隔离不同的机器平台的特点，从而方便系统的设计与实现。若这两级中间表示都与机器无关，则有利于二进制翻译系统的移植，即无论是源机器改变，还是目标机器改变，只需要相应地调整前端解码器或后端代码生成器，即可实现一个适用于新机器的二进制翻译系统。

1.1.3 后端优化编码器

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。它根据目标机器以及目标机器采用的操作系统的特点，将中间表示翻译为目标机器上可执行的二进制代码，综合了常规编译系统中的后端代码优化和生成器，以及类似链接器和装载程序的功能。

1.2 二进制翻译方法分类及比较

根据翻译时机的不同，二进制翻译分为静态二进制翻译和动态二进制翻译。

1.2.1 静态二进制翻译

静态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。其结构如图 1-1 所示。静态翻译器离线翻译程序，有足够的时间进行优化，效率较高；但是静态二进制翻译有其局限性，现在使用的计算机几乎都还是采用冯·诺依曼结构，指令和数据存储在一块，并以相同的方式表示。因此，要静态地发现一个程序中所有的指令代码一般是不可能的。而且有些信息在静态时是处理不了的，例如，间接分支、共享库、自修改代码以及精确异常状态等。可见静态翻译不能完全处理程序，它需要依赖解释器的支持，开销巨大，而且需要终端用户的参与，缺乏透明性。

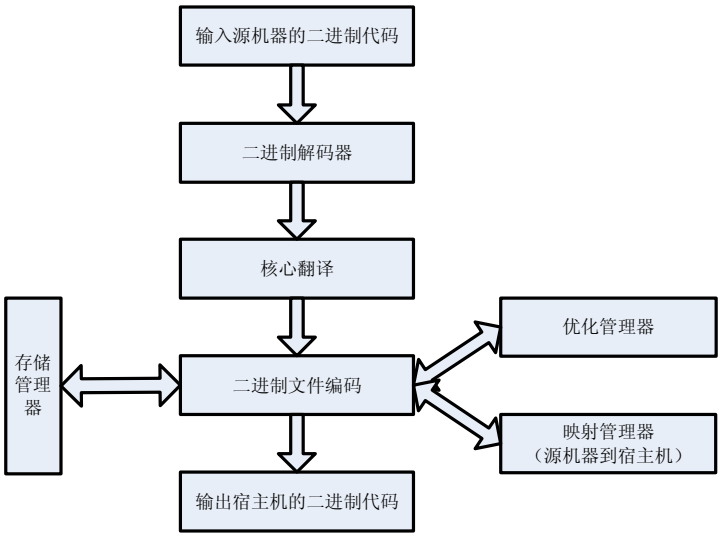

图 1-1 静态二进制翻译系统的基本结构

Figure 1-1 Static binary translation system framework

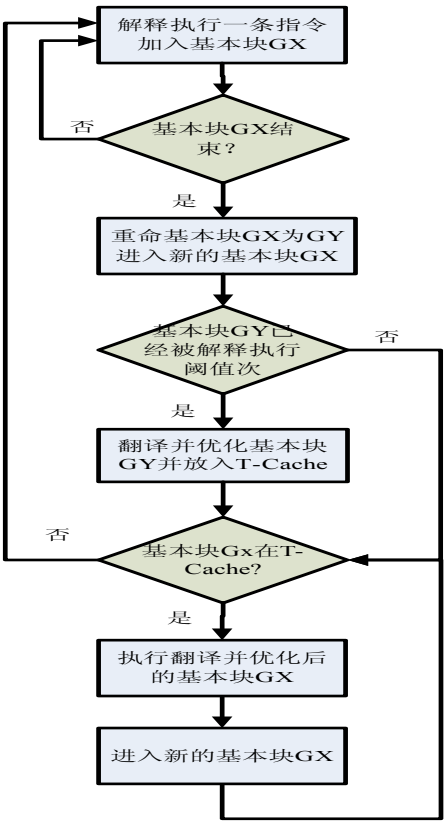

图 1-1 动态二进制翻译系统的基本结构

Figure 1-2 Dynamic binary translation system framework

1.2.2 动态二进制翻译

动态二进制翻译采用一边翻译一边执行的方法。原机器代码只有在被执行到时才被翻译。通常二进制翻译采用解释执行和二进制翻译相结合的策略，以基本块（Basic Block）作为翻译的基本单位，一个基本块是一个以一个控制转移（如一个分支、调用或跳转指令）结束的指令序列。其执行过程如图 1-2 所示，在执行一个基本块之前先判断该基本块是否已经被翻译，如果在翻译缓存(T-Cache: Translation cache 用于存放翻译产生的代码的一段连续内存地址空间)中可以找到翻译好的代码就可以直接执行，否则只能解释执行基本块内的所有指令。如果该基本块被解释执行的次数超过了某个阈值（被频繁执行），则翻译该基本块并做一定的优化，翻译后的代码被放入到 T-Cache。最后进入下一个基本块的处理。

动态二进制翻译能够收集到程序动态运行时的信息，有的放矢的翻译和优化源机器代码。它冲破了一些静态二进制翻译的局限性，拥有很好的透明性，无需用户参与。但是由于在运行代码的同时，目标机器还要做翻译代码和统计运行信息的工作，动态二进制翻译牺牲了自己的性能。目前的大部分动态二进制翻译系

统都没能达到用户的性能需求。所以说动态二进制翻译的优化就成为了目前非常紧迫的任务。如果用户可以在宿主机上获得接近于源机器上的程序运行速度，动态二进制翻译技术将会很快得到广泛运用。

1.3 二进制翻译技术的研究热点

二进制翻译技术可以应用在不同领域，满足不同的需求。在设计二进制翻译系统的时候可以从很多方面来权衡和选择。二进制翻译可以是解释执行或者翻译/优化；静态翻译或者动态翻译；模拟一个虚拟机器或者模拟真实机器；完整系统或者用户级；操作系统相关或者操作系统独立；同一指令集或者不同指令集等等。二进制翻译技术在以下几个方面还需要完善，即为目前的研究热点：

1. 自修改代码：源机器的代码被修改时，与该代码段对应的任何翻译都要被置为无效。
2. 异常的精确性：同步异常(如页面故障)和异步异常(如时钟中断)发生时，翻译器的异常处理机构必须提供一个与原结构状态一致的、正确的状态。
3. 地址翻译：不同硬件结构的地址空间设计存在一定的差异，对 I/O 地址的处理也大相径庭，全系统的二进制翻译还必须完成虚拟地址和物理地址之间的转化。
4. 自引用代码：自引用代码会进行自校验或者查看自己的代码，所以二进制翻译系统必须保存源机器代码的备份。
5. T-Cache 的管理：T-Cache 的大小是有限的，当 T-Cache 满时，需要为新翻译生成的代码腾空间；被置为无效的翻译，也要在 T-Cache 标识。
6. 实时行为：二进制翻译系统必须根据代码是否已经被翻译、被如何翻译来决定执行速度的不同，执行时间是一个不定因素。
7. BOOT 和 BIOS 代码：在全系统的二进制翻译中，控制源机器的最低级代码必须被如实地翻译到宿主机上。

由于研究人员的不懈努力，二进制翻译技术已经日趋成熟，这些二进制翻译面临的挑战也已经能得到比较妥善的解决，翻译的开销能够被控制在可以忍受的范围之内。

1.4 本文的贡献

二进制翻译之所以发展得这么快并且一直受到人们关注，是因为不论从研究的角度还是从商业的角度来看，二进制翻译的一些特性非常具有吸引力：首先，可以把二进制翻译的思想引入芯片设计中，简化硬件设计。兼容性方面由二进制翻译系统负责，硬件设计就可以轻装上阵，原本看来与主流结构（如 X86、PowerPC）不兼容的新结构思想都可以采用，更容易做到简单、高效和节能。其次，软件也可以得到更加宽松的发展空间。程序员不必考虑底层硬件的详细信息，而且开发的软件可以在多种不同的硬件平台运行，也不会因为硬件的升级而被迫更新，甚至被丢弃。第三，二进制翻译技术还带来了极大的灵活性，可以针对不同的应用底层结构进行裁剪以获得高性能，底层结构的实现和翻译优化软件可以独立地进行升级。最后，动态二进制翻译可根据程序运行的具体情况来，选择被频繁执行的代码块进行针对性的优化，优化的结果可以供将来直接使用，从而提高程序的执行效率。

上海交通大学于 2004 年申请的项目《二进制翻译技术》(“国家重大基础研究（973）前期研究专项”，项目编号：2004CCA02600)。该项目前期做了很多关于二进制翻译技术的研究和探讨，并提出了一个多源多目标二进制翻译系统的构想。我们是在前面的研究成果上，进一步从事动态二进制翻译的优化技术的研究。在此过程中我们掌握了现有的多种著名二进制翻译系统的实现机制，以及他们所采用的优化技术。在借鉴他们成功经验的同时，我们还创新性的提出一些重要的优化方法并收到了很好的优化效果。

正因为动态二进制翻译技术能够突破静态二进制翻译技术的局限性，所以得到了大多数研究者的青睐。但是动态二进制翻译在运行代码的同时，宿主机还要做翻译的工作并协调两者的工作进度。性能较低往往是动态二进制翻译的瓶颈所在，性能优化成为了非常紧迫的任务。但是动态二进制翻译可以窥探代码行为，收集动态运行信息，有非常广阔的优化空间。近年来，动态调度和动态编译器优化技术的发展给动态二进制翻译优化提供了很多可以借鉴的经验。动态二进制翻译的优化成为了非常有潜力和有意义的研究方向。我们在借鉴当前二进制翻译系统所采用的优化技术基础上，提出并实现了一些创新性的优化方法，并根据实际应用选择最佳的优化策略。我们在 QEMU 动态二进制翻译系统实验平台上，针对

翻译过程每个基本块运行结束之后过于频繁的状态切换和判断提出跳转控制优化方案。即在明确跳转目标时判断目标基本块是否已经被翻译过，如果当系统发现跳转目标基本块已经被翻译时，可以直接把跳转的源基本块和跳转目标基本块在 T-Cache 中连接起来，从而降低了动态翻译系统自身的开销。针对 QEMU 动态二进制翻译系统中将中间变量映射到宿主机寄存器上的翻译机制，对寄存器的不同映射方案进行了性能测试。发现了在目前翻译机制下，中间变量的确是使用最为频繁，最有价值映射的部分。最后提出了取消中间变量的新翻译机制设想。针对 QEMU 动态二进制翻译系统中每个基本块以头指令 pc 作为唯一标识的方式，发现了基本块覆盖的存在。即基本块可能是某个基本块的一部分，也有可能包括一些基本块。对此提出了减少基本块覆盖现象的方案，并且将其实现。实验数据表明优化方案的确可以提高系统的整体性能。

1.5 文章的组织

论文的组织结构如下：

第二章介绍了目前国内外有关二进制翻译技术的研究状况。

第三章介绍了我们的实验平台 QEMU。重点介绍它的整体框架设计，以及具体翻译机制的实现过程。

第四至六章是论文的核心部分，详细介绍了我们在动态二进制翻译优化技术领域的研究成果。阐述了对二进制翻译系统的总统设计以及不同体系结构之间翻译机制的具体优化实现方法。其中总体设计方面有地址空间的设计和基本块的连接；而具体翻译机制方面包括了寄存器的映射。

第七章对全文进行总结，对二进制翻译技术进行了总结，并对进一步的研究工作提出了展望。

第二章 二进制翻译技术研究现状

二进制翻译作为代码移植的重要方法，得到了广泛重视。从最早20世纪80年代开始出现第一个二进制翻译系统至今，二进制翻译技术取得了许多研究成果，并相继研制出了很多实验性和商用的二进制翻译系统。

本章首先概述二进制翻译技术的发展过程，接着分类介绍了一些有代表性的翻译系统的设计框架和采用的技术，以及它们对第一章提到的某些热点问题的解决方法。

2.1 二进制翻译技术发展概述

二进制翻译系统已有近二十年的研究历史，产生出了丰硕的研究成果和有商业应用价值的二进制翻译系统。下面的表2-1中按照年代先后顺序列出了二进制翻译研究历史中一些有代表性的系统，他们基本反映了二进制翻译的研究历史和发展过程，基本涵盖了各个研究阶段的技术成果。对这些已有技术和系统的研究，将会对我们研究二进制翻译技术起到重要的指导意义，有益我们创造出更加高效实用的二进制翻译技术和系统。

表2-1按照源平台和目的平台对二进制翻译系统进行了分类和总结，下面将会按照对这些系统的特点进行详细的分析和介绍。

名称	研究单位	特点	源平台	目的平台
Bergh etal (1987)	HP	最早的商用二进制翻译系统;用于软件模拟和目标代码翻译	(HP3000, MPE V)	(HP Precision Architecture, MPE XL)
Mimic (1987)	IBM	对每条源机器指令代码扩展倍数为 4 的软件模拟器	IBM system/370	IBM RT PC
Accelera tor (1991)	Tandem	将 CISC 移植到 RISC 的静态翻译器，采用解释器作为补充	TNS CISC	TNS/R
VEST, mx (1993)	Digital	从 Digital 公司的 VAX 和 MIPS 到 64 位 Alpha 静态翻译器，采用解释器作为补充	(VAX, OpenVMS), (MIPS, Ultrix)	(Alpha, OpenVMS), (Alpha, OSF/1)

Flash-port (1994)	AT&T	跨越多个源和目标平台的二进制翻译，需要人为干预	68x0 Mac, IBM system/360, 370,380	Power Mac, IBM RS/6000 SPARC, HP, MIPS, Pentium
MAE (1994)	Apple	在 PowerMac 上开发的一个 Motorola 68000 解释器.后来改进成翻译器。	680x0	RISC-based Unix
Freeport Express (1995)	Digital	静态翻译器，采用解释器作为补充，翻译用户模式程序，32 位、64 位都可	(SPARC, SunOS 4.1.x)	(Alpha, OSF/1)
FX!32 (1996)	Digital	从流行的 x86 应用程序到 Alpha 的混合模拟器和二进制翻译器	x86, Windows NT	Alpha, Windows NT
Daisy (1996)	IBM	利用二进制翻译调度 PowerPC 代码到超长指令字(VLIW)处理器，增加并行性	PowerPC, UnixV	VLIW
Aries (1999)	HP	解释和动态翻译相结合，简化了应用程序从 HP Precision Architecture 到 IA-64 的翻译	HP Precision Architecture	IA-64
BOA (1999)	IBM	动态翻译了 PowerPC 的整个系统，用简单的指令实现原来语义，简化硬件	PowerPC, UnixV	PowerPC, UnixV
UQBT, UQDBT	Queensland 大学	利用机器指令描述开发的可变源、可变目标的静态(动态)二进制翻译器	可变	可变
Code Morphing software (2000)	Transmeta 公司	动态翻译运行 x86 代码，翻译全部程序包括 Windows 操作系统	x86	Transmeta 芯片
Bintran	维也纳技术大学	多源多目标的动态翻译器	可变	可变
Strat	弗吉尼业大学、微软	多源多目标的动态翻译器	可变	可变
IA-32 Execution Layer	Intel	通过软件方法在 IPF 上执行 IA32 的应用程序	IA32 (x86)	Itanium Process Family

表 2-1 二进制翻译系统简介

Table 2-1 Brief Introduction of Binary Translation System

从表2-1可以看出，这些二进制翻译系统按照源和目标是否可变大致分为两类，一类是固定源和目标的二进制翻译系统，即翻译系统的源机器和目标机器都

是确定的;另一类是可变源和目标的二进制翻译系统，即翻译系统可以为多种源机器和目标机器服务。下面将分别对这两类翻译系统的发展进行介绍。

2.1.1 固定源和目标的二进制翻译系统

早在1987年，HP公司就开发了最早的一个商用二进制翻译系统，用来将HP3000的客户转移到新的队体系结构上。

1991年Tandem公司开发了一个将CISC移植到R工SC的静态翻译器，采用解释器作为补充。

1992年开始DEC公司开发了一系列的二进制翻译器，用来将VAX/VMS,MIPS/Unix, Spare/Unix以及x86/WinNT上的代码翻译到他们新开发的Alpha机器上这其中以FX!32最有代表性。

1994年Apple公司在PowerMac上开发了一个Motorola68000解释器，后来改进成翻译器。

IBM公司1996年开发的Daisy，是利用二进制翻译调度PowerPC代码到超长指令字(VL工W)处理器，增加并行性。1999年开发BOA系统，动态翻译了PowerPC的整个系统，用简单的指令实现原来语义，简化硬件。

200 0年Transmeta公司宣布了Transmeta处理器芯片和同态代码软件，用来在完全不同的硬件基础之上翻译运行x86代码，甚至包括Windows。

2003年，Intel公司开发了IA32执行层软件IA32EL，通过软件方法在IA64机器上执行IA32的应用程序。

对于固定源和目标的二进制翻译系统，由于其源和目标机器确定，容易对系统进行机器相关的优化，获得较好的翻译效率。但这类系统不能为新二进制翻译系统的开发提供可用的代码，因而每次开发新系统时，都要花费大量时间从零开始。

2.2节我们将会对固定源和目标的二进制翻译的代表系统进行更加详细的介绍和分析。

2.1.2 可变源和目标的二进制翻译系统

上一节提到的翻译器都与机器特性高度相关，因而重利用非常困难。为了能

够加快二进制翻译器的开发过程，人们开始研究Aj变源和目标的二进制翻译系统。

1994年AT&T公司开发的Flashport二进制翻译器可以运用到多个源、目标平台。但不能完全自动化，需要一个专业用户通过图形用户界面(GUI)进行交互完成。

Queensland大学先后开发的UQBT以及UQDBT，都是基于对机器指令和操作系统属性描述的可变源、可变目标的二进制翻译器框架，可以看作是一个翻译器的生成器。

维也纳技术大学开发的Bintran动态二进制翻译系统，根据源、目标机器的机器描述产生一个C语言写的翻译器，达到可变源、目标的目的。

弗吉尼亚大学和微软公司共同开发的Strata系统[51][52]，既是一个可变源和目标机器的动态二进制翻译器，又可以通过裁剪来满足用户的不同目的，具有很好的可扩展性。

可变源和目标的二进制翻译系统，可以通过修改部分机器描述，重新生成新的二进制翻译系统，降低了开发新系统的开销。但是，由于机器特性差别很大，这类翻译系统无法针对某种特定机器做机器相关的优化;同时这类系统为了提取与机器无关的可重用代码，必然有从机器相关描述到机器无关描述的抽象过程。因此,可变源和目标的二进制翻译系统的效率通常不如固定源和目标的二进制翻译系统。

2.3节我们将会对可变源和目标的二进制翻译的代表系统进行更加详细的介绍和分析。

2.1.3 静态和动态二进制翻译在上述两类系统中的应用

从表2第四列可以看出，无论是固定源和目标的翻译系统，还是可变源和目标的翻译系统，都可以采用静态翻译、动态翻译、解释执行、以及它们之间相结合等各种翻译策略。

由于静态二进制翻译器的局限性，所有的实用系统都不采用纯静态的翻译。早期开发的系统多采用静态翻译加上解释器动态模拟支持的方法，典型如DEC公司开发的VEST/mx和FX!32翻译系统。

近期的研究多集中于动态翻译和动态优化，动态翻译可以保证程序能够正确地被翻译执行，动态优化则利用程序执行的信息，找出执行的热点进行优化，生成更高效率的代码，从而提高翻译系统的整体效率，如Daisy, Aries, BOA, Transmeta, IA32 EL等系统。此外，这种方式还可以对用户透明，从而无需用户对其过程进行干涉。

在2.2节和2.3节的二进制翻译系统分析中，我们包含了静态二进制翻译系统和动态二进制翻译系统，在2.4 节中，我们还将对动态优化技术进一步介绍。

2.2 固定源和目标的二进制翻译系统

二进制翻译技术初期的研究，以生产硬件的公司厂商为主，他们都是以自己的商业利益为目的，开发能够使新推出的处理器可以运行某种特定老处理器机器上的程序，使得那些习惯使用老机器上的软件的用户可以接受新机器。由于开发目标明确，公司开发的二进制翻译系统多是固定源和目标的系统，如DEC开发的VEST/mx和FX!32系统、IBM公司开发的Daisy和BOA系统、HP公司开发的Aries、Transmeta开发的Code Morphing软件、以及Intel开发的IA32 EL软件等。

2.2.1 FX!32 系统

DEC公司1996年研发的FX!32系统[53]，是一个轮廓信息指导的(profile-directed)二进制翻译器，目的是为了能将运行在x86/WinNT系统下的应用程序运行在Alpha/WinNT下。它结合静态翻译和动态解释，具有正确，高效而且透明的特点。FX!32第一遍执行时，用解释器进行解释执行，并将执行路径等信息保存下来，由另一个后台程序根据这些profiling信息进行静态翻译和优化，第二遍之后的执行，就开始部分使用翻译后的本地码，部分仍然需要解释，执行中仍然要生成profiling信息，与前面的信息合并，再用于静态翻译优化。通过这个循环过程，随着程序执行次数增加，速度日益加快。FX!32系统能够正确翻译WinNT下的Lotus, Excel, Word等实用程序，且使翻译后的代码在Alpha500上运行与Pentium200性能相当，真正达到了实用。但FX!32第一遍采用解释执行，因而第一遍的执行速度较慢，且依赖纯静态的profiling和静态翻译不能根据程序执行时的变化动态优化代码。

2.2.2 Aries 系统

HP公司1999年开发的Aries软件仿真器[54]，是一个基于软件的工A64转化设备。该系统结合快速解释和动态翻译两种翻译手段，可以仿真PA-RISC全部指令集，无需用户干涉。Aries只动态翻译经常使用的代码，仿真过程结束后放弃所有翻译的代码，而不修改原来的应用程序。因此，动态翻译既可提供快速仿真，又不破坏仿真的PA-RISC应用程序的完整性。快速解释和动态翻译相结合，用户就可以在运行HP-HX操作系统的IA-64机器上透明、准确、有效的执行PA-RISC应用程序。这种结合还可以得到比其它仿真方法代价更低，而性能更高的指令集体系结构仿真方法。代价低是因为这个过程中只翻译较少的代码;效率高则是由于翻译的代码比仿真的代码快。Spec2000整数测试表明，经Aries转化的程序在Itanium2 (1000M)机器上的性能约为本地码在PA-8700 (750M)机器上运行的0.5,浮点性能略差。Aries系统也是实用的商业代码转化系统。

2.2.3 Daisy 系统

Daisy和BOA系统是IBM公司分别于1996年和1999年开发的，虽然都用到二进制翻译优化技术,解决了诸如精确中断和自修改代码等二进制翻译通常会遇到的难题，但这两个系统的研究开发目标并不相同。Daisy是用于仿真现存体系结构的二进制翻译系统[55] [56]，以使旧体系结构上现存的软件(包括操作系统内核)可以在超长指令字(VLIW)体系结构下运行。VLIW结构设计简单而且指令发射率高，但却与现存的软件不兼容,使VL工W得不到真正使用,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，以后再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，动态解决了包括自修改代码、精确中断、内存一致性等问题。

2.2.4 BOA 系统

BOA动态翻译器系统[57] [58]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。

2.2.5 Code Morphing 软件

Transmeta公司2000年开发的Code Morphing软件[59]，使自己研发的芯片能够兼容X86的软件，利用二进制翻译技术开创了一个新的软、硬件开发模式。Transmeta公司生产的crusoe芯片，是由逻辑上被Code Morphing软件裹着的硬件引擎构成的。该引擎是个VLIW的CPU，其设计的出发点只是利于低功耗实现。Code Morphing软件能够把x86指令变成VLIW指令，从而使x86程序仿佛直接运行在x86硬件上。Crusoe芯片与Code Morphing软件的结合说明微处理器可以被当做软硬件的混合体来实现，软件部分的升级独立于芯片，硬件设计与系统和应用软件分开使得硬件设计师更新设计时不用担心影响遗留的软件。Code Morphing软件本质上是一个动态翻译系统。它驻留在ROM之中，是处理器启动后第一个执行的程序。Code Morphing软件支持x86代码，并且是用VLIW指令写的唯一一个程序。由于Code Morphing软件把x86程序(包括BIOS与操作系统)与crusoe芯片本身的指令集隔开，所以改动它本身的指令集一点也不影响x86软件。唯一要移植的程序是Code Morphing软件本身，这个工作由Transmeta完成，并且随每次体系结构改变做一次就够了。Code Morphing软件一直在透明的重编译和优化运行的x86代码。这种技术的代价是占用处理器运行时间，因而为了能具有好的系统性能，需要仔细设计Code Morphing软件来达到最大的效率，最低的开销。

2.2.6 IA32 EL 软件

Intel公司2003年开发的IA-32 Execution Layer (EL)软件[60],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。EL把IA32上的应用程序加载到翻译器自己使用的地址空间上，翻译器为了能够在多平台上运行，把操作系统无关的翻译算法放在一个操作系统无关的模块（BTGeneric)，而把库函数调用这种操作系统相关的封装到BTLib模块中，这两个模块之间通过约定好的API进行通信。IA-32 EL是两个阶段动态翻译的翻译器。运行时把翻译的代码缓存起来，一旦这个进程结束，就把已经翻译的代码扔弃。第一阶段的翻译，是冷代码的翻译，在这个阶段，要求翻译开销小，做了最少的优化，而且需要为第二阶段的翻译提供热点第二阶段的翻译，也就是热代码的翻译。冷代码的翻译，基本上以基本块为单位，一个基本块平均包含4-5条IA32指令.对于热代码的翻译，以热路径为单位，一般包含20条左右的IA32指令。虽然Itanium 2上也可以硬件执行IA32的程序，但现在IA32 EL的执行效率已经超过了硬件提供的执行效率。

上述FX!32、Aries翻译系统和IA32 EL软件都是在操作系统环境之上运行，着眼于应用程序的翻译；而Daisy、BOA、Transmeta系统则是对整个系统的翻译，包括应用程序、操作系统、以及其它特权级的指令[61]。

2.3 可变源和目标的二进制翻译系统

随着二进制翻译技术的发展，二进制翻译的研究又出现了新的方向，以澳大利亚的Queensland大学、奥地利维也纳技术大学为代表开始研究与机器无关的二进制翻译系统。他们期望通过使用机器描述刻画不同机器的特点，用工具为不同机器自动生成解码器、编码器、以及更高层的语义抽象单元，并且将部分优化抽象到与机器无关，从而降低二进制翻译系统移植代价，达到缩短二进制翻译系统开发周期的目标。

2.3.1 UQBT 和 UQDBT 系统

Queensland大学先后研究开发了可变源和目标的静态(UQBT[62][63][64])和动态(UQDBT[65][66][67][68])二进制翻译系统框架，引导了二进制翻译研究中一个新的研究方向。UQBT框架根据不同的二进制文件格式描述文件，自动生成文件编解码器；

根据不同的编解码描述文件[69]自动生成指令编解码器；还要根据不同的语义描述文件[70][71]，自动生成语义抽象转换器。从而使机器不相关部分的工作分离出来，给二进制翻译器编写者提供可重用的代码，使之仅需在独立于机器的抽象表示层上研究不同的变换优化算法，如果要移植到其它机器，则仅需重写描述文件即可。目前UQBT仅处理用户代码(应用程序代码)，而不处理内核代码或动态连接库。UQBT使用两种中间表示，机器相关的寄存器传输列表(RTLs)和机器无关的高层寄存器传输语言(HRTL)。将源程序通过文件解码、指令解码生成RTLs，然后语义抽象到HRTL高级表示，在此基础上即可进行机器无关的分析和转换，最后再由HRTL生成目标机器的二进制代码。UQBT项目证明了通过使用描述，低代价支持不同机器创建可适应性二进制翻译环境的可行性。写描述比写与机器相关的部分源代码无论是时间还是代码数量都少了很多，而且开发者还可以重用本系统提供的机器无关的分析。

UQBT在可重用二进制翻译器框架的搭建上是非常成功的，这在他们后期开发其他类型的前端和后端的实践中可以证明，可以耗费较少的人力和时间就完成了其它源、目标机器翻译系统的实现工作[72]。就效率而言，UQBT采用现有gcc或cc编译器的后端优化器，性能与本地代码执行相当，而UQDBT作为动态翻译系统，优化受到时间限制，因而UQDBT系统仅作了寄存器映射的优化，速度有2－6倍的下降。目前UQBT和UQDBT系统能够通过的测试例基本都是小程序，因而是一个很好的实验平台但不是实用系统。

2.3.2 Bintran 系统

奥地利维也纳技术大学研究开发的机器可适应(machine-adaptable)的Bintran动态二进制翻译系统[73][74]。它的最终目标是希望在不同机器描述协助下，能够实现所有的CISC，RISC以及VLIW体系结构之间的代码翻译。Bintran系统最主要的部分是翻译器的生成器，它能够根据源、目标机器的机器描述产生一个C语言写的翻译器。此外还包含机器无关的调度器和寄存器分配器。系统调用接口和内存管理模块主要是与操作系统相关，而与指令集体系结构的相关度较小。最后，Bintran系统还包括若干体系结构相关部分:目标汇编码写的用于从调度器到翻译生成代码之间切换的切换代码;解决一些特殊情况的代码(如变长指

令、目标机器寄存器少于源机器寄存器、不支持长立即数、以及条件码等情况)。与UQDBT使用三种描述语言不同,Bintran仅使用一种描述语言用来描述源和目标机器的语法语义，而加载不同的源二进制文件是手写的，并且Bintran不使用中间表示，而是用机器描述直接产生指令选择器。目前Bintran仅支持静态连接的ELF二进制码，模拟了Linux系统调用接口，支持PowerPC到Alpha体系结构，X86到Alpha体系结构的翻译，根据文献[73]提供的测试数据可知，速度大概比本地代码执行下降2.6－5.3倍。

2.3.3 Dynamite 软件

Transitive公司拥有的Dynamite软件技术[75]，是英国曼彻斯特大学的超过20年的研究成果。该技术可以使本地码和非本地码的程序都能无缝透明的执行。Dynamite软件分为三部分:原二进制码的解码(前端)、核心优化以及目标二进制码生成(后端)。前端和后端都是可以替换的，因而能支持多种处理器，这种结构还可以同时支持多个前端，因此允许多种二进制码流输入。前端接收源处理器的二进制码，翻译成IR, IR代码流用有向无环图(DAG)表示。核心优化是在Dynamite设计的IR上进行的，使翻译的程序可以以相等甚至高于本地码的速度执行。优化包括:识别基本块和基本块组(根据基本块间高频率的交互和跳转生成)；识别热点：当基本块和基本块组执行频率高于其它部分代码时，被判为热点，对其高度优化，进行分支预测，代码重排；死代码删除:根据程序上下文，生成特殊实例代码，删除不需要的指令。后端主要是生成目标二进制指令，同时还根据目标机的特点进行后端优化，如指令级并行，代码调度。

可变源和目标的二进制翻译系统目标是提供一个可重用的框架，提供某种中间表示并在上面进行优化，在开发某个具体的二进制翻译器时，可以重利用这些公共的代码，加速定制的二进制翻译器的开发。然而目前这些可变源和目标的一进制翻译系统，仅在一定范围的源和目标处理器中通过了一些简单测试程序，还没有一个真正实用的系统。

2.4 动态优化系统

动态优化技术既可以作为一种独立的代码优化技术，又可以作为二进制翻译

所必需的后端优化器。由于动态二进制翻译系统的翻译开销属于被翻译程序的运行时间，使得动态翻译没有时间充分分析和优化，因而在快速完成翻译之后需要对经常执行的热代码重新优化，这就需要动态优化技术的支持。

2.4.1 Dynamo 系统

HP公司开发的Dynamo[76][77]是一个动态优化器的原型。它的输入是本地二进制可执行代码，通过解释执行并观察程序的行为而不需要任何采样代码不需要对代码进行预分析，也不需要为以后的执行写出信息。解释执行程序时收集的profile信息帮助动态选择频繁执行的热路径(hot trace)，然后对这些热路径运用代码重布局、消除间接转移、以及一些常规优化技术，然后将优化生成的代码存放在一个软件cache中，当再执行到这些路径的时候就不解释而直接执行软件cache中优化后的代码，从而使程序的执行效率得到大幅度提升。

2.4.2 Java 虚拟机

Java编译器将java源代码编译成平台无关的Java byte code格式，然后被分发到各种平台，由Java虚拟机(JVM)实时解释执行。在高性能实现的JVM中，Just-In-Time (JIT)编译器实时地将Java字节码翻译成本地码，来减少解释执行的开销。由于翻译本身在程序执行期间进行，编译时间成为程序执行时间的一部分，因而编译中的优化需要考虑动态优化的特点，即需要在优化的代码质量和编译时间之间进行权衡。许多Java虚拟机中都采用了JIT编译，如Sun公司商业虚拟机JDK[78]、汉城国立大学和IBM合作开发的开放源码Latte VM[79][80]，Intel的开放源码MRL VM[81][82]。

需要注意的是，虽然动态优化可以很好地提高动态翻译生成代码的效率，但我们也不能忽视其自身的开销。因而用较低的动态优化开销换取关键路径上的高效代码，是动态优化系统大幅度提高代码执行效率的关键。

2.5 国内的研究现状

国内目前也已经有不少高校和研究所在从事二进制翻译技术的研究，并取得

了比较重大的突破：比如说中国航空计算技术研究所已经开发出了 BTASUP 系统[46]，把程序解释执行和动态翻译结合起来，在 PowerPC 处理器上实现对 1750A 处理器的二进制可执行代码的透明执行。中科院也成功开发了 Digital Bridge[47]动态二进制翻译系统。可以在 MIPS 机器上运行 X86 的程序。清华大学的 Skyeye[17]项目也在 ARM 的仿真器中添加了动态二进制翻译模块，显著的提高了仿真速度。

2.6 本章小结

本章概述了二进制翻译技术的发展和研究现状，详细介绍了固定源和目标的二进制翻译、可变源和目标的二进制翻译、以及二进制翻译中动态优化三类代表性系统

固定源和目标的翻泽系统都是公司为了特定目标开发的。如FX!32、 Aries、Code Morphing软件、IA32 EL软件等，是硬件公司利用二进制翻译技术解决新开发的处理器与老机器代码的兼容问题;Daisy和BOA则是为设计新的处理器体系结构服务,Daisy希望能够使用二进制翻译技术将PowerPC机器代码调度到VLIW体系结构，而BOA则希望利用二进制翻译技术降低硬件设计的复杂度。这类系统一般都解决了二进制翻译中的代码挖掘、精确中断、自修改等问题，基本都可以实用。

可变源和目标的翻译系统以UQBT和UQDBT系统，以及Bintran系统为代表它们都是研究机构开发的，以研究二进制翻译框架和可移植性为目的，为二进制翻译的研究指出了一个新的方向。但这类系统都是研究性质的，能通过的程序有限，并没有达到实用。

动态优化系统则以Dynamo系统为代表，通过监测程序执行行为，提取热代码路径并进行优化，对某些程序达到了很好的效果。动态优化既可以作为静态编译的补充，动态提高代码执行效率；也可以作为动态二进制翻译的后端优化器，提高二进制翻译系统的效率。

二进制翻译及相关优化技术的研究，引起了众多公司和研究机构的重视，已经成为是现代编译技术研究的热点之一，是非常有前景的研究方向。

第三章 QEMU 翻译系统简介

QEMU 系统是目前较为先进的支持多源平台的二进制翻译系统。其基本设计结构如图 3-1 所示，系统可以实现将 arm（同时支持 arm7 与 arm9）、x86、MIPS 下的 ELF 格式的可执行文件翻译到中间代码（由低级 c 实现），然后编译到目标 ISA 上，如 x86 的 IA-32 上运行。

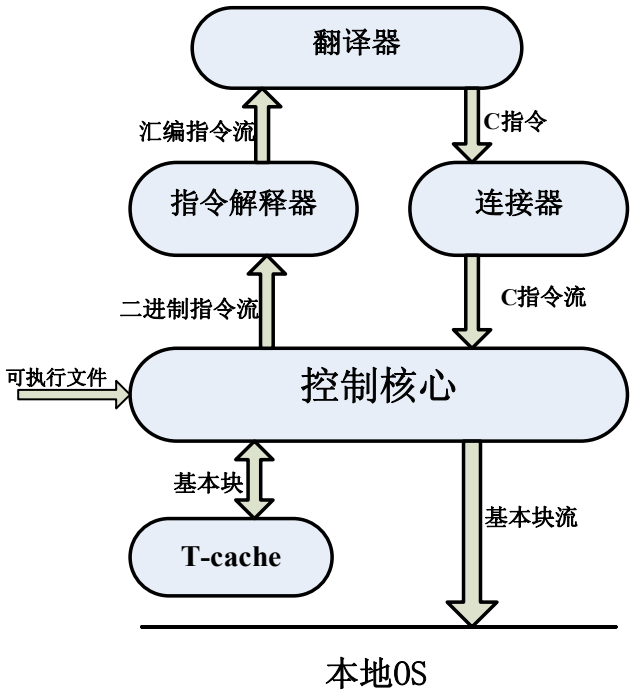

图 3-1 QEMU 系统基本结构

Figure 3-1 Framework of QEMU system

系统由控制核心、解释器、翻译器、编译器和翻译缓存等几部分组成。运行时，控制核心会维护一个软件的目标机虚拟 CPU 状态，称为 env，它包括通用寄存器、段寄存器、标志位寄存器等。目标机的所有资源都通过 env 基址加上特定的偏移来访问。首先由控制核心对进入的 arm 下的 ELF 文件进行加载，这个过程由一个类似 Linux 下的 load_binary 函数的 loader 完成，实现区分文件中的各个段、找出符号表、并且返回文件执行起始位置。这样就可以得到文件执行时的代码段入口（或者理解为一个二进制指令流）。然后这些二进制指令流由指令解释器处理识别，分解成单条的汇编指令，形成汇编指令流送入翻译器。翻译器将汇编指令翻译成等价的微码片段（由 C 函数实现），这些 C 函数事先已经被编

译过，编译后的二进制码称为微操作码。最后由动态链接器将这些微操作码连接起来，生成可以在目标机器上连续运行的二进制指令流。

3.1 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。同时具有用户级和系统级两种的模拟功能：不同的模式会导致被应用的层次的不同。用户级二进制翻译系统提供了一个虚拟的操作系统，如图 3-2（A）所示。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。一个用户级的二进制翻译系统实际上就是一个运行独立进程的平台。它只支持支持进程运行，在进程运行时创建，而且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 或者图形化用户接口。

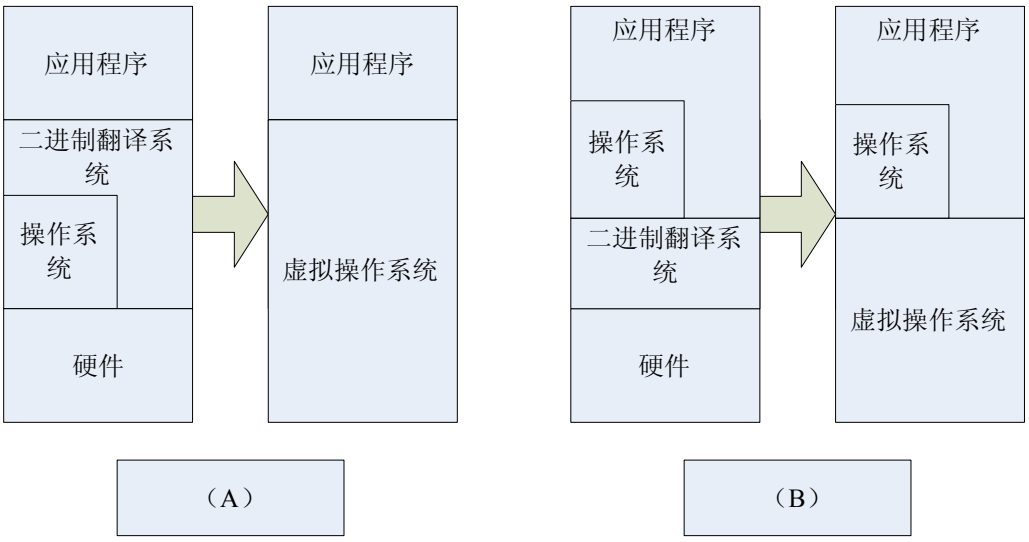

图 3=2 用户级和系统级二进制翻译系统结构

Figure 3-2 User Level and System Level binary translation framework

这种方案通常只模拟应用程序所能看到的部分，主要包括系统调用接口和源机器指令集。在不同的操作系统的系统调用存在比较大的差异，它们可能具有不同的系统调用编号，不同的传递参数方法；也可能在宿主机操作系统上无法找到对应的源机器操作系统的系统调用。用户级二进制翻译系统必须正确且高效地处理系统调用。用户级模拟的例子比较多，比如说 Bintrans[18]和中科院开发的

Digital Bridge[47]等等。也有一些系统单独做系统调用接口的翻译，比如说：FreeBSD 可以在 i386 系统上运行 Linux i386 的二进制代码;WINE 做得更加彻底，它并不是模拟 Windows 系统调用接口，而是提供了系统库（DLLs）的具体实现

系统级的二进制翻译系统提供了一个虚拟的机器，如图 3-2（B）所示。源机器的操作系统运行在这个虚拟的机器上。这在一般意义上被说成是虚拟机：比如说 Virtual PC，Vmware， Bochs 等。模拟一个完整的源机器硬件，往往包括了硬件的内存管理单元（MMU），一些外部设备，例如显卡、网卡、声卡等。如果要让程序能跨平台运行，还需要模拟源机器的指令集。从操作系统和其支持的应用程序的角度来看，它们在这个机器上运行。这个机器将提供完全的运行环境，可以支持大量的进程在其上运行。这些进程共用一个文件系统和几个 I/O 资源。这个机器可以为这些进程分配真实的内存空间和 I/O 资源。操作系统无法识别这个机器和真实的机器的差别，事实上等于它提供了一个操作系统与机器之间的接口。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。如果不是跨平台运行程序，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。而特权指令却需要特殊处理，因为被模拟的操作系统往往都运行在用户模式下，而用户模式不具备特权指令的执行权限，只能通过宿主机的操作系统内核才能执行。所以这些特权指令要用一些特定的代码来实现，这些代码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一定的检测，确保安全的情况下才能允许执行。

这两种模式相比而言各有利弊，系统级模拟可以是让操作系统安装在不同体系结构的硬件上，而操作系统上的用户根本不会也没有必要意识到这一点。而用户级模拟可以使用户程序更好的和宿主机的运行环境结合在一起，能够非常高效的模拟操作系统的系统调用，而且避免了模拟全部硬件，和硬件相关的操作都由宿主机的操作系统来实现。所以通常来说用户级模拟往往比较简单而且高效，在源机器操作系统和宿主机操作系统是同一个系列的时候，优势更加明显。

本文主要讨论的是用户级的二进制翻译系统，值得一提的是二进制翻译系统既可以直接运行在硬件平台，如 IBM 的 Daisy 和 Transmeta Crusoe 上的 morphing 软件层，也可以运行的宿主机的操作系统上，比如说上面提到的 QEMU，

Bintrans 等等。本文主要讨论后者。

3.2 QEMU 的翻译单位

区别解释器、动态二进制翻译系统、静态二进制翻译系统的最主要标志是翻译单位的选取，即一次翻译的代码长度。如果每次翻译一条指令然后立即运行，那么就是解释器。传统的解释器的主循环逐条取指令，分析，根据指令的操作码进入一个庞大的 switch-case 结构寻找并执行对应的处理函数，最后回到主循环。传统的解释器由一系列函数调用组成。如果将翻译单位拉长到整个被翻译程序，则成为了静态二进制翻译系统，其具体细节请参考 1.2.1 节。当翻译单位选取在解释器和静态二进制翻译系统之间时，就称为动态二进制翻译系统，其具体细节请参考 1.2.2 节。

解释器的优点是实现方法简单，能够精确到指令级的模拟，而且具有可移植性。其缺点也是显而易见：调用函数的开销非常高；而且解释器不保存解释结果，当碰到相同的代码段时依旧要重新解释执行每一条指令；此外由于翻译单位是单条指令，缺少代码优化的可能性。各种原因都导致解释器的效率很低，代码膨胀率高达 100:1。静态翻译器的优点是离线翻译程序，有足够的时间进行优化，效率较高；但是静态二进制翻译有其局限性，现在使用的计算机几乎都还是采用冯·诺依曼结构，指令和数据存储在一块，并以相同的方式表示。因此，要静态地发现一个程序中所有的指令代码一般是不可能的。而且有些信息在静态时是处理不了的，例如，间接分支、共享库、自修改代码以及精确异常状态等。可见静态翻译不能完全处理程序，它需要依赖解释器的支持，开销巨大，而且需要终端用户的参与，缺乏透明性。

$$\text{代码膨胀率}=\frac{\text{翻译产生的代码长度}}{\text{源机器代码长度}}$$

目前大部分的动态二进制翻译系统都采用基本块作为翻译单位，包括 QEMU 系统。所谓基本块是一段只有一个入口一个出口的程序段，通常一个基本块包括了 4 到 7 条指令。选择基本块做翻译单位，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。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

更多的优化空间。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。采用基本块作为翻译单元，动态二进制翻译器都会把翻译和优化的结果保存起来，下次遇到相同代码段的时候，就可以直接执行预存起来的翻译后代码。

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[25]的技术，省去了函数调用的开销和堆栈的维护开销。

为了能无缝连接微操作，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。编译时将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、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。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，在设计微操作函数时不使用参数和局部变量。然而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。有些常量必须在执行的时候才能确定，而且还和不同的体系结构相关。QEMU 的微操作函数使用全局变量来为这些常量预先占好位置，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 buffer 中的数据拷贝到原来全局变量的引用位置。并在翻译产生的代码中以立即数的形式出现，以此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，并通过一定的技巧去除函数的末尾的返回指令。以此来获取微操作函数中真正有意义的代码。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。QEMU 是以基本块为单位进行翻译的，所有它也只能精确到基本块，比如说对于 PC 寄存器，QEMU 并不是每执行完一条源机器指令就更新 PC，而是在基本块结束的时候才更新 PC，这可以节省很多不必要的开销。

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 QEMU 的翻译单元依旧是一条条指令，源机器的指令被分解成若干个微操作，

通常这些微操作都是由简单的 c 语句组成，宿主机的编译器对这些微操作分别编译，优化的空间非常狭小。这使得翻译生成的代码非常低下。QEMU 的实现从 ARM 到 X86 的翻译的时候，需要 5.8 左右的代码膨胀率。代码膨胀率是二进制翻译系统中非常重要的性能指标，ARM 到 X86 的翻译属于 RISC 到 CISC 系统的翻译，X86 的代码密度要远远大于 ARM 的代码密度，从理论上说从 ARM 到 X86 翻译的代码膨胀率应该控制在 1 以下。然而 QEMU 却需要 5.8 左右。当然 QEMU 的代码膨胀率居高不下还有其他原因，但是把单条指令作为最小的翻译单位是最为重要的因素。

3.3 QEMU 的翻译策略

所谓翻译策略，就是确定什么时候对翻译后的基本块进行优化。显然，最简单的办法就是从不优化或者对每次遇到的没有翻译过的代码在翻译时都进行优化。从不优化当然不是我们所需要的答案，然而优化是有一定的开销的，有些代码可能只会碰到一次，那么对他们做优化就显得得不偿失了。因此，系统需要能够找出值得进行优化的部分。现有的动态二进制翻译系统一般在基本块之间采用的类似于解释执行的方式，即翻译一个基本块执行一个，有当某个基本块被反复执行时，才单独对该基本块实施进一步的优化，然后将优化后的结果存入 T-Cache 中，以备再次执行到该基本块时使用。QEMU 采用的就是这种处理办法，当遇到没有还没有翻译的基本块时，就先翻译这个基本块并存入 T-Cache 中，然后再执行。

研究人员在设计 Dynamo 翻译系统的时候提出了热路径概念。一个热路径是一组很有可能被连续执行的指令系列，它是若干个基本块的结合体，因此也被称作是：具有一个入口，多个出口的特殊基本块。热路径跨越了多个基本块，为优化提供了更加广阔的空间，可以充分挖掘基本块之间的并行性，延迟分支跳转处理等。热路径内的所有基本块都被放入一个连续的地址空间，这样可增强生成代码的空间局部性，从而提高 Cache 的命中率，同时还可以减少跳转指令数。采用热路径作为翻译单位的二进制翻译系统只有在发现一条热路径的时候才对代码进行翻译和优化。这么做的目的是生成高效的宿主机代码，提高二进制翻译的性能。

3.4 QEMU 的运行流程

用户级的 QEMU 的运行流程如图 3-3 所示：当 QEMU 启动后，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。遇到系统调用的时候则翻译源机器的系统调用，并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期望的功能。当源机器程序结束的时候，QEMU 也同时退出。

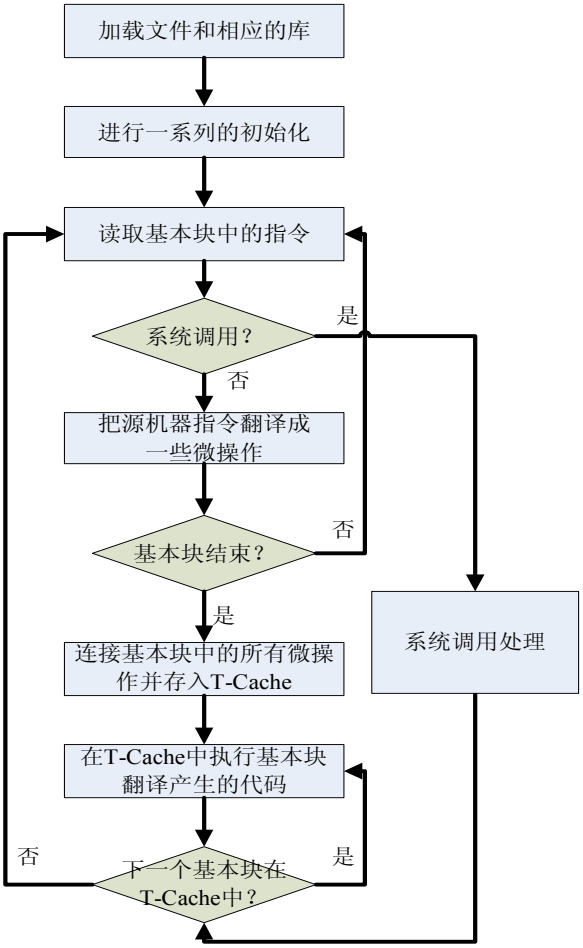

图 3-3 QEMU 的运行流程

Figure 3-3 Framework of QEMU

3.2.1 加载被翻译程序和初始化

由于整个系统是用户级的，所以系统在运行时使用和翻译源程序相同的虚拟地址空间，这样就产生了潜在的地址冲出的问题。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。这样的处理方式是建立在被翻译程序不会使用很高的地址空间的假设上的，对于 Linux 由于提供了 4G 的虚地址空间，所以这个假设一般都是成立的。如果假设不成立，需要在翻译源程序加载时动态判断其要加载的空间是否已经被使用，如果已经被使用就需要重新定位。

加载过程有控制核心完成。具体的加载过程是分为两步的，一是打开文件检查属性，二是将文件加载到指定位置。而整个过程的核心内容是识别翻译源文件的结构。

本系统使用类 UNIX 系统下通用的可执行文件格式 ELF 作为其翻译源文件[3]。ELF 文件可以从两个角度去分解，一是从文件链接的角度，文件由一个个 section 组成，二是从程序执行的角度，由 segment 组成，如图 3-4 所示。在整个结构中只有 ELF header 的结构和位置是固定的，其他各段的起始位置都是可变的，但信息会被保存在 Section header table 或 Program header table 中，这个两个表的信息记录在 ELF header 中。

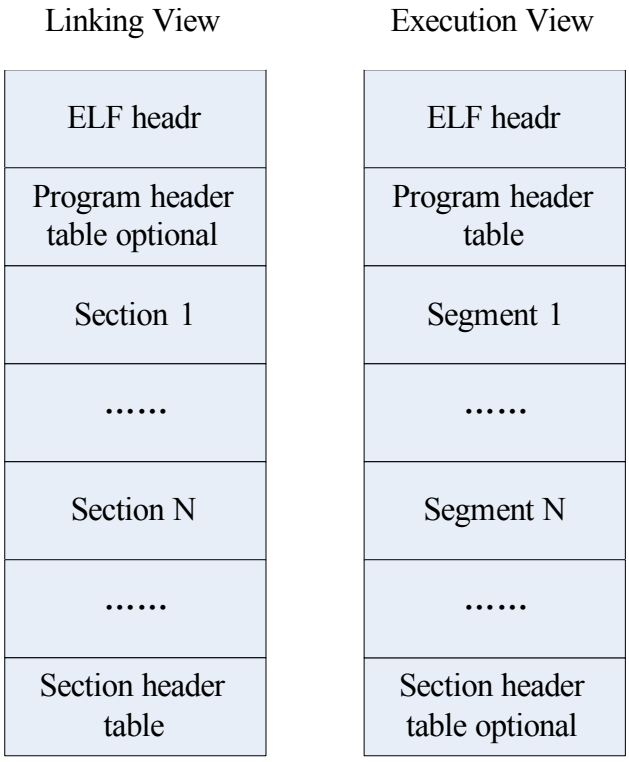

图 3-4 ELF 文件格式

Figure 3-4 Format of ELF file

加载时，控制核心中会生成一个数据结构体 linux_binprm（如下所示）用于保存文件的基本参数。记录 linux_binprm 的过程中，就可以进行检查文件属性与

权限。

```
struct linux_binprm {
    char buf[128]; //用于保存 elf 的文件头
    unsigned long page[MAX_ARG_PAGES];
            //Pages 用于保存参数
    unsigned long p;
            //pages 能放的 int 类型的个数
    int sh_bang;
            //表示是否可执行文件。
    int fd;      //文件头
    int e_uid, e_gid;   //文件的用户和用户组
    int argc, envc;
    char * filename;   //文件名
    unsigned long loader, exec;   //加载器
    int dont_iput;
}
```

确认文件属性和权限之后，控制核心就可以分解 ELF 文件。从 linux_binprm 中可以得到ELF 文件头，从中可以确定 Program header table 的位置，读出 Program header table 就能知道各个 section 的起始位置和段名，然后对应其加载位置进行映射。最后返回翻译源程序的入口地址。

3.2.2 主要翻译过程

QEMU 动态二进制翻译的核心主循环的流程如图 3-3 所示。QEMU 用 C 语言设计一系列微操作，这些微操作在编译 QEMU 的时候，被编译成了宿主机的二进制代码。而在运行源机器程序时，QEMU 先把每条源机器指令分成若干个微操作，然后通过编译连接技术把这些编译后的微操作无缝地拼接起来并存入 T-Cache 中。通过执行 T-Cache 中的代码来完成源机器代码的语义。进入每一个基本块之前，QEMU 都会先判断该基本块是否已经被翻译，如果已经翻译就可以直接执行 T-Cache 中相应的代码。

翻译的过程主要由指令解释器、翻译器和连接器完成，由于整个翻译系统采用动态机制，翻译时还需要控制核心的调度。

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。这个里面存在一个问题，就是 C 函数本身是有输入参数的，而参数的决定势必使得程序段间不可能无缝的拼接起来。本系统的做法是所有 C 函数本身都不具有参数，而把程序所需立即数和跳转地址储存在由控制核心维护的一个堆栈中。在拼接的时候把参数取出，写入代码之中。这样就既实现了程序的无缝拼接又解决了参数输入问题。

例如对于以下的 arm 指令：

```
add r3 r3 16            # r3 = r3 + 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 采用这些中间变量可以大大减少微操作数量[16]，降低系统设计难度。微操作不再包括 32 个 PowerPC 寄存器之间的 mov 操作，而只有它们和 3 个中间变量之间的 mov 操作；而且几乎所有的运算操作都只在 T0，T1，T2 上进行，微操作中不存在以源机器寄存器为操作数的运算操作。

和绝大多数二进制翻译系统一样，QEMU 也是以基本块为单位进行翻译执行的。QEMU 保存所有已翻译基本块的基本信息，包括基本块的起始地址、长度以及对应翻译后代码在 T-Cache 中的位置，同时还包括和其他基本块之间的连接信息等等。此外 QEMU 还设计了一个 hash 表，用于快速的判断以某个 PC 为起始的基本块是否已经被翻译，并确定翻译后代码的位置。

3.2.3 对于系统调用的处理

正如上面所说系统调用是用户级模拟需要解决的一个重要问题。QEMU 在源机器代码中遇到系统调用时，先结束当前的基本块，并保存该系统调用的下一条地址作为后续基本块的首地址；然后把系统调用翻译到宿主机操作系统的系统调用，并通过后者来为源机器程序服务；系统调用执行完后回到 QEMU 的主循环，从系统调用的下一条指令开始继续翻译执行源机器程序。QEMU 利用 setjmp 和 longjmp 调用来处理系统调用，其详细实现过程如图 3-5 所示：

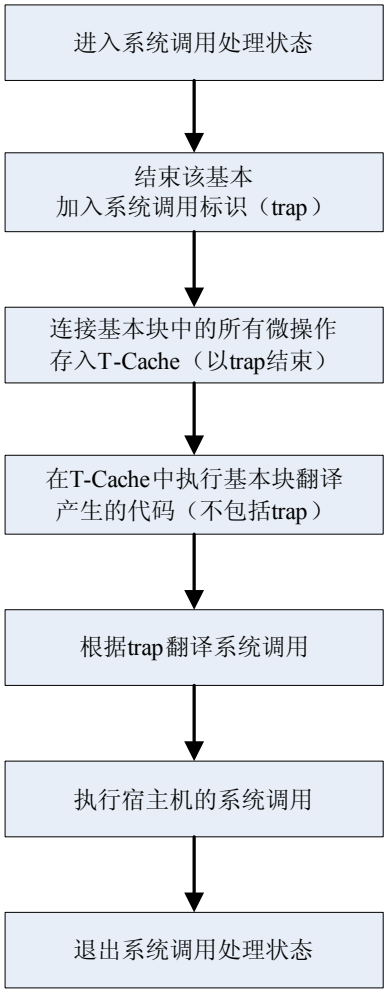

图 3-5 QEMU 系统调用的处理流程

Figure 3-5 Process of System Call in QEMU

当程序第一次进入CPU_EXEC时，setjmp返回0，不过此时并没有发生陷入，QEMU进入翻译执行基本块的循环体。如果基本块还未翻译，则必须先翻译。在翻译时如果遇到了系统调用则结束该基本块，把系统调用翻译成设置陷入号和调用longjmp（jmpbuf,1)两个操作。在执行以系统调用结尾的基本块时就会调用longjmp（jmpbuf,1)，从而回到setjmp，此时setjmp的返回值为1，执行else并回到大循环，再次执行setjmp。此时setjmp返回0，而且发生了陷入，CPU_EXEC返回陷入号给CPU_LOOP。CPU_LOOP通过以下三步处理系统调用：

1. 根据源机器的寄存器的值确定是哪个系统调用;
2. 把系统调用翻译成宿主机的系统调用，并做一定的修改;
3. 调用宿主机的系统调用完成源机器程序期望的功能。

CPU_LOOP 处理完系统调用之后又再次进入 CPU_EXEC,如此周而复始。

QEMU 的用户级模拟限制在 Linux 操作系统上，在这种情况下，可以快速的

把利用宿主机的Linux系统调用来实现源机器程序的Linux系统调用。一般情况下都可以直接调用宿主机的系统调用，只有个别系统调用需要特殊的处理，如fork等。值得一提的是对于exit系统调用的处理，一般程序的末尾都是通过调用exit来退出的。QEMU在处理源机器程序中的exit系统调用时，把它翻译成宿主机上的exit，把自己直接退出了。这种做法虽然看起来QEMU是非正常结束，然而这正是用户级二进制翻译系统所希望看到的——在系统上的负载结束的瞬间，系统自身应该马上退出。

然而用户级的程序并不总是直接调用系统调用，现代的操作系统上面，往往都会提供一系列的库，而应用程序也会经常使用这些库。二进制翻译系统没有办法处理源程序中的库函数调用。当然也可以像处理系统调用一样把这些库函数的调用翻译成宿主机操作系统上的库函数，这样可以获得非常高的执行效率。但是这么做会大大增加二进制翻译的负担，二进制翻译不仅要处理这些库的对应关系，而且要处理库里面的每一个函数的映射。QEMU 使用比较简便的办法，他为不同平台提供了一些经常使用的库，这些库和应用程序一块被加载。源机器程序执行的时候可以直接使用这些库，如果库函数没有使用系统调用，那么就像程序中的普通代码一样翻译执行，只有遇到系统调用的时候才翻译成宿主机操作系统的系统调用。

第四章 二进制翻译中的跳转优化

4.1 引言

在计算机的发展过程中，软件与硬件之间的兼容性问题一直都是困扰设计人员的一个十分棘手的问题。而二进制翻译就是解决这个问题的一种行之有效的方法。二进制翻译（Binary translation）是将一种指令集体系结构（Instruction Set Architecture, ISA）上运行的代码转化成另一种指令集体系结构的指令的技术。

现有的二进制翻译系统共同面对的一个问题就是运行效率。在动态二进制翻译中以基本块为单位进行翻译和执行，而一般程序中平均每 4－7 条指令就存在一个跳转，而跳转破坏了程序的“本地性”，产生了大量相应的调度开销。因此如果可以对基本块间的调度，即跳转处理进行有效的优化，减少其间的转换时间将有可能极大的提高系统的运行效率。

本人研究了 QEMU 动态二进制翻译系统对从 arm 到 x86 的翻译中使用的跳转控制指令处理技术，以及对其进行的优化，包括直接跳转和间接跳转。然后通过对于实验数据的分析说明了优化的效果与其中存在的问题，并提出改进方法。

4.2 QEMU 中的跳转控制处理

基本块是 QEMU 系统中的基本执行单位，根据翻译机制，在执行完一个基本块后，需要判断下一个基本块是否已经被翻译。如果基本块已经被翻译了，那么就直接执行 T-Cache 中的翻译后基本块，否则必须翻译基本块，然后才能执行。这种方法能够非常准确的模拟翻译源程序的行为，也是通常动态二进制翻译系统采用的做法。然而根据统计数据，基本块的平均长度只有 4 到 7 条指令。换句话说，程序平均每执行 4 到 7 条指令就需要从执行状态切换出来，判断，然后再进入执行或翻译状态，过于频繁的状态切换和判断是一笔巨大的开销。

在 QEMU 系统中跳出执行状态时会首先保存执行现场，包括把虚拟的寄存器信息保存到内存中的特定位置；然后找到下一条指令的位置，并通过这个位置判断下个基本块是否已经被翻译过；最后继续翻译或执行。所幸的是 QEMU 系统是把翻译部分和执行部分放在同一个进程来实现的，否则这个开销还要包括翻译进

程和执行进程之间的上下文切换开销。

基本块最后一条指令的类型不同，其连接开销也不同。指令的类型可以分为普通指令、跳转指令、分支跳转指令、调用指令、返回指令、间接跳转指令和间接调用指令。一般来说，普通指令、跳转指令只有一个后继，其连接只需要一条跳转指令即可:分支跳转指令需要增加条件判断，并根据条件跳转到不同目标地址;调用指令本身的处理类似跳转，但为了方便返回指令的处理，可以增加一个函数调用的辅助栈，在栈中存放调用指令的返回地址。间接跳转和间接调用由于没有固定的转移目标，其处理就更加复杂，本系统采用了缓存部分转移目标，加速查找的方法。综上所述，代码连接的开销并不固定，通常需要几条到几十条微指令，这比经过上下文切换和基本块目标的查找开销，有数量级的降低。

在 QEMU 系统中跳转控制指令是标识基本块结束位置的指令之一，而控制跳转指令按照目标是否确定可以分为两种：分支目标在编译时就可以确定的控制跳转指令，比如说非常普遍的直接跳转指令，包括条件跳转指令和无条件跳转指令；分支目标只有在执行的时候才能确定的控制跳转指令，比如说间接跳转的目标就只有在执行到该指令的时候才能确定，而且每次执行的跳转目标不一定相同，例如函数调用时的 return 指令。

由于直接跳转和间接跳转目标地址的不同特性，本人采取了不同的方案对其分别进行了优化，并在 QEMU 系统上进行了实现和性能测试。

4.3 直接跳转优化

在 QEMU 系统原有的直接跳转处理方法中当翻译器识别出跳转指令后就将标识当前基本块结束，然后在其后加入跳转目标在翻译源程序中的地址，这样就最简便的解决了对跳转指令的处理。但这样所带来的问题是无论下个基本块是否被翻译过，在进入之前都要返回翻译源程序对其进行判断。频繁的状态切换将大大影响系统执行效率，因此需要对系统这部分处理方式进行优化。

4.3.1 优化方案设计

优化的主要思路是基于在直接跳转的情况下，跳转目标是确定的且不发生变化，所以当系统发现跳转目标基本块已经被翻译时，可以直接把跳转的源基本块

和跳转目标基本块在 T-Cache 中连接起来。

这个优化的具体方法为修改源基本块的末尾部分，增加一个翻译后的跳转指令，让其直接跳转到已经翻译好的目标基本块处继续执行，而不再回到翻译程序中进行判断。也就是将前后两个基本块连接起来，从而跳过原来在其间的判断目标基本块是否已被翻译、保存寄存器值信息等操作。不过在这种情况下，系统的翻译精确度已经无法达到基本块级了，因为 PC 不再是每个基本块都更新一次，只有在遇到没有被翻译的基本块时才更新 PC。在理想的状态下，二进制翻译器大部分时间都将在执行翻译后的代码，也就是说沿着修改翻译后程序所组成的基本块链，即热路径，执行。

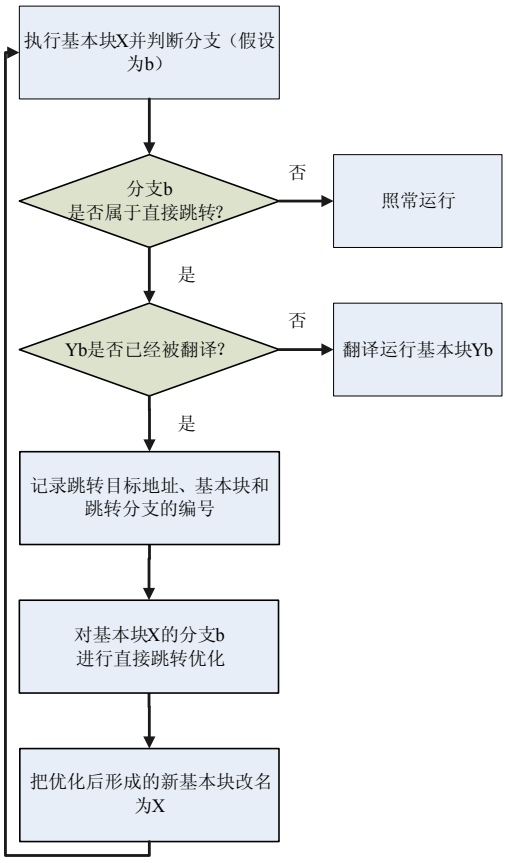

图 4-1 直接跳转优化的设计方案

Figure 4-1 Design Plan of Direct Jump Optimization

一般的程序的分支处理通过三个步骤来实现，首先判断一个条件，并设置相应的标志位；然后检查对应的标志位，如果标志位符合指令要求，就跳转到目标地址，即 J_PC 所指位置；否则程序继续执行，即从 PC＋1 开始。在处理这种分支过程的翻译时，我们分别设计两个分支出口，如图 4-2 所示，为 Jmp0 和 Jmp1

两个分支出口。其中设定 Jmp0 处理目标为 J_PC 的分支，而 Jmp1 处理目标为 PC＋1 的分支。每个分支的处理过程都主要有以下四个步骤组成：首先预先设置一个跳转指令，目标先设定为 PC＋1 以保证程序正确执行。直接跳转优化就是通过修改这条指令来实现的；然后把分支目标地址保存到模拟的 PC 寄存器，翻译程序通过该寄存器来获取目标基本块的首地址；之后再把基本块和分支的编号保存。在实际设计的时候还需要运用了一定的技巧：基本块编号其实是存放该基本块信息的数据结构的首地址，这个首地址是后 2 位对齐的，后面两位可以用来标识分支的编号。因此实际上保存的并不是两个数据，而是基本块信息的首地址跟分支的编号的和；最后返回翻译部分，继续执行[83]。

4.3.2 实现过程

优化后的跳转处理在遇到直接跳转指令后会判断其跳转目标所在基本块是否已经被翻译，如果已经被翻译则将原来保存的跳转目标在翻译源程序中的地址修改为跳转目标在 T-cache 中的地址。这样当程序再一次执行到这个跳转指令时会连续执行下一个基本块而不用进行切换和判断。

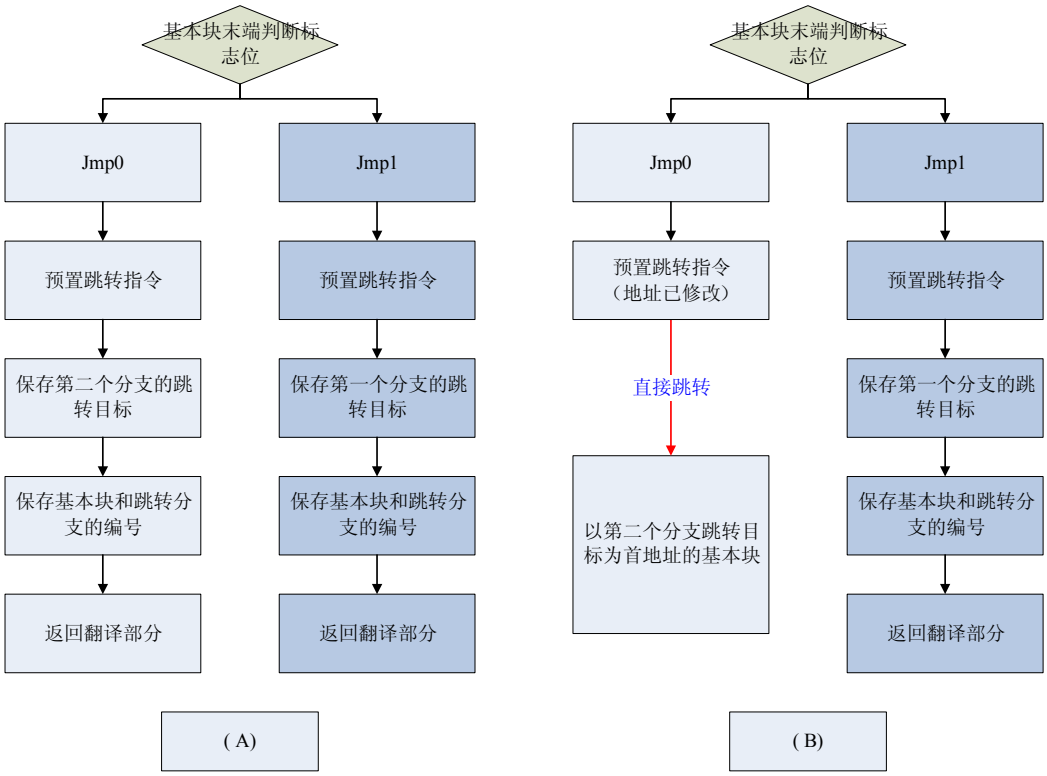

图 4-2 直接跳转优化的实现前后程序流程

Figure 4-2 Program Flow before and after Direct Jump Optimization

实现后的二进制翻译系统的优化过程可以参考图 4-1 图。执行基本块 X 的末尾会判断根据当时的执行情况来判断是选择 Jmp0 还是 Jmp1 分支。假设程序选择了 Jmp1 分支（称作为分支 b，目标基本块称作是 Yb）。如果分支 b 已经是做过直接跳转优化了，就可以通过 Jmp1 里的埋伏的 Jmp 指令直接跳转到翻译后的 Yb 的首地址，直接执行 Yb 回到循环的开始。否则必须记录 Jmp1 的跳转目标，即 Yb 的首地址，同时保存基本块 X 和分支的编号。然后返回到翻译部分判断 Yb 是否已翻译，如果 Yb 没有翻译则先翻译。当翻译好的 Yb 被放到 T-Cache 时就可以修正基本块 X 的分支 b 了，也就是直接修改 Jmp1 中 Jmp 指令，让它直接指向翻译后的 Yb 的首地址（如图 4-2（A）图）。最后再执行翻译后的 Yb，回到了循环的开始部分。当程序后面再次选择分支 b 的时候，就可以根据刚刚修正的跳转指令，直接从基本块 X 跳转到 Yb（如图 4-2（B）图）。

这里需要特别指出的是，并不是说任何以翻译后的基本块为目标的跳转，都一定预先做好直接跳转优化。笔者的做法是使得只有在程序选择一个分支执行后，才对这个分支做直接跳转优化。当程序再次选择这个分支，就可以享受直接跳转带来的好处。

4.4 间接跳转优化

间接跳转同直接跳转在跳转目标地址产生上有着明显的不同，直接跳转的目标地址是相对固定的，并事先知道；而间接跳转的目标地址只有在执行到跳转指令的时候才能确定，它往往取决于某个寄存器或变量的值，而且每次执行到同一个跳转指令的时候跳转的目标地址还可能不一样，所以在前面直接跳转优化中的方法并不适用于间接跳转。于是必须使用其它的实现方式来进行间接跳转优化，我们对最普遍的间接跳转指令 return 进行了如下优化处理。

在 arm 指令中，一般情况下函数的调用由调用者的 call 指令实现，而由被调用者的 ret 指令来实现调用后的返回。在 Call 指令的动作中会把下条指令位置（PC+1）作为返回地址保存起来，然后才跳转到被调用者的程序部分。而被调用者的 return 指令会使程序返回先前所保存的位置。因此理论上可以在任何地方调用任何函数，而且还可以发生任意次的函数的嵌套调用。

我们的优化方案是，如果某个 call 指令的 PC＋1 位置是一个已经翻译的基本

块的起始地址，那么按照之前在直接跳转优化中使用的连接基本块的思想，return 指令应该可以直接跳到翻译后的基本块。具体实现方法为，在 QEMU 翻译 call 指令的时候做一定的修改，如果发现以 PC+1 为起始的基本块已经被翻译，就直接修改 call 指令，不再保存 PC+1 为返回地址，而是保存翻译后的基本块在 T-Cache 中的首地址，否则仍旧直接保存 PC+1。另外，在 QEMU 系统中 T-Cache 的地址空间和翻译源程序存放的地址空间是相互独立的，在翻译 return 指令时可以正确地区分出对应 call 指令保存的地址。如果在源程序地址空间内则按照原来的处理方法，否则 return 指令通过该保存的地址，直接跳转到翻译后的基本块。这种优化的前提是在调用 call 到执行 return 之间，程序不会修改调用者的返回地址。然而事实上完全可以编写出修改调用者返回地址的程序，而且由于允许函数的嵌套调用，return 和对应的 call 可能横跨不少函数调用，这样就更加难以保证 call 保存的地址不被修改了。不过，在一般的体系结构下，都认为这种操作是非法的，也是编译器和程序员所应该避免的，所以这种优化方法依然是可行的。

4.5 实验结果及分析

我们选择的目标平台的配置是 3.06GHz Pentium 4 处理器、1GB 内存的机器，运行 redhat9 操作系统，编译器采用 Crosstool[28]提供的 gcc 编译器。测试集为包括 7 个程序集的 nbench benchmark suite[27]（QEMU 官方网站推荐用于评测性能的测试程序）。结果为五次测试的算术平均值，数据如表 4-1 所示。

表 4-1 使用 nbench 的测试数据（单位 Iterations/sec）

Table 4-11 Test result (Based on nbench, Iterations/sec)

	Numeric sort	String sort	**B**itfield	FP emulation	fourier	Idea	Huffman
原系统	101.24	10.668	3.146e＋07	8.7859	80.272	239.62	85.587
直接跳转优化	124.49	16.521	5.658e＋07	13.756	83.287	517.11	140.89
间接跳转优化	101.00	10.479	3.142e＋07	8.8011	80.504	249.81	85.940

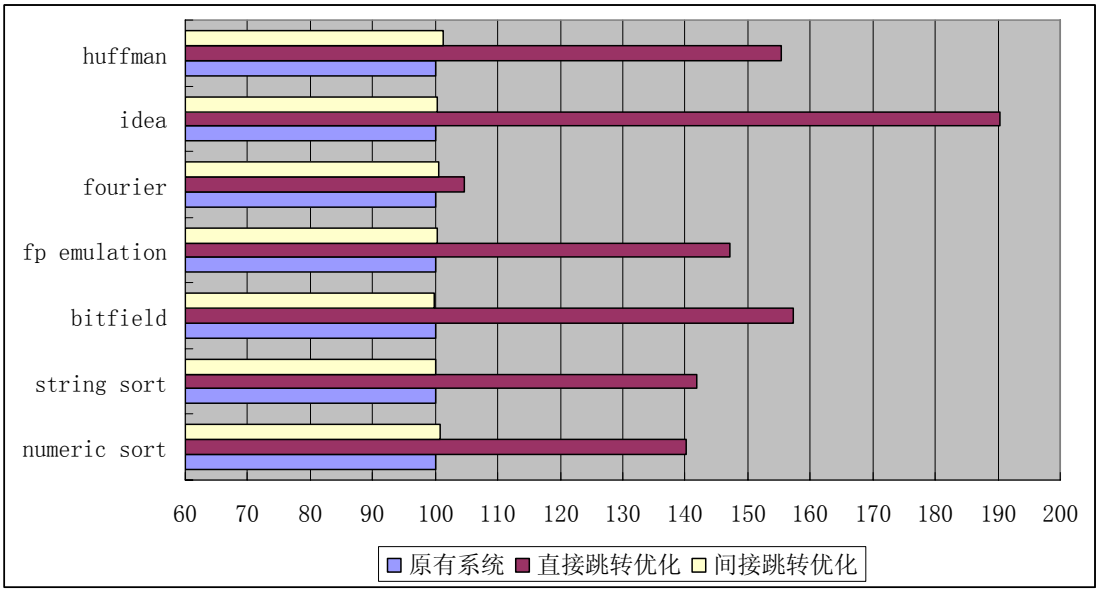

图 4-2 跳转优化对系统的性能影响

Figure 4-2 Direct Jump Optimization's Effect

在图 4-2 中的所有执行速度都是相对值。由图中可以看出，对于大部分 benchmark，直接跳转优化都可以给翻译系统带来 0.5 倍的性能提高，优化后的翻译几乎能够以两倍的速度翻译执行“idea”benchmark。

可见直接跳转优化能够给单进程设计的本翻译系统带来显著的性能提高，而间接跳转优化却几乎未产生任何性能的提高。经分析，笔者发现由于间接跳转的目标只有在执行到的时候才能确定，每次翻译执行间接跳转指令时，都必须计算出目标地址，并判断以该目标地址为起始的基本块是否已经翻译。由于 QEMU 系统为单进程设计，基本块之间切换本来就不存在上下文的切换开销，因此最主要的开销为判断 PC+1 为起始的基本块是否被翻译过。而上述的优化方法至少需要一次判断。而且，如果基本块没有被翻译，该优化思想还将再进行一次判断，这就是为什么有些测试项的结果相对不优化的情况还有下降的原因。此外该优化方法还要求 return 指令能区分返回地址。因此这种优化方法并不能显著地提高使用单进程设计的翻译系统的性能。

4.6 总结与讨论

经过对 QEMU 动态二进制翻译系统中所对直接跳转和间接跳转的处理进行不同的优化，发现可以通过连接已翻译的基本块来提高系统的性能。其中对直接跳转的优化被证明的确可以减少系统在基本块切换间的开销，经实验发现可以产生

40％以上的性能提高。对于间接跳转优化由于受到系统翻译机制的限制而未能获得性能提高。

下一步，本人计划对翻译机制进行改进。将单进程的翻译系统改为双进程，一个进程负责翻译，另一个负责执行。由此产生的进程切换的开销可以通过以下办法来减少：把判断下一个基本块是否已经被翻译的操作安排在执行进程里面。如果下一个基本块已经被翻译，则直接跳入执行该基本块，从而避免了大量的进程间切换。只有在下一个基本块没有被翻译的情况下才要切换到翻译进程。同时还可以对 T-cache 的管理方法进行优化，以进一步跳高二进制翻译系统的性能。

第五章 二进制翻译中的寄存器优化

5.1 引言

动态二进制翻译系统动态地翻译目标机指令来维护虚拟的目标机资源状态，其中目标机资源都被映射到宿主机资源上。一个非常普遍的问题是：不同的体系结构拥有不同的寄存器组，他们所拥有的寄存器数量并不一致，通常 CISC 机的寄存器要比 RISC 机少得多，而且寄存器的使用方法也差别比较大。因此二进制翻译系统通常采用内存来虚拟目标机寄存器，然而这样会大大增加访存频率，降低了动态二进制翻译的性能。把目标机寄存器映射到宿主机寄存器，是一个非常行之有效的办法。

与此不同的是，QEMU 首先把系统翻译机制中引入的中间变量映射到宿主机寄存器上。通过实验我们发现，中间变量的使用频率要比目标机寄存器的高很多，QEMU 的做法可以带来更高的性能提高。但是本人认为 QEMU 的翻译机制引入的中间变量并不是必须的，它恰恰成为了影响动态二进制翻译性能的因素。为此我们提出了优化 QEMU 的方案。

5.2 QEMU 中的寄存器处理

5.2.1 不同的寄存器

本人在试验时，使用 arm 作为源平台，x86 为目标平台。Arm 的典型的 RISC 体系结构，而 x86 是 CISC 体系结构，二者的寄存器有着明显的不同。

X86 的寄存器：

标准的 X86CPU 包括八个通用寄存器，4 个段寄存器和 IP 跟 FLAGS 两个标志寄存器，32 位或者 64 位的 X86 的寄存器都是在这个基础上扩展得到的。通用寄存器除了一般的运算用途之外，各自有特殊的隐含用途。比如说 SP 时堆栈指针，CX 时循环计数等。这种设计可以减少一些指令的长度，因为使用这些寄存器的特殊用途时，指令不必再指出这些操作数。二进制翻译器必须能识别这些特殊用

途，否则就无法完成源机器程序的语义。

ARM 的寄存器：

ARM 处理器共有 37 个寄存器。其中包括：31 个通用寄存器，包括程序计数器(PC)在内。这些寄存器都是 32 位寄存器。以及 6 个 32 位状态寄存器。但目前只使用了其中 12 位。ARM 处理器共有 7 种不同的处理器模式，在每一种处理器模式中有一组相应的寄存器组。任意时刻(也就是任意的处理器模式下)，可见的寄存器包括 15 个通用寄存器(R0～R14)、一个或两个状态寄存器及程序计数器(PC)。在所有的寄存器中，有些是各模式共用的同一个物理寄存器；有一些寄存器是各模式自己拥有的独立的物理寄存器。表 5-1 列出了各处理器模式下可见的寄存器情况。

表 5-1 ARM 的寄存器组

Table 5-1 Register Set of ARM

用户模式	系统模式	特权模式	中止模式	未定义指令模式	外部中断模式	快速中断模式
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_inq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_inq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR SPSR_svc	CPSR SPSR_abt	CPSR SPSR_und	CPSR SPSR_inq	CPSR SPSR_fiq

5.2.2 翻译时的寄存器处理

二进制翻译系统要完成两部分工作：虚拟目标机的状态，包括寄存器、内存等信息，通常用宿主机的一段内存空间来表示；利用宿主机上的一些指令来完成目标机指令的功能，并通过这些指令用来改变虚拟的目标机状态。由于在不同的体系结构下面，寄存器的数量和用途都相差非常大，所以一般的二进制翻译系统，都把寄存器映射到宿主机的内存中（称作虚拟寄存器）。在目标机中寄存器的访问速率要比内存高很多，正因为这样寄存器的使用频率也非常高。而在一般的二进制翻译系统当中寄存器其实只是一个普通的内存变量，这样一来目标机对寄存器的高频访问被二进制翻译系统翻译成对宿主机内存的高频访问，大大增加了 load/store 的开销。所以说虚拟寄存器的做法虽然比较简单，但是大大影响了二进制翻译的性能，同时宿主机寄存器也不能得到充分的利用。所以说把目标机寄存器映射到宿主机寄存器，成为了非常有潜力的优化方法。然而并不是所有宿主机的寄存器都能够用于映射目标机寄存器，宿主机中能被利用的寄存器数量可能超过目标机寄存器，也可能小余目标机寄存器数量。

QEMU 首先把系统翻译机制引入的中间变量映射到宿主机寄存器上，只有当宿主机寄存器足够多的情况下才映射目标机寄存器。通过分析 QEMU 的翻译机制我们发现，中间变量的使用频率要比目标机寄存器的高很多，所以说 QEMU 的做法是完全有依据的，从实验的结果来看 QEMU 的做法可以带来 3 到 5 倍的性能提高。而做目标机寄存器映射却只能得到 5%左右的提高。但是本论文认为做目标机寄存器映射无法大幅提高性能是主要受限于 QEMU 的特有的翻译机制。在 QEMU 的翻译机制中目标机寄存器，都要先拷贝到中间变量，操作完成后再存回目标机寄存器。但是从二进制翻译技术的角度来看，引入这些中间变量并不是必须的，恰恰相反，它们成为了影响动态二进制翻译性能的因素。为此我们提出了一种新的二进制翻译方案，有望能进一步提高 QEMU 性能[84]。

5.3 优化方案设计与实现

QEMU 是一个开源的通用 CPU 模拟器。通过采用可移植的动态二进制翻译技术，QEMU 实现对多种 CPU 的模拟，并获取较高的模拟速度。QEMU 用 C 语

言设计一系列的微操作，这些微操作都在编译 QEMU 的时候，编译成了宿主机的二进制代码。而在运行目标机二进制代码的时候，QEMU 先把各个目标机器指令分成若干个微操作，然后再把这些编译后的微操作连接起来。

QEMU 在翻译过程中设定了中间层代码，用于两种体系结构之间的过渡。这种中间层代码由一整套微操作组成，这些微操作由 C 语言实现，在编译整个系统时被编译成目标机器的二进制码。这些微操作码在实现过程中大量使用了目标机器的寄存器，这种设计方法使得 QEMU 对寄存器的使用较为频繁，要进行优化也较有难度。于之前所说的将源机器寄存器映射到目标机器上的构思不同，QEMU 使用的方式是通过 GCC 静态寄存器扩展变量（register）把翻译机制中引入的中间变量以及源机器状态的基址（*ENV）映射到目标机寄存器上。具体实现语句如下：

register struct CPUARMState *env asm(“ebp”);

register uint32_t T0 asm(“ebx”);

register uint32_t T1 asm(“esi”);

register uint32_t T2 asm(“edi”);

QEMU 用一个大的数据结构 env 来保存目标机器的状态，比如说我们关心的目标机寄存器。目标机的机器指令，被翻译成一组宿主机的机器指令，通过这些指令来改变 env 中的目标机资源。目标机的所有资源都通过*ENV 基址加上特定的偏移来访问，所以*ENV 的访问率非常高，。翻译后的代码都以*ENV 里面的内存变量作为操作对象，和当年引入通用寄存器一样，QEMU 引入了三个中间变量，并把这些中间变量映射到宿主机寄存器上。*ENV 中的变量都被先调入这些中间变量，然后再进行操作，最后把操作结果写回 env。比如说一条 ARM 指令：add r3 r3 r2 将被分成以下微操作，T0=r2；T1=r3；T0＝T0＋T1；r3=T0；T0，T1 作为其中的中间变量。QEMU 采用这些中间变量还可以带来其它方面的优势：比如说可以大大减少微操作的数量[7]，进而简化翻译过程，提高代码的重用率并减少代码量。所有操作的对象都被限制在 T0、T1、T2，而不像一般的机器，指令是操作码和操作数的组合，其中操作数又有多种寻址方式。

ebp 在 X86 中本来为帧指针，QEMU 通过使用-fomit-frame-pointer 编译选项，使得程序不再使用 ebp 来维护堆栈，这种做法有两个好处，一是有利于

微操作之间的无缝连接，即每个微操作片断之间不再传递数据，二是能够腾出一个寄存器用于优化设计。在 QEMU 中 ebp 用于存放*ENV，在 QEMU 的整个系统（包括翻译和执行两个部分）中，都不会使用 ebp，所以存放在 ebp 的*ENV，不必担心被程序非法修改。而在一次执行中*ENV 的值也正好一直保持不变。此外 QEMU 还使用了 X86 上的可分配的寄存器[50]：ebx，esi 和 edi，分别映射微操作中的中间变量 T0，T1，T2。

我们实验的目的是为了证明 QEMU 把翻译机制中引入的变量映射到宿主机寄存器，能提高翻译器的性能，并且这种方案比映射目标机寄存器的更加有效，更加合理。为此我们以 QEMU（arm on x86）作为实验平台，设计了四种映射方案，如图 5-1 所示。第一种方案没有利用任何宿主机寄存器，中间变量和寄存器都是直接放在内存中；第二种方案并不映射中间变量和目标机状态基址，而是映射了 ARM 中使用相对比较频繁的 R0，R11，R13，R15。第三种方案我们映射了目标机状态基址和 R0，R11，R15；第四种方案就是 QEMU 目前的做法，把引入的中间变量和目标机状态基址映射到宿主机寄存器中。如图 5－1 所示。

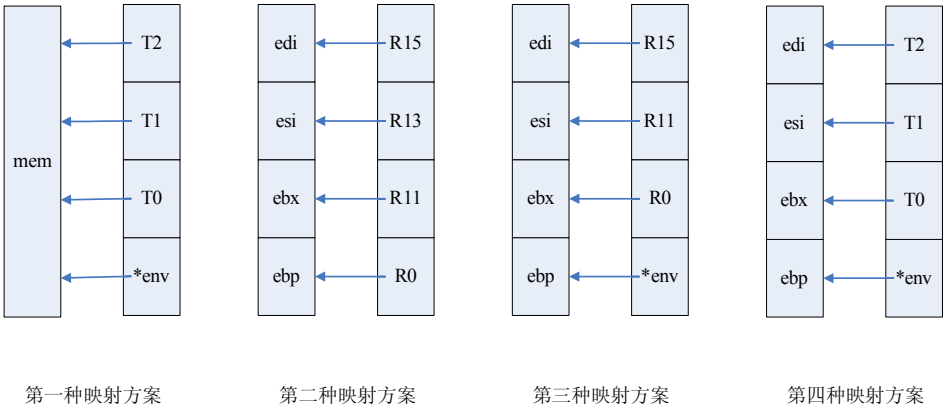

图 5-1 四种寄存器映射方案

Figure 5-1 Four Register Mapping Plans

5.4 实验结果及分析

为了对比各种映射方案对 QEMU 性能的影响，我们在 QEMU（ARM on X86 user level）上运行 nbench benchmark suite[2]（QEMU 官方网站推荐用于评测性能的测试程序）。所有的执行速度（Iterations/sec）都是五次测试的算术平均值。其中 nbench 采用 Crosstool[3] 提供的编译器“arm-unknown-linux-gnu-gcc”

(gcc-2.95.3-glibc-2.1.3, CFLAGS = -s -static -Wall -O3)编译。QEMU 运行在轻负荷的平台上，Cpu: Pentium 4 3.06G HyperThread, 内存: 1G, 操作系统: redhat9。按照我们的分析，第一种方案没有利用宿主机上的寄存器，其它优化方案都应该获得比它更高的性能，所以我们把第一种方案作为参考，图 5-2 中所有数据都是相对于它的百分比比较数据。

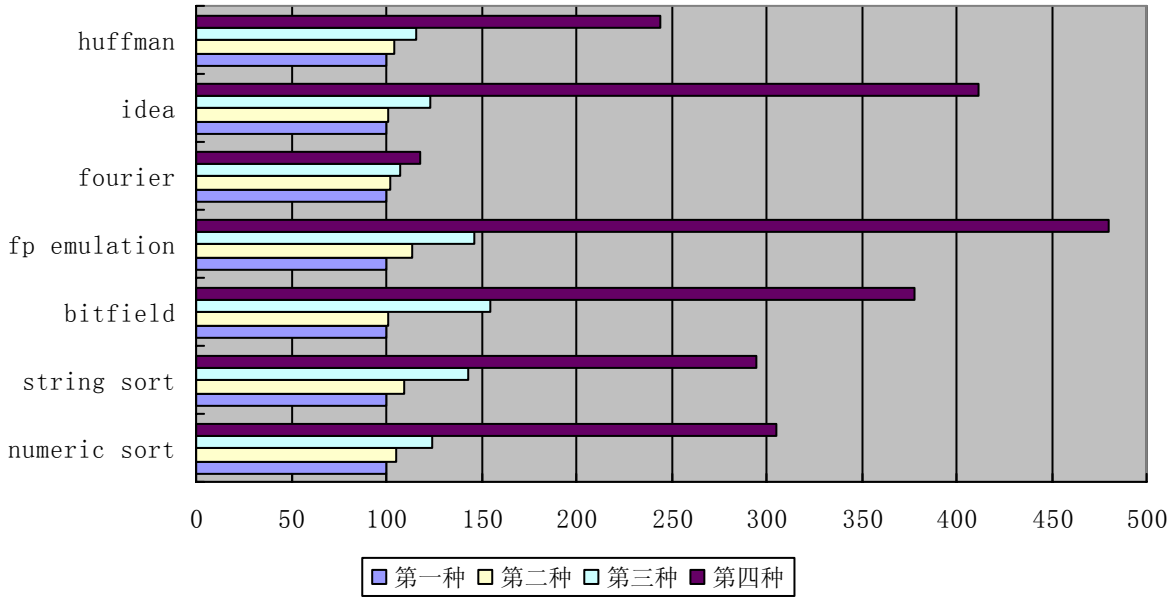

图 5-2 不同映射方案对性能的 QEMU 性能的提高

Figure 5-2 Speedup of QEMU in Different Mapping Strategies

虽然各种方案对 QEMU 性能的提高程度在不同的 benchmark 之间存在波动。但是通过分析实验结果我们可以推断出以下的结论：相对于第一种方案而言，第四种方案也就是 QEMU 现在所采用的方案能获得最为显著的性能提高，大多数 benchmark 都能达到 3 至 5 倍。而在 QEMU 现有的翻译机制下如果把目标寄存器映射到宿主机寄存器上（第二种方案），性能的提高微乎其微，只在 5%附近。第三种方案的实验结果告诉我们 env 的基址被访问的频率要远远高于寄存器，把它映射到 ebp 寄存器将获得 30%左右的性能提高。对比第三种方案和第四种方案可以看出，是否映射 QEMU 翻译机制中的中间变量对性能的影响最为明显。

实验的结果告诉我们，在 QEMU 中选择 T0、T1、T2 和*env 映射到宿主寄存器是完全正确的，明显优于选择目标机寄存器。这主要有以下两个原因：

首先，选择的目标寄存器的使用率并不一定非常高，而按照 QEMU 的翻译机制 T0、T1、T2 和*env 却被非常频繁的使用；此外即使选择的目标寄存器被使

用到了，QEMU 依旧要先把这些寄存器的值调入到 T0、T1、T2，操作完毕后再保存为目标寄存器，显然这是多此一举的。表 5-2 中列举出了在各种方案下，两条普通 arm 指令对应的翻译后 x86 指令组的访存次数。从表中可以看出，虽然我们映射了 R0 R11 R13 R15，但并不是说这四个一定会被使用到，针对这两条指令，就只命中了 R11，在这种情况下对系统的提升几乎微乎其微，这可以非常好的解释第二种方案对性能影响不大的原因。第五种方案选择 r2、r3、r11、r15 映射到宿主机寄存器，使得两条指令中遇到的目标寄存器都被命中，即使在这种情况下，翻译后指令的访存数量还是远远超过第四种方案。

ARM 指令			add r3,r3,r2		
微操作	T1＝r2	T0＝r3	T0＝T0+T1	r3=T0	合计
第一种	3	3	3	3	12
第二种	3	3	3	3	12
第三种	2	2	3	2	9
第四种	1	1	0	1	3
第五种	1	1	3	1	6
ARM 指令			str r3,[fp,#-16]		
微操作	T1=r11	T1=T1-16	T0=r3	(T1)=T0	合计
第一种	3	2	3	3	11
第二种	1	2	3	3	9
第三种	1	2	2	3	8
第四种	1	0	1	1	3
第五种	1	2	1	3	7

表 5-2 不同映射方案的访存次数

Table 5-2 Memory Operations in Different Mapping Strategies

其次，根据 QEMU 的翻译机制，所有源机器寄存器都要先加载到中间变量才能做运算操作，即使它们已经被映射到宿主机寄存器。源机器寄存器和中间变量之间的冗余拷贝操作严重限制了映射源机器寄存器优化的效果。表 4-1 中的第五种方案，选择了 R2，R3，R11，R15，两条 ARM 指令中出现的所有寄存器都被命中。按照寄存器映射优化思想，这种情况下应该不需要任何访存操作。然而 QEMU 仍然需要 13 次，此外这些冗余的拷贝操作也是 QEMU 翻译的代码膨胀率居高不下的一个重要原因。代码膨胀率是二进制翻译中一个重要性能指标。QEMU（ARM to X86）翻译 nbench 的平均代码膨胀率为 5.8，从 ARM 到 X86 的翻译属于 RISC 到 CISC 的翻译，X86 的代码密度要远远小于 ARM 的代码密度，从

理论上说从 ARM 到 X86 翻译的代码膨胀率应该控制在 1 以下。高代码膨胀率将导致在翻译阶段增加代码的拷贝开销，浪费 T-Cache 空间。在执行阶段增加宿主机指令的取指令和译码等开销，其中取指令同样需要访存。总之 QEMU 引入的中间变量削弱了映射源机器寄存器的优化效果。

5.5 总结与讨论

经过对 QEMU 的翻译机制的分析，我们解释了在目前 QEMU 的翻译机制下，首先选择中间变量做映射的的确是最为高效的方案。并通过实验证明选择中间变量做映射相对于其他映射方法将获取 3 到 5 倍的性能提高，而选择目标机寄存器的只能带来微乎其微的提高。

为了进一步改善翻译性能，我们提出了新的翻译机制修改方案。我们认为从动态二进制翻译技术的角度来说，这些中间变量并不是必须的，而且也正是因为这些中间变量的引入，动态二进制翻译性能受到了一定的影响。

我们以 arm 指令 add r3 r3 r2 为例，按照 QEMU 的做法翻译后的 X86 代码将要进行三次访存。如果修改 QEMU 的翻译机制，不再使用中间变量，那么该指令的中间代码用 c 语言表示为 r3+=r2。在 X86 的宿主机上将使用易变寄存器[9]eax 作为中间变量，编译成两条机器代码 eax=0x8(env), 0xc(env)=eax+0xc(env) 和 QEMU 一样要三次访存。但是我们可以充分使用 x86 宿主机的可分配寄存器，把目标机器的寄存器映射到宿主机上，如果命中 r2 就只需要两次访存，如果命中 r3 则仅需要一次访存，如果两者都命中则不需要访存。此外我们还可以采用类似于 Bintrans 的动态分配技术，来提高目标寄存器的命中率[5]。由此可见通过修改 QEMU 的翻译机制，完全可以进一步提高它的性能。

第六章 二进制翻译中的基本块覆盖优化

6.1 引言

动态二进制翻译器在翻译，执行源机器的二进制程序代码时，收集 Profile 信息，根据 Profile 信息对执行频率较高的基本块或热路径进行翻译和优化，生成高效的宿主机代码并存入到 T-Cache 中。当程序再次执行到这个基本块或热路径时就执行 T-Cache 中相应的代码块，从而避免重新翻译并提高程序执行速度。但是 T-Cache 毕竟空间有限，不可能无限制增大，因此，必须尽量充分利用 T-Cache 空间，以提高动态二进制翻译器的性能。

如 3.2 节所言，QEMU 系统选择基本块作为基本翻译单位，在基本块之间采用类似解释器的处理方式，而在基本块内部采用类似静态二进制翻译器的处理方式。即在基本块内部，一次翻译整个基本块内所有指令再运行，而在基本块之间每翻译一个基本块块就执行一个（热路径除外，情参考 4.3 节）。在 QEMU 中每个基本块是以其起始指令的 pc 作为唯一标识的。这种设定就产生了基本块覆盖的可能，即有的基本块可能是其他基本块中的一部分。这样在翻译是等同与对同样的代码段进行了两次甚至多次的缓存，极大的占用了 T-Cache 空间。如果能够消除这种现象，就能极大的提高 T-Cache 空间的利用率，同样可以提高 QEMU 系统的性能。

本人研究了 QEMU 动态二进制翻译系统中所存在的两类基本块覆盖现象，然后对其进行了消除优化，并通过实验数据说明了优化的效果与其中存在的问题，并提出改进方法。

6.2 QEMU 中的基本块覆盖

在 QEMU 中，每个基本块是以其起始指令的 pc 作为唯一标识的。由于不同的起始点可能有同样的结束位置，这样就造成了两个不同的基本块相互包含的情况，如图 6-1 所示的情况。

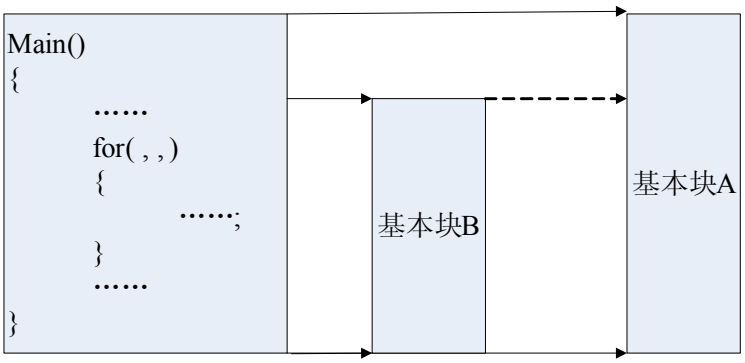

图 6-1 基本块覆盖的产生

Figure 6-1 The Emergence of Overlapping

6.2.1 基本块覆盖的分类

QEMU 中的基本块覆盖主要有两类情况，第一类是新遇到的基本块是某个已经被翻译过并储存在 T-cache 中的基本块的一部分。如图 6-1 所示，如果程序运行流程为先遇到 pc1，则翻译从 pc1 到 jmp 的指令，生成 BB1，并把 BB1 放在 T-Cache 中。然后程序从某处跳到 pc2，由于翻译器无法找到 pc2 开头的基本块，于是就将 pc2 将作为是另一个基本块的开始，重新翻译从 pc2 到 jmp 的指令，生成 BB2，并把 BB2 放在 T-Cache 中。显然 pc2 到 jmp 的指令被重复翻译了，同样的目标机代码在 T-Cache 中占据了两倍的空间。同样的道理，如图 6-2 所示，如果翻译器先遇到了 pc2，那么它将翻译 pc2 到 jmp 的指令，生成新的基本块 BB2，并存入 T-Cache 中。在程序流程来到 pc1 是，翻译器由于无法找到翻译了的 pc1 开头的基本块，于是从 pc1 开始翻译。由于翻译器只有在碰到 jmp 的时候才能当作基本块的结束，于是 pc2 到 jmp 的指令就又被翻译了一遍，生成 BB1。同样这部分的翻译生成代码在 T-Cache 中也有两个拷贝。

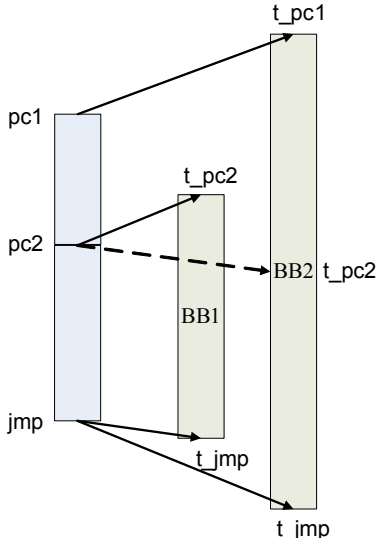

图 6-2 两种相互覆盖

Figure 6-2 Two Kinds of Overlapping

通过上面的分析，我们可以看出，同样的一个程序段，根据程序执行顺序的不同，将出现两种不同的相互覆盖情况，我们把前者归类为第一种相互覆盖，后者归类为第二种相互覆盖。上面的例子只是针对只有两个入口的基本块，如果一个基本块具有多个入口，情况将变的更加复杂，两种覆盖还可能同时出现。

6.2.2 基本块覆盖出现的概率

现在已经明确，按照现行的翻译机制会出现基本块相互覆盖的现象，如果不消除这些基本块之间的相互覆盖，那么相互覆盖的部分代码将被两次或多次翻译，翻译产生的代码在 T-Cache 中也对应地出现多个拷贝，浪费了宝贵的 T-Cache 空间。去除相互覆盖的基本块，既可以减少翻译时间，又能节约 T-Cache 空间，从而提高二进制翻译器的性能。但由于优化本身是有一定额外性能开销的，所以这种优化是否有意义取决于基本块覆盖现象出现的概率。

为了获得这个信息，我们先修改了 QEMU 的源代码，加入一些统计量，然后上运行 nbench benchmark suite。通过新加入的统计量统计覆盖的代码长度和所占的百分比。图 6-3 和图 6-4 中列举了被覆盖的源机器代码长度和百分比，对于翻译生成的宿主机代码也做了类似的统计。从中可以看出，每一个 benchmark 均有一定数量的代码被覆盖。第一种覆盖的代码在源机器代码中的平均百分比为 6.73%，特别是“fp-emulation”benchmark，达到了 16 .41%。而第二种覆盖的平

均百分比也达到了 3%以上。和第一种覆盖相似，覆盖百分比在不同 benchmark 之间存在一定的浮动，其中“String Sort”benchmark 达到最高的 6.93%。在这些统计数据的基础上，我们可以得出结论：消除基本块之间的覆盖，将对动态二进制翻译器性能有举足轻重的影响，实现这种优化是非常有必要的。

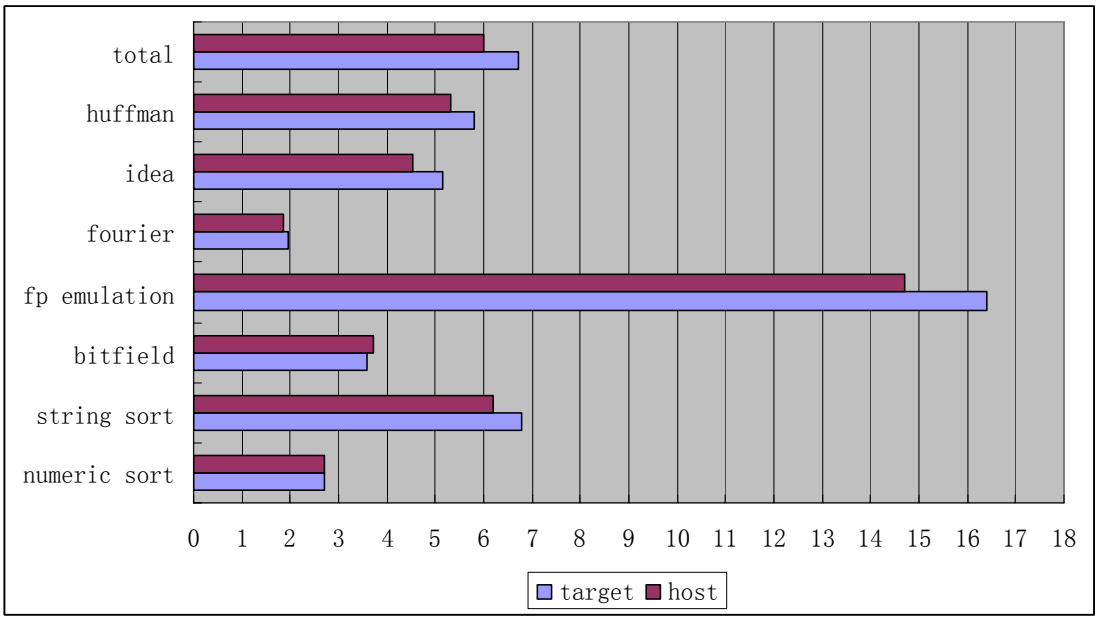

图 6-3 第一种相互覆盖的统计

Figure 6-3 Statistic of First Class Overlapping

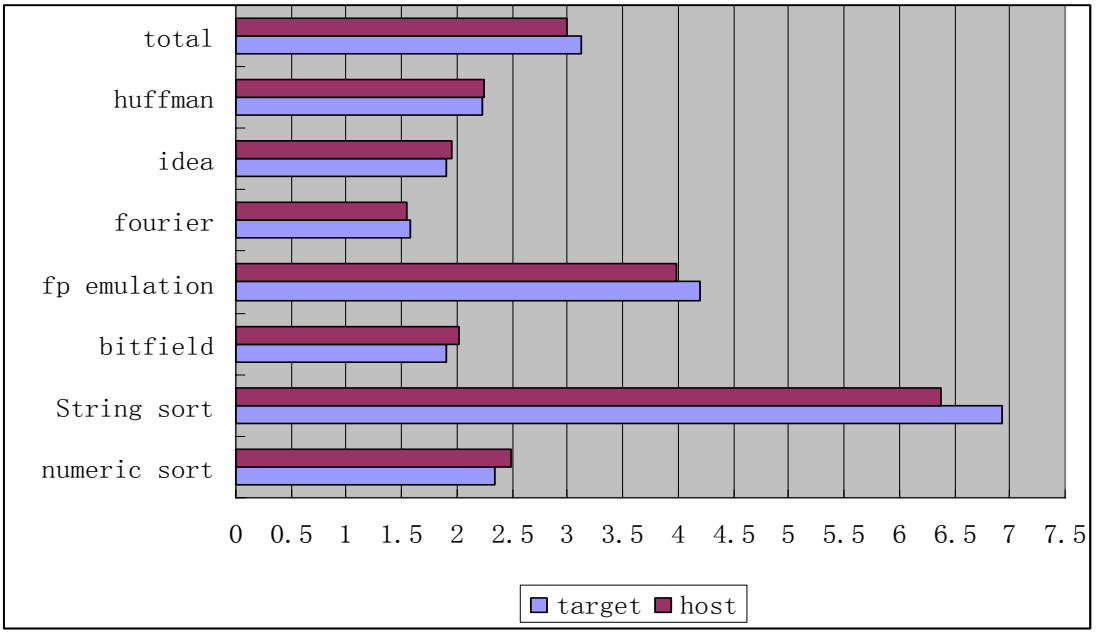

图 6-4 第二种相互覆盖的代码统计

Figure 6-4 Statistic of Second Class Overlapping

6.3 优化方案设计与实现

优化方案的主要思路是在翻译基本块之前检查即将翻译的基本块是否在T-cache 中全部或部分存在，如果存在则采取措施避免重复翻译。由于两类基本块覆盖现象的特征有较大差异，所以各自所用的检测方式也有所不同。

6.3.1 第一类基本块覆盖的优化

第一类基本块覆盖指，准备翻译的基本块有可能是一个已经翻译过的基本块的后半部分。在检测时需要判断即将被翻译的基本块的第一条指令是不是位于一个已经被翻译过的基本块之内，即图6-2中的pc2。根据QEMU的实现机制，所有翻译后的基本块的基本信息都被保存在一个数组中，其中包括了基本块的第一条指令的PC，和基本块的长度size。我们可以遍历这个数组，如果其中存在一个BB1（假设第一条指令PC为pc1），满足条件：pc1<pc2<(pc1+BB1->size)，那么就说明BB2就是BB1的后半部分，也就是说检测出了第一类基本块覆盖。

当确定第一类基本块覆盖发生的基本块后，即BB1，就需要确定pc2在BB1中的位置。在QEMU系统中，每条源机器指令被解释成为几条微操作的组合，每部微操作由c语言实现，并编译成为目标机器代码，因此每条源机器指令被翻译成目标机指令后对应代码的长度不一致，所以我们只能将从pc1到pc2之间的所有指令翻译后的代码长度进行累加，得到t_pc2相对t_pc1的偏移，如图6-5所示。

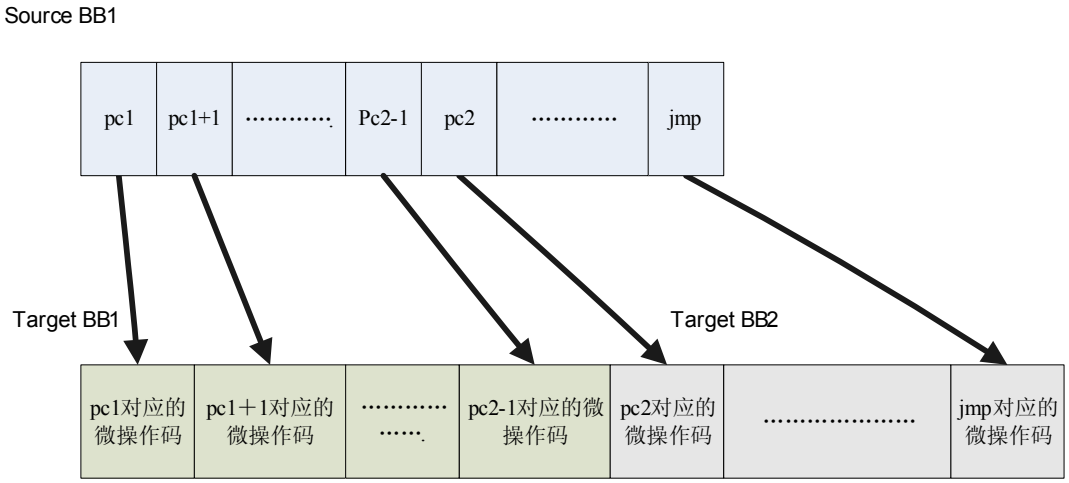

图 6-5 第一类基本块覆盖的消除

Figure 6-5 Elimination of Kind One Overlapping

在确定 t_pc2 后，我们只需在保存基本块基本信息的数组中增加一条以 pc2 为起始指令地址，长度为 BB1->size-(t_pc2- t_pc1)的基本块信息即可，其对应 T-cache 中的地址为 t_pc2。

6.3.2 第二类基本块覆盖的优化

第二类基本块覆盖指，准备翻译的基本块有可能包含一个已经翻译过的基本块，即已经翻译过的某个基本块是其的后半部分。在检测时需要判断即将被翻译的基本块中是否有那条指令是某个已经翻译过的基本块的起始指令，即图 6-2 中的 pc2。假设目前需要翻译的基本块为 BB1，其第一条指令 PC 为 pc1，我们需要遍历保存已翻译基本块信息的数组，如果存在一个如果其中存在一个 BB2（假设第一条指令 PC 为 pc2），满足 pc2 位于目前需要翻译的基本块之中，那么就说明 BB2 就是 BB1 的后半部分，也就是说检测出了第二类基本块覆盖。由于这项检测需要对 BB1 中的每一条指令进行一次，因此性能消耗较大。

当确定第二类基本块覆盖发生后，我们首先翻译 pc1 到 pc2-1 的所有指令，然后利用直接跳转优化中的思想，在翻译后的 pc2-1 代码后添加一条目标机上的跳转指令，目标为翻译后的 pc2 代码所在位置，即 t_pc2 所指位置，如图 6-6 所示。然后将这个新基本块存入 T-cache 中，并添加进入基本块信息数组。

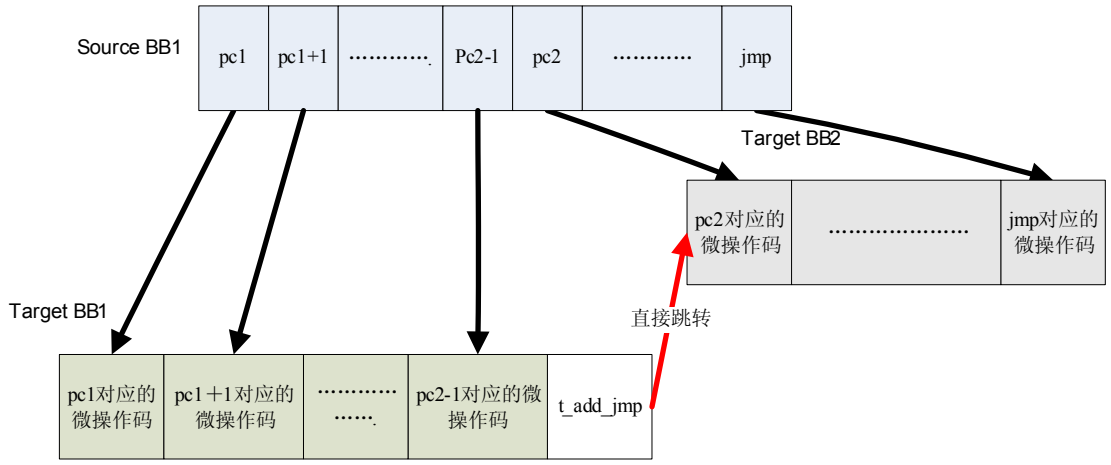

图 6-6 第二类基本块覆盖的消除

Figure 6-6 Elimination of Kind Twe Overlapping

6.4 实验结果及分析

实验的主要目的是证明消除基本块间相互覆盖的优化方法将有效的提高 QEMU 的性能，为此我们在用户模式的 QEMU（ARM 到 X86）上运行 nbench benchmark suite。所有的执行速度（Iterations/sec）都是五次测试的算术平均值。其中 nbench 采用 Crosstool 提供的编译器“arm-unknown-linux-gnu-gcc” (gcc-2.95.3-glibc-2.1.3, CFLAGS = -s -static -Wall -O3)编译。QEMU 运行在轻负荷的平台上，Cpu: Pentium 4 3.06G HyperThread, 内存: 1G, 操作系统: redhat9。为了能明显地表示优化对 QEMU 性能的影响，我们在原始 QEMU 上运行 nbench。在图 6-7 中，所有其他数据都是相对于原始的 QEMU 而言的，包括第一种优化方案（消除第一种相互覆盖），第二种优化方案(消除第二种相互覆盖)和他们之间的组合。

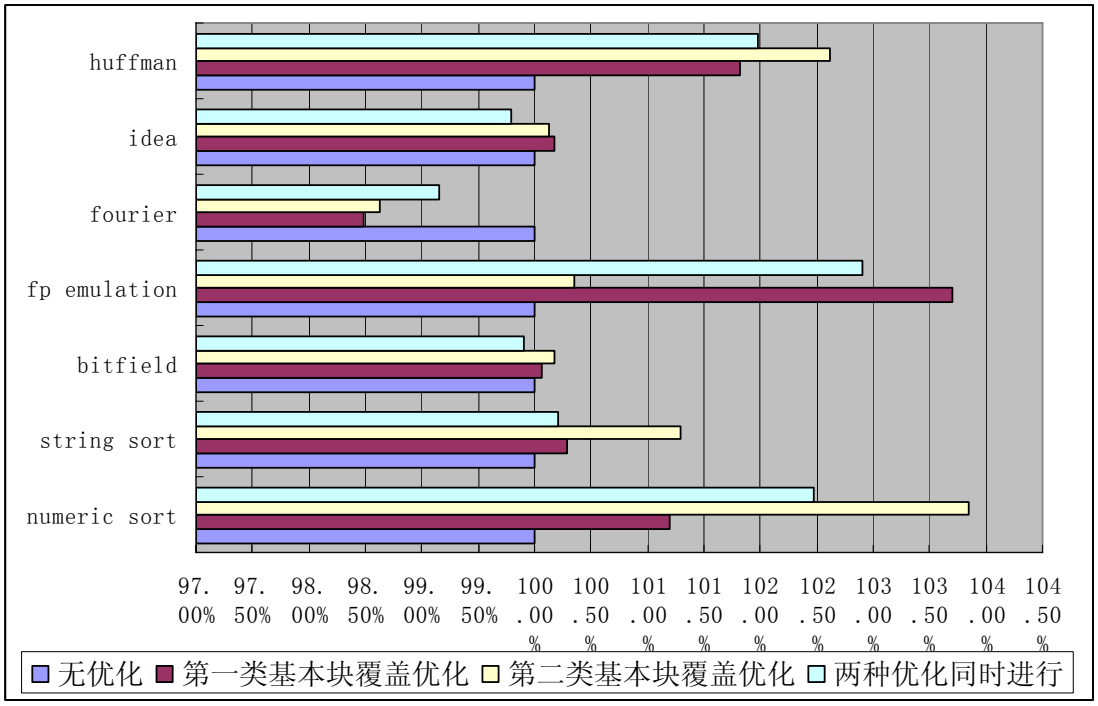

图 6-7 不同的优化方案对 QEMU 的性能提高

Figure 6-7 Performance Improvement of Different Optimizations

实验数据表明，优化的效果尽管在不同的 benchmark 之间存在一些浮动，但大部分测试项目中无论是消除哪一种相互覆盖的优化方法能够带来 1%到 4%的性能提高，可见我们做的优化是有实际意义的。对于测试中所发生的性能下降的项目，对比之前概率统计可以发现是基本块覆盖现象发生较少的项目，因此在这

个项目上，优化所带来的好处不足以弥补优化自身产生的开销，因此反而造成性能下降。

6.5 总结与讨论

经过对 QEMU 动态二进制翻译系统中两类基本块覆盖现象的优化，发现可以通过消除基本块覆盖来较少重复翻译和提高 T-cache 的利用率，进而提高系统的性能。实验证明在大部分情况下对基本块覆盖的消除是可以提高系统性能的，但在某些情况下，由于基本块覆盖现象较少而造成系统性能下降。

下一步，本人计划对简单的优化过程进行改进，希望能够较少优化自身的性能开销，以进一步提高系统整体性能。

第七章 结束语

本论文致力于在动态二进制翻译优化技术的研究与实现。论文得到“国家重大基础研究（973）前期研究专项”支持。作者在该项目中动态二进制翻译优化组中从事研究工作，主要负责项目的前期理论积累工作以及项目研究方案的确立，并实现多中动态二进制翻译优化技术。从 2005 年 7 月起，就对二进制翻译中涉及到的核心技术以及现有系统进行了深入的研究，其中主要包括计算机仿真与模拟技术、编译与反编译技术、计算机体系结构以及二进制翻译和虚拟机技术等。

我们把本文的贡献总结如下:

1、 研究了目前二进制翻译领域的典型翻译系统，详细研究了动态二进制翻译系统QEMU的翻译机制、运行方式、翻译策略，并使用其用户级系统作为我们的实验平台。

2、 QEMU动态二进制翻译系统实验平台上，针对翻译过程每个基本块运行结束之后过于频繁的状态切换和判断提出跳转控制优化方案。即在明确跳转目标时判断目标基本块是否已经被翻译过，如果当系统发现跳转目标基本块已经被翻译时，可以直接把跳转的源基本块和跳转目标基本块在T-Cache中连接起来，从而降低了动态翻译系统自身的开销。

3、 针对QEMU动态二进制翻译系统中将中间变量映射到宿主机寄存器上的翻译机制，对寄存器的不同映射方案进行了性能测试。发现了在目前翻译机制下，中间变量的确是使用最为频繁，最有价值映射的部分。最后提出了取消中间变量的新翻译机制设想。

4、 针对QEMU动态二进制翻译系统中每个基本块以头指令pc作为唯一标识的方式，发现了基本块覆盖的存在。即基本块可能是某个基本块的一部分，也有可能包括一些基本块。对此提出了减少基本块覆盖现象的方案，并且将其实现。实验数据表明优化方案的确可以提高系统的整体性能。

由于本人的研究深度和广度是十分有限的，因此很多问题都需要进一步的深入研究，系统的实现也有待进一步完善。后面的工作将主要体现在以下几个方面：

1. 进一步加强编译原理方面的知识储备，特别是动态编译技术，对 QEMU 翻译生成的宿主机代码，进行重优化，进一步提高二进制翻译系统的性能。
2. 进一步了解不同体系结构的设计细节，征对不同体系结构之间的翻译提出高效、直接的方法，并且充分发挥宿主机的优势。

将来的工作希望能继续提出一些创造性的动态二进制翻译技术，并开始着眼于虚拟机领域的研究，拓宽它们在计算机安全领域的应用。

参考文献

1. E. R. Altman, D. Kaeli, and Y. Sheffer, "Welcome to the opportunities of Binary Translation", IEEE Computer 33(3), March 2000
2. R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson, “Binary translation”, Communications of the ACM, 36(2):69-81, February 1993.
3. Apple Corporation. Macintosh application environment. http://www.mae.apple.com/, 1994.
4. Digital. Freeport express. http://www.digital.com/amt/freeport/, 1995.
5. K. Andrews and D. Sand, “Migrating a CISC computer family onto RISC via object code translation”, In Proceedings ASPLOS V, pages 213–222, Oct. 1992.
6. R.J. Hookway and M.A. Herdeg, “Digital FX!32: Combining emulation and binary translation”, Digital Technical Journal, 9(1):3-12, 1997.
7. V. Bala, E. Duesterwald, S.Banerjia, “Transparent Dynamic Optimization: The Design and Implementation of Dynamo”, HP Laboratories Cambridge, HPL-1999-78, June, 1999
8. K. Ebcioglu, E. R. Altman, “DAISY: Dynamic Compilation for 100 percent Architectural Compatibility”, Proc ISCA24, New York: ACM Press, 1997: 26-37
9. M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak, D. Applenzeller, “Dynamic and Transparent Binary Translation”, Computer, IEEE Computer Society Press, 2000: 33(3): 54-59.
10. D. R. DITZEL, “Transmeta’s Crusoe: Cool chips for mobile computing”, In Hot Chips 12: Stanford University, Stanford, California, August 13–15, 2000 (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2000), IEEE, Ed., IEEE Computer Society Press.
11. C. Cifuentes, M. V. Emmerik, and N. Ramsey, “The design of a resourceable and retargetable binary translator”, In Working Conference on Reverse Engineering (WCRE'99), pages 280--291, October 1999.
12. D. Ung, C. Cifuentes, “Machine-Adaptable Dynamic Binary Translation”, Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization. 41-51, 2000
13. C. Cifuentes and D. Ung, “Walkabout - a retargetable dynamic binary translation framework”, Technical Report TR-2002-106, Sun Microsytems Laboratories, Palo Alto, CA 94303, February 2002.
14. K. Scott, A. J. Davidson, “Strata: A software dynamic translation infrastructure”, In IEEE Workshop on Binary Translation (2001).
15. “QEMU: The open source processor emulator,” http://fabrice.bellard.free.fr/qemu/about.html
16. F. Bellard,; “QEMU, a fast and portable dynamic translator,” USENIX Annual Technical Conference, APR 10-15, 2005, USENIX ASSOCIATION PROCEEDINGS OF THE FREENIX/OPEN SOURCE TRACK : 41-46, 2005
17. “Skyeye: An integrated simulation environment in Linux and Windows” http://www.skyeye.org
18. M. Probst, "Fast Machine-Adaptable Dynamic Binary Translation", Workshop on Binary Translation 2001
19. Ebcioglu, K.; Altman, E.; Gschwind, M.; Sathaye, S.; “Dynamic binary translation and optimization,” Computers, IEEE Transactions on Volume 50, Issue 6, June 2001

Page(s):529 – 548
20. V. Bala, E. Duesterwald and S. Banerjia, “Dynamo: A Transparent Dynamic Optimization System”, Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, Vancouver, Canada, 2000.
21. D. Bruening, E. Duesterwald, and S. Amarasinghe, “Design and Implementation of a Dynamic Optimization Framework for Windows”, 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4), December 1, 2001, Austin, Texas.
22. D. Ung and C. Cifuentes. Optimising hot paths in a dynamic binary translator. In Workshop on Binary Translation, October 2000
23. M. Probst, “Dynamic Binary Translation” UKUUG Linux Developer's Conference 2002
24. M. Probst, A. Krall, B. Scholz, “Register liveness analysis for optimizing dynamic binary translation,” Reverse Engineering, 2002. Proceedings. Ninth Working Conference on 29 Oct.-1 Nov. 2002 Page(s):35 – 44
25. I. Piumarta, F. Riccardi, “Optimizing direct threaded code by selective inlining”, Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).
26. D. Williams, “Threaded Software Dynamic Translation Master’s Project” August 30.2005 unpublished material
27. “nbench: a port of the well known BYTEmark benchmark for Linux,” http://www.tux.org/%7Emayer/linux/bmark.html
28. “Crosstool: gcc/glibc cross toolchains,” http://kegel.com/crosstool/
29. K. Hazelwood and M. D. Smith. “Code cache management schemes for dynamic optimizers”. In 6th Workshop on Interaction between Compilers and Computer Architectures, pages 102–110, February 2002.
30. D. Bruening, T. Garnett, and S. Amarasinghe, “An Infrastructure for Adaptive Dynamic Optimization”, in Proceedings of the International Symposium on Code Generation and Optimization, pp. 265–275, IEEE Computer Society, 2003.
31. R. J. Hookway , M. A. Herdeg1 Digital FX !32 : Combining emulation and binary translation1 Digital Technical Journal , 1977 , 9: 3～12
32. P. Hohensee , M. Myszewski , D. Reese. “WABI CPU emulation”, Hot Chips VIII , Palo Alto , CA , 1996
33. C. Cifuentes , M. Van Emmerik. UQBT : Adaptable binary translation at low cost. IEEE Computer , 2000 , 33 (3) : 60～66
34. A. Klaiber. “The technology behind Crusoe processor”, Transmeta Corporation , Tech Rep , 2000
35. Y.Y. Tsai, “ A Software Design of Binary Translation System,” A thesis for degree of Master of Science in Electrical Engineering at the National Cheng Kung University June 2003
36. A. Chernoff and R. Hookway, "DIGITAL FX!32 - Running 32-Bit x86 Applications on Alpha NT." Proceedings of the USENIX Windows NT Workshop, USENIX Association, Berkeley CA, August 1997.
37. V. Bala, E. Duesterwald, and S. Banerjia. “Dynamo: A transparent runtime optimization system,” In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00), June 2000.
38. E. Altman, M. Gschwind, and S. Sathaye. BOA: the architecture of a binary translation

processor. Research Report RC21665, IBM T.J. Watson Research Center, Yorktown Heights, NY, March 2000.
39. R. P. Goldberg, “Survey of virtual machine research”, IEEE Computer Magazine, 7(6), 1974.
40. M. Rosenblum, Tal. Garfinkel: Virtual Machine Monitors: Current Technology and Future Trends. IEEE Computer 38(5): 39-47 (2005)
41. B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. “Xen and the art of virtualization”, In Proceedings of the ACM Symposium on Operating Systems Principles, October 2003.
42. A. Whitaker, R. S. Cox, M. Shaw, S. D. Gribble “Rethinking the Design of Virtual Machine Monitors”, IEEE Computer 38(5): 57-62 (2005)
43. P. M. Chen, B. D. Noble, "When Virtual Is Better Than Real", hotos, p. 0133, Eighth Workshop on Hot Topics in Operating Systems, 2001.
44. J. E. Smith, R. Nair, “Virtual Machine Versatile Platforms for Systems and Process”, Release Date: June, 2005 ISBN: 1558609105
45. D. Larochelle and D. Evans, “Statically detecting likely buffer overflow vulnerabilities”,In Proceedings of the 2001 USENIX Security Symposium, 2001.
46. 黄英兰,杨晋兴,钟 珊, “二进制翻译系统 BATSUP 中的动态翻译器的设计与实现”, 航空计算技术 第 35 卷 第 3 期 2005 年 9 月
47. 白童心，冯晓兵，武成岗，张兆庆，“优化动态二进制翻译器 DigitalBridge”，计算机工程第 31 卷 第 10 期 2005 年 5 月
48. 谢海斌，武成岗，张兆庆，冯晓兵，“动态二进制翻译中的代码 Cache 管理策略”，计算机工程 第 31 卷 第 10 期 2005 年 5 月
49. 马湘宁 武成岗 唐 锋 冯晓兵 张兆庆，“二进制翻译中的标志位优化技术”，计算机研究与发展 42 (2) : 329～337 , 2005
50. Fischer, C. N.; LeBlanc,Jr, R. J.著，郑启龙，姚震 译， “Crafting A Compiler With C”“编译器构造 C 语言描述”机械工业出版社 2005 年 7 月第一版 P352
51. Kevin Scott, Jack Davidson， " Retargetable and Reconfigurable Software Dynamic Translation", Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, 2003.
52. Kevin Scott, Jack Davidson，"Strata: A software dynamic translation infrastructure", in IEEE Workshop on Binary Translation, 2001.
53. Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S .Bhamdwaj Yadavalli, and John Yates, "FX!32: a Profile-Directed Binary Translator", IEEE Micro, vol.18, no.2, 1998
54. Cindy Zheng and Carol Thompson, "PA-RISC to IA-64: Transparent Execution, No Recompilation," Computer, Vol.33, No.3, March 2000, IEEE Computer Society Press, pp 47-52.
55. K.Ebcioglu and E.Altman, "DAISY: Dynamic Compilation for 100 Percent Architectural Compatibility," Proc.ISCA24, ACM Press, New York, 1997, pp 26-37
56. K.Ebcioglu et al., "Execution-Based Scheduling for VLIW Architectures," Proc. Europar99, Lecture Notes in Computer Science 1685, Springer Verlag, Berlin, 1999, pp 1269-1280.
57. Michael Gschwind et al., "Dynamic and Transparent Binary Translation," Computer, Vol.33, No.3, March 2000, IEEE Computer Society Press, pp 54-59.
58. Michael Gschwind, Erik Altman, "Inherently Lower Complexity Architectures using

Dynamic Optimization," Proc. Workshop on Complexity Effective Design in conjunction with ISCA-2002, Anchorage, AK, May 2002.
59. Alexander Klaiber, "The Technology behind Crusoe Processor," Transmeta technology report, Jan.2000, pp 3-12
60. Leonid Barez, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang and Yigal Zemach, " IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium based systems", Proceedings of the 36th Annual IEEE/ACM International Symposium on Micro architecture (MICRO-36), IEEE-CS Press.
61. Altman ER. Ebcioglu, K. Gschwind, M. Sathaye S., "Advances and future challenges in binary Translation and optimization", In Proceedings of the IEEE, vol.89, no.11, Nov. 2001, pp.1710-22 Publisher: IEEE, USA.
62. C. Cifuentes and M. Van EmmeriK "UQBT: Adaptable Binary Translation at Low Cost, " Computer, Vol.33, No.3, March 2000, IEEE Computer Society Press, pp.60-66.
63. 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
64. C. Cifuentes, M. Van Emmerik, D. Ung, D. Simon and T. Waddington, "Preliminary Experiences with the Use of the UQBT Binary Translation Framework," Proceedings of the Workshop on Binary Translation, Newport Beach, Oct 16, 1999. Technical Committee on Computer Architecture Newsletter, IEEE-CS Press, Dec.1999, pp.12-22
65. C. Cifuentes, B. Lewis and D. Ung, "Walkabout - A Retargetable Dynamic Binary Translation Framework," Fourth Workshop on Binary Translation, Sep.22, 2002, Charlottesville, Virginia
66. D. Ung and C. Cifuentes, "Dynamic re-engineering of binary code with run-time feedbacks", Proceedings Seventh Working Conference on Reverse Engineering. IEEE Comput. Sec.2000, pp.2-10, Los Alamitos, CA, USA
67. D. Ung and C. Cifuentes, "Machine-Adaptable Dynamic Binary Translation," Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, Boston, USA, Jan 2000,ACM Press, pp.30-40.
68. D. Ung and C. Cifuentes, "Optimising Hot Paths in a Dynamic Binary Translator," Second Workshop on Binary Translation, Oct.19, 2000, Philadelphia, Pennsylvania.
69. Norman Ramsey and Mary F. Femandez, "Specifying Representations of Machine Instructions," ACM Trans. Programming Languages and Systems, Vol.l9, No.3, May 1997, pp.492-524
70. C. Cifuentes and D. Simon, "Procedural Abstraction Recovery from Binary Code," Technical Report 448, Department of Computer Science and Electrical Engineering, the University of Queensland, Sep. 1999
71. C. Cifuentes and S. Sendall, "Specifying the Semantics of Machine Instructions" International Workshop on Program Comprehension, Ischia, Italy, 24-26 June 1998, IEEE-CS Press, pp.126-133
72. C. Cifuentes, Mike Van Emmerik, Norman Ramsey, Brian Lewis, "Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework, " Sun Labs Tech Report TR-2002-105, 2002.1
73. Mark Probst, "Fast machine-adaptable dynamic binary translation," In Proceedings of the

Workshop on Binary Translation 2001, September 2001
74. Probst M. Krall, A. Scholz B, "Register liveness analysis for optimizing dynamic binary translation", Proceedings Ninth Working Conference on Reverse Engineering WCRE 2002. IEEE Compute, Soc. 2002, pp.35-44
75. http://www.transitives.corn/pmssroom.htm#briefs
76. Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, "Transparent dynamic optimization: The design and implementation of Dynamo", Hewlett Packard Laboratories Technical Report, HPL-1999-78, June 1999, Dallas, 173-181
77. 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
78. Java Development Kit, http://www.sun.oom/
79. B. S. Yang, S. M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman, "LaTTe: A Java VM Just-in-Time compiler with fast and efficient register allocation", In International Conference on Parallel Architectures and Compilation Techniques, Oct.1999.12
80. Seoul National University, IBM, http://latte.snu.ac.kr 1999
81. Intel ORP project, http://www.iaLel.com/research/mrl/orp
82. M. Cierniuk, G. Lueh, and J. Stichnoth, "Practicing JUDO: Java Under Dynamic Optimizations," In Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, October 2000.
83. 李增祥、管海兵、李小勇, “动态二进制翻译的优化”，计算机应用与软件
84. 管海兵、李增祥、梁阿磊，“动态二进制翻译的寄存器优化”，计算机研究与发展

致 谢

首先感谢我的导师李晓勇副教授。在我的整个研究工作和本文的撰写过程中始终给予我深入细致的指导和及时有效的建议。李老师对学生发自内心的关爱、严谨的治学态度、孜孜不倦的工作热情、以及坦然面对困难的精神，给我留下了深刻的印象，并将不断鞭策我前进。在此，我衷心地祝愿李老师工作顺利。

感谢慈父般的白英彩教授。在我两年半的学习期间内，他总是从对我的学习和生活给予关注热情的帮助。

感谢梁阿磊、管海滨副教授。你们在专业知识方面给予的指导，以及在论文结构等方面提出的建设性意见，使我受益匪浅。

感谢“动态二进制翻译的优化”项目组的所有成员，感谢李增祥、李俊、张量、包云程、陈建、林凌、陈昌鹏等同学，感谢他们给予过我的帮助。和你们在一起，我度过了研究生阶段的日日夜夜，并体会了其中的酸甜苦辣。谢谢。

感谢上海交通大学信息安全学院的学长和同学，特别是李伦、马燕、陈杰、陈玉来、黄金华和邓乐。从你们身上我学到了很多宝贵的知识。和你们的日子总是充满了快乐与轻松。虽然这段时间转瞬即逝，但它必将在我的记忆中永存！

感谢我的父母亲人！你们的理解延续了我的耐心，你们的鞭策坚定了我的决心，你们的支持永远是我最大的财富。在今后的日子里，相信我将会用我的努力使你们感到欣慰。

攻读硕士期间的科研及学术论文

1、 吴浩、管海兵、梁阿磊：用户级动态二进制翻译系统设计；计算机应用与软件；已接收
2、 吴浩、梁阿磊、李晓勇：动态二进制翻译中的跳转优化技术；四川大学学报；已接收